The Design of a Sparse Vector Processor

T.M. Hopkins

Ph.D.

University of Edinburgh

1993

Abstract

This thesis describes the development of a new vector processor architecture capable of high efficiency when computing with very sparse vector and matrix data, of irregular structure.

Two applications are identified as of particular importance: sparse Gaussian elimination, and Linear Programming, and the algorithmic steps involved in the solution of these problems are analysed. Existing techniques for sparse vector computation, which are only able to achieve a small fraction of the arithmetic performance commonly expected on dense matrix problems, are critically examined. A variety of new techniques with potential for hardware support is discussed. From these, the most promising are selected, and efficient hardware implementations developed.

The architecture of a complete vector processor incorporating the new vector and matrix mechanisms is described – the new architecture also uses an innovative control structure for the vector processor, which enables high efficiency even when computing with vectors with very small numbers of non-zeroes. The practical feasibility of the design is demonstrated by describing the prototype implementation, under construction from off-the-shelf components.

The expected performance of the new architecture is analysed, and simulation results are presented which demonstrate that the machine could be expected to provide an order of magnitude speed-up on many large sparse Linear Programming problems, compared to a scalar processor with the same clock rate. The simulation results indicate that the vector processor control structure is successful – the vector half-performance length is as low as 8 for standard vector instruction loop tests. In some cases, simulations indicate that the performance of the machine is limited by the speed of some scalar processor operations.

Finally, the scope for re-implementing the new architecture in technology faster than the prototype's 8MHz is briefly discussed, and particular potential difficulties identified.

Declaration

I hereby declare

- that this thesis has been composed by myself; and
- that the work described herein is my own, except where stated in the text.

T.M. Hopkins

22nd January 1993

Acknowledgements

Many thanks are due to my supervisor, Roland Ibbett, for his advice, patience, and friendly encouragement of this work, and for setting up and managing the ESP project. The weekly meetings of the group working on ESP have provided many useful ideas, and other members of the group have also been of great help. Ken McKinnon, in particular, carried out the Linear Programming simulation study, and through conversations with him, many useful changes have been made to the ESP architecture. On the machine itself, Rainer Thonnes has carried out the detailed circuit design, and Peter Lindsay is wiring the many thousands of connections. Two students in the Department of Computer Science, Alisdair Manning and Goh Boon Seng, contributed greatly by developing further simulations of the ESP architecture.

The author's research work was for a time supported by the Science and Engineering Research Council, who, together with High Level Hardware Ltd., have also supported the construction of the prototype machine. All other facilities for this research have been provided by the Department of Computer Science.

Table of Contents

1. Introduction **1**

1.1 Supercomputer architectures 1

1.1.1 Vector processing 2

1.1.2 Parallel processing 4

1.1.3 Using vector and parallel machines 6

1.2 Sparse matrix problems 7

1.2.1 Use of vector processors for sparse problems 9

1.2.2 Use of parallel processors for sparse problems 10

1.3 The Edinburgh Sparse Processor project 11

2. Sparse Matrix Algorithms **14**

2.1 Introduction 14

2.2 Direct solution of linear equations 15

2.2.1 Choice of pivot element 16

2.2.2 Gaussian elimination on sparse matrices 17

2.2.3 Sparse matrices with regular structure 17

2.2.4 Sparse matrices with irregular structure 18

2.2.5 Pivot choice in sparse Gaussian elimination 18

2.2.6 Pivot choice on sparsity grounds only 19

2.2.7 Size and sparsity of typical problems 20

2.3 Iterative solution of linear equations 21

2.4 Linear Programming 22

2.4.1 The simplex method 22

2.4.2 Computational steps in the revised simplex method 25

2.4.3 Size and sparsity of typical LP problems 28

2.5 Summary 29

3. Implementing Sparse Matrix Computation **31**

3.1 Introduction 31

3.2 Array storage for vectors 32

3.3 The order vector mechanism 34

3.3.1 Handling fill-in in the order vector method 36

3.4 The index/value array mechanism 37

3.4.1 In-phase scan implementations of arithmetic 38

3.4.2 Scatter/gather implementations of arithmetic 39

3.4.3 Comparison of in-phase scan and scatter/gather methods . 42

3.5 The index/value list mechanism 43

3.6 Associative memory storage 44

3.7 Other data structure candidates 47

3.7.1 Tree structures 48

3.7.2 Hash table based structures 49

3.8 Data structures for whole matrices 50

3.9 A vector storage mechanism for ESP 51

4. Sparse Matrix Mechanisms in ESP **56**

4.1 The array form 56

4.2 The list form 57

4.2.1 Implementing linked lists in a single-level memory environment 59

4.2.2 Implementing linked lists in a hierarchical memory environment 60

4.2.3 Writing list vectors 64

4.2.4 Freeing list vector space 65

4.3 Indirection and vector registers 68

4.4 The Sideways List Unit 69

4.5 Summary 72

5. The Architecture of ESP **73**

5.1 General structure 73

5.2 The scalar processor and associated memories 77

5.3 The vector processor 78

5.3.1 Vector instructions 79

5.3.2 Instruction decode and control 83

5.3.3 The Vector Read circuit 84

5.3.4 The Index Match circuit 85

5.3.5 The Arithmetic Unit 86

5.3.6 The Vector Output circuit 87

5.3.7 The Vector Write circuit 87

5.3.8 The Garbage Collection circuit 88

5.3.9 The Sideways List Unit 88

5.4 The interface between the scalar and vector processors 89
5.4.1 Instruction transfer 89
5.4.2 Synchronisation between the scalar and vector processors . 90
5.4.3 Data transfer between the processors 91
5.4.4 Vector exceptions 91
5.5 The control processor 92
5.6 The instruction set 93

6. The Implementation of ESP **96**

6.1 Introduction 96
6.2 Basic design decisions 96
6.3 Physical partitioning of the machine 98
6.4 The host interface board 101
6.5 The control processor 103
6.5.1 The scalar bus 104
6.6 The host interface software 105
6.7 The scalar processor 105
6.8 The memory controller 107
6.8.1 The Vector Read circuitry 109
6.8.2 The Vector Write circuitry 111
6.8.3 The Garbage Collection circuitry 112
6.8.4 Vector memory request arbitration 113
6.8.5 The data transfer controllers 114
6.9 The vector memory 116
6.10 The vector Arithmetic Section 119

6.10.1 The Index Match circuit 120

6.10.2 The Arithmetic Unit 122

6.10.3 The Vector Output circuit 123

6.11 The Sideways List Unit 123

6.12 Testing ESP 124

7. The Performance of ESP **127**

7.1 Introduction 127

7.2 The LP simulation study 130

7.2.1 The model 130

7.2.2 The results 131

7.3 The first SIM++ simulation study 134

7.3.1 The model 134

7.3.2 The results 136

7.4 The second SIM++ simulation study 138

7.4.1 The model 138

7.4.2 Results for vector instruction loops 139

7.4.3 Results for Gaussian elimination 142

7.5 Summary 144

8. Summary and Conclusion **147**

8.1 Evaluation of the ESP architecture 148

8.1.1 The sparse vector handling mechanisms 149

8.1.2 The Sideways List Unit 149

8.1.3 The vector pipeline control strategy 151

8.1.4 The scalar processor/vector processor interface 151

8.1.5 The implementation 152

8.1.6 Scalability of the design 153

8.2 Evaluation of the design methodology 154

8.3 Further work on ESP 156

8.4 Conclusion 157

Bibliography **159**

A. The Vector Processor Instruction Set **164**

A.1 Vector Read circuit instructions 165

A.1.1 Two operand output modes 165

A.1.2 Single operand output modes 166

A.1.3 Null mode 166

A.1.4 Instruction termination 166

A.2 Index Match circuit instructions 167

A.2.1 I/V/V triple output modes 167

A.2.2 I/V pair output modes 168

A.2.3 Instruction termination 169

A.3 Arithmetic Unit instructions 169

A.3.1 Data movement operations 170

A.3.2 Arithmetic operations 171

A.3.3 Logical operations 174

A.3.4 Special operations 175

A.3.5 Instruction termination 175

A.4 Vector Output circuit instructions 176

A.4.1 Instruction termination 176

A.5 Vector Write circuit instructions 177

A.5.1 Sideways List Unit instructions 178

A.5.2 Instruction termination 178

B. Reprint of published paper **179**

Chapter 1

Introduction

1.1 Supercomputer architectures

Most of the largest and most time-consuming numerical computer applications involve data in the form of vectors and matrices. Computing with such data requires large numbers of arithmetic operations, each acting on single elements of the vectors or matrices involved. The standard way of coding such a computation in, for example FORTRAN (most scientific and engineering programs have, since the 1960's, been written one or another version of FORTRAN), is as a nest of DO loops indexing through FORTRAN ARRAYs of one or more dimensions, and this coding implies sequential execution of the elemental arithmetic operations. However, these arithmetic operations can to a very large extent be carried out in any order, without affecting the result, or they can be done simultaneously, and thus these algorithms exhibit a degree of potential parallelism. The finest-grained parallelism, between the elemental parts of a single vector operation, is *homogenous*, in the sense that each of the parallel operations is identical, although each operates on different data. Many numerical computations also exhibit potential parallelism at higher levels, between separate vector operations, and between even larger chunks of code. This coarser-grained parallelism is often heterogenous – it involves different operations, which can be carried out in parallel.

Conventional computers are often able to overlap individual instructions, and in some cases this process is assisted by executing instructions in a slightly different order from that defined by the program code itself. The maximum speed of most conventional machines is limited by the fact that they can issue at most

one instruction per clock cycle, although the latest *superscalar* architectures are able to average less than one cycle per instruction. However, all these standard *scalar* processors are unable to take advantage of any of the levels of parallelism described above. As there appears to be no limit to the size of numerical problem which scientists and engineers can find good reason to solve (even on the fastest machines available today, crucial numerical problems in fundamental physics take months to solve), much effort has been spent, over three decades, in developing improved computer architectures which can solve these problems faster. To do so requires extra hardware, and that hardware has been arranged in one of two general ways. *Vector processors* incorporate a *pipeline* of hardware elements for executing, in sequence, each step of a single vector elemental operation (address calculation, data fetch, arithmetic and data store). Several elemental operations may be in execution simultaneously, in different stages of the pipeline, and this design takes advantage of the finest-grained parallelism described above. *Parallel processors* contain a number of identical processing units. Depending on the control arrangement, a parallel processor may be able to exploit fine- or coarse-grained parallelism in an algorithm.

1.1.1 Vector processing

Early examples of vector processors were the **STAR-100** [24], later redeveloped into the **CYBER-205** [12], and the **CRAY-1** [43]. The vector pipeline in these machines is split into several tens of stages, through which a vector elemental operation flows, one stage per clock cycle. When the pipeline is fully busy, a different vector elemental operation is in execution in each pipeline stage, and one elemental operation is completed per clock cycle. With this number of pipeline stages, each stage can be kept simple, and the clock cycle is correspondingly fast. An entire vector arithmetic operation is performed as a result of the issue of a single *vector instruction*, which specifies the whereabouts of the vector operands, the elemental operation to be performed (add, scalar product, *etc*), and the number of elemental operations to be performed. The circuits required to perform a floating-point add and a floating-point multiply are different, and separate add and multiply units are

therefore provided (each of which is split over several pipeline stages). Vector processors can therefore include vector instructions which use both these arithmetic units, to implement vector operations requiring both multiplication and addition (*eg* scalar product), and so, on these instructions, the peak floating point performance of a single vector processor is two floating point operations (one add and one multiply) per clock cycle.

In practice, it is difficult to obtain anything approaching this peak performance from a vector processor, for two principal reasons. Firstly, although once a vector instruction has got started, it proceeds at the rate of one elemental operation completed (one or two floating-point operations performed) per clock cycle, the instructions take a considerable time to start up. The instruction must be decoded, the vector processor must be configured, and the first elemental operation must pass through the length of the pipeline, before the first result is produced. The start-up time for the **CRAY X-MP** [13] (the successor to the **CRAY-1**) is around 50 cycles, while for the **CYBER-205** it is closer to 100 [25]. As a result, the overall performance of a vector instruction is only close to the peak performance if the vector operands are long; if the operands have the same number of elements as the number of cycles start-up time, for example, the overall floating point performance of a vector instruction can be no more than half the peak.

The second reason that processor performance may be much less than the theoretical peak is that application programs do not consist entirely of operations on vectors; much of the code involves operations on single scalar values, and control operations. As a result, the vector processing hardware may be idle much of time, while the machine executes scalar instructions. (The fact that the speed-up available through the use of a vector processor is limited by the fraction of the code which must execute on the scalar processor is known as "Amdahl's law" [6].) This problem can be alleviated by careful algorithm design, including the use of optimised hand-written assembler code, but it is also important that the processor is able to carry out scalar operations fast enough to balance the vector processor speed. The biggest failing of the commercially unsuccessful **STAR-100** design, corrected in its successor, the **CYBER-205**, was the poor scalar performance.

The mechanisms used to synchronise the execution of scalar and vector instructions are also important in this respect.

Since the introduction of the first commercial vector machine, the **CRAY-1**, in 1976, vector processing has been highly successful, with vector machines in widespread use for numerically demanding applications. Other successful machines include the **CDC CYBER-205**, the **CRAY X-MP**, **CRAY Y-MP** and **CRAY-2** ranges, Japanese machines such as the **Fujitsu FACOM** series, the **Hitachi HITAC** and the **NEC SX1** and **SX2** (described in [25]), and the vector processor version of IBM's mainframe series, the **IBM 3090VF** [10]. More recently, however, there has also been much interest in the parallel processing alternative.

1.1.2 Parallel processing

Parallel numerical processing involves the use of more than one arithmetic processor simultaneously. However, parallel machines differ in their number of processors, in the way in which instructions are issued to the separate arithmetic units, and in the way that memory access is organised.

Array processors

One way of organising the use of parallel identical arithmetic units is to build them into a processor which fetches a single stream of instructions which operate on replicated data structures, such as vectors and matrices, but in which the arithmetic operation(s) for each element in the operand structures are performed in a different processor. Execution of the elemental operations proceeds in parallel. The number of processing units may be a hundred or more, and each accesses data from its own memory. Vectors/matrices are distributed through the memory banks in a suitable way before computation commences, and instructions cause all the processing units to operate simultaneously. A limited degree of variation in instruction execution between different processors is usually supported – a flag in each processor may disable instruction execution, on a processor by processor basis, or an address

register in each processor may allow different processors to carry out the same instruction on data stored at different addresses. These machines are commonly known as *array processors* or *SIMD* (Single Instruction Multiple Data) parallel processors. The first such machine to be built was the **ILLIAC IV** [18], an experimental design. The **Burroughs BSP** [8] was developed in the light of experience with the **ILLIAC IV**, but was not exploited commercially, while commercial array processors include the **ICL DAP** [40] and its successor, the **AMT mini-DAP** from Active Memory Technology Ltd., and the **Thinking Machines CM-1** [23] and **CM-2**. In the **DAP** and **CM** machines, the individual arithmetic units are much slower than those used in vector processors. The high level of parallelism, however, (the CM-1 can be configured with 65,536 processors) means that very high overall arithmetic rates can be achieved, but only on vectors/matrices with very large numbers of elements. Also, the distribution of data must be carefully arranged, and hence programming is more difficult than for the vector processors.

Multiprocessors

Other parallel processors fetch a separate instruction stream for each processor – the processors are executing separate program threads, but can communicate data, and synchronise with each other. These machines are commonly called *multiprocessors* or *MIMD* (Multiple Instruction Multiple Data) machines. In machines with a small number of processors, the processors may share a common memory; examples include the **IBM** range of mainframe multiprocessors, and the **Sequent Balance** series (described in [27]). Access to the shared memory soon becomes a bottleneck as the number of processors is increased, and so machines with more than a few processors use large caches or local memory for each processor, in addition to the shared memory. Alternatively, all the memory may be physically distributed at the processors, but with each processor's memory mapped into the other processors' address spaces, and accessible via a high-bandwidth interconnection, as in the **BBN Butterfly** [41]. Other multiprocessor machines use no shared memory at all – each processor operates with program and data held in a local memory, and a communication network allows processors to synchronise and

exchange data. Examples of these *distributed memory* MIMD machines include systems based on the **INMOS Transputer** [29] range of microprocessors, such as the **Meiko Computing Surface** series [27]. Distributed memory MIMD parallel machines may contain several hundred processors.

1.1.3 Using vector and parallel machines

The early popularity of vector processors was no doubt because they provided a performance boost of perhaps ten times (compared to conventional scalar processors of the same technology), at reasonable extra cost in hardware, and because existing programs could easily be adapted to run on them. However, the full performance of these machines is only available on problems which are sufficiently *vectorisable*, *ie* the vectors must be long enough, and the algorithm must be capable of expression as vector code without serious scalar bottlenecks. On the other hand, for *highly* parallel problems, such as matrix multiplication, the peak performance of a vector machine is limited by the limited degree to which the parallelism of the problem is exploited in the single vector pipeline.

Parallel processors with many arithmetic units are in theory able to reach much higher peak performance, but at the cost of more difficult programming. Single instruction stream array processors, even more than vector processors, are only utilised efficiently on problems with a sufficient degree of fine-grained homogenous parallelism, and the increased speed then provided by the multiple processing units makes scalar bottlenecks more likely on array processors. These machines also present the programmer with the additional problem of data distribution.

MIMD multiprocessors can be effective on any problem which can be decomposed into parallel parts, even if the decomposition is heterogenous (*ie* different processors are carrying out different subtasks) – this class of problems includes many which cannot make efficient use of vector or array architectures. However, the programmer has the often difficult task of finding a problem decomposition which will fit the machine architecture in such a way that each processor has a sim-

ilar amount of work to do, and such that synchronisation and data communication between processors does not delay processing inordinately.

The vector processor and parallel processor approaches to performance improvement are not exclusive alternatives. Some commercial vector machines, such as the **CYBER-205** and the **NEC SX-2**, can incorporate up to four vector pipelines, and divide the elemental parts of a vector operation between the pipelines, thus combining the vector and array approaches. Peak performance is four times greater, but vectors must be four times longer to obtain the same percentage of that peak.

There is potentially more to be gained from combining the vector and MIMD multiprocessor approaches. The processors in an MIMD machine are usually relatively loosely coupled, in the sense that synchronisation and data communication between processors takes many cycles – this is particularly true of the larger, distributed memory systems. Because of this, MIMD machines are more suited to exploiting the coarser-grained parallelism in an algorithm. An MIMD machine consisting of a number of vector processors can simultaneously exploit parallelism at different levels of granularity in many large-scale numerical problems, and combines the performance advantages of each class of machine. Many vector processors are available in small parallel MIMD configurations, with a shared memory architecture supporting up to four processors. Examples include the **CRAY X-MP**, **Y-MP** and **CRAY-2** series of machines, and the **IBM 3090VF** (in which the maximum number of processors is six).

1.2 Sparse matrix problems

Sparse matrices are those in which many of the matrix elements have the value zero. They arise naturally in the formulation of problems in application areas such as design, simulation and optimisation, and in finite approximation methods for solving differential equations. In all cases, the matrix is sparse because most or all of the linear equations in the formulation involve only a small subset of

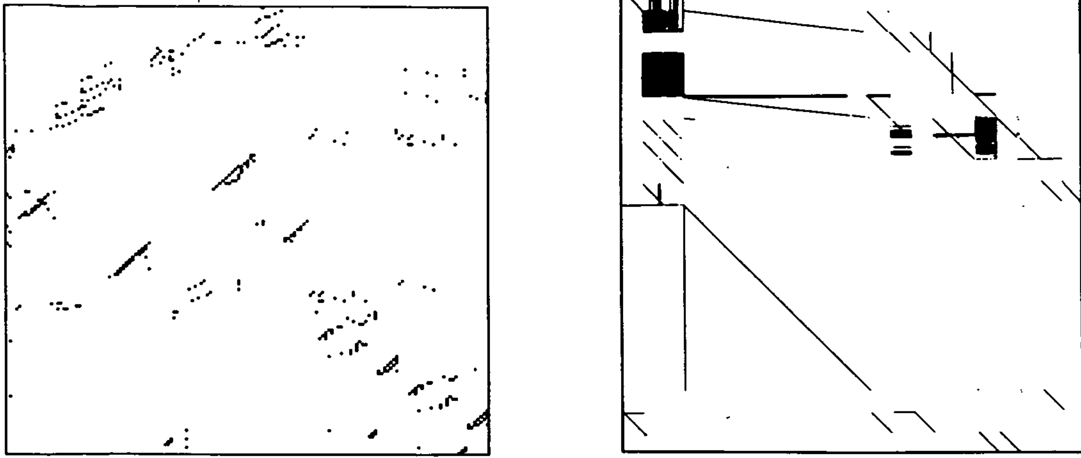

Figure 1–1: Irregular sparse matrices arising from a chemical plant model (left) and from an economic model (right), from [16]

the variables. This reflects the nature of the physical problem modelled – in a structural model, each piece of the structure usually connects only to a few other pieces; in a large optimisation problem, most of the constraints involve only a few of the variables; in a finite element formulation of a differential equation problem, each element is affected only by adjacent elements. In the last example, the regular pattern of elements and the fact that each 'connects' only to adjacent elements mean that the pattern of non-zeroes in the resulting matrix is regular (often a band along the diagonal), while for the other examples, the pattern reflects the pattern of 'connections' between the elements of the model, and can be very irregular (figure 1–1).

Many useful engineering or optimisation problems involve matrices which are very large (tens of thousands of rows) and very sparse (the constraint matrix for a large Linear Programming problem might typically have 5 to 7 non-zero elements per column). Clearly such matrices must be treated specially – a matrix of order 20,000 is much too large to handle on a computer if it were stored as an ordinary, 'dense' matrix. These large sparse problems can only be solved because the matrices can be stored in a compressed form with the zeroes removed, and because arithmetic need only be performed on the non-zero elements. The amount

of arithmetic required to solve a sparse problem depends on the number of non-zeroes in the matrix, but the arithmetic operations themselves tend to create new non-zeroes as the computation proceeds – a process known as *fill-in* of the matrix. The amount of fill-in which occurs depends very greatly on the order in which computation is carried out, and thus sparse problems require special algorithmic techniques to ensure a computation order which keeps fill-in to the absolute minimum.

Sparse matrix computation researchers over the last three decades have developed a variety of ways of minimizing fill-in, and have experimented with a number of data structures for sparse matrices which support rapid identification and execution of the required arithmetic operations. Much of the sparse matrix software developed over this period has been for conventional *scalar* processor architectures; little use has been made of vector processors, which are designed specifically to operate on dense vector data structures. However, interest in vector and parallel solution of sparse matrix problems has grown over the past few years, partly as a result of the introduction of vector processor instructions which can, to a limited extent, operate on sparse vectors stored in compressed form, and partly as a result of the increasing availability of MIMD parallel machines able to take advantage of coarser-grained parallelism in sparse matrix problems.

1.2.1 Use of vector processors for sparse problems

Vector processor instructions are designed to operate on dense vectors stored as *arrays* – the elements of the vector are all present in memory, and occupy consecutive memory locations. In some sparse problems, where the non-zero structure of the original matrix is regular (as it is in matrices arising from finite element methods, for example), or where the matrix can be permuted to bunch together the non-zeroes, dense vector data structures can be used for that small part of each vector which contains the non-zeroes, and implementations can take then advantage of vector processor facilities. Many problems, however, involve a highly

irregular pattern of non-zeroes, and the vectors must necessarily be stored in some compressed form on which standard vector instructions will not operate.

In recognition of this problem, some vector processor developers have considered ways of providing vector instructions which operate on sparse vector operands stored in compressed form. This work was pioneered by the designers of the **CYBER-205** [12], who introduced in that machine two sets of novel instructions, operating on sparse vectors stored in two different forms: the "order vector" form and the "index vector" form. These mechanisms are discussed in chapter 3, and for reasons explained there, the "order vector" mechanisms were not a success. The "index vector" mechanisms actually comprised two operations only, called "scatter" and "gather", which carry out subparts of a common implementation of vector arithmetic on compressed sparse vectors. Using the scatter and gather vector instructions, an addition operation between two sparse vectors stored in compressed form can be implemented by a sequence of eight vector instructions, and this can be considerably faster than a scalar implementation of the same vector arithmetic. The scatter and gather operations have been used in successfully in sparse matrix software [19,16], and have become a standard feature on vector processors (all Cray models from the **X-MP/48** onwards have incorporated them, as do the Japanese machines and the **IBM 3090VF**). Because, even with scatter and gather instructions, a sparse vector arithmetic operation requires a sequence of several vector instructions, most of which involve data rearrangement, not arithmetic, the overall arithmetic rate obtainable from a vector processor on sparse vector arithmetic operations remains far below that obtainable from the same processor on dense vectors.

1.2.2 Use of parallel processors for sparse problems

Of the two classes of parallel machine with substantial parallelism, the SIMD array and MIMD multiprocessor, only the array processor has been available for applications research for any length of time. Attempts have been made to mount sparse matrix problems on such machines [39,37], but these have run into similar

difficulties as those arising with conventional vector architectures – the machines work well on dense vectors stored as arrays, but are hard to adapt to problems involving arithmetic on compressed sparse vector data structures.

MIMD parallel processors (with a reasonable degree of parallelism) have only become available for widespread research on program design in the past five years, and parallel implementations of sparse matrix problems are thus at an earlier stage of experimentation [7,14,34,45,51]. Nevertheless, it seems likely that the parallelism of such machines can be exploited in many sparse problems, although the best parallel decomposition of program and data is likely to be machine architecture dependent. One MIMD parallel machine, the **(SM)^2-II** [3,4,5] has been developed specifically for sparse matrix problems, and a small prototype is under evaluation.

As with dense matrix code, the parallelism exploitable on MIMD architectures is coarser grained than that exploitable by vector processors, and research into the implementation of sparse matrix software on both these classes of machine remains potentially fruitful.

1.3 The Edinburgh Sparse Processor project

The **Edinburgh Sparse Processor (ESP)** project developed out of research on the **MU6V** machine carried out at Manchester University [47,33,26]. In its final form, **MU6V** was an MIMD parallel machine, designed for vector and matrix applications, and each processor in the parallel machine was itself intended to be a vector processor, with an instruction set which included vector instructions to operate on sparse vectors, as well as the usual dense vector instructions. The focus of research, however, was on the MIMD parallelism, and in the prototype each vector processor was emulated using scalar code running on a **Motorola 68010** microprocessor. A mechanism was proposed for implementation of sparse vector storage and arithmetic, based on the **CYBER-205** "index vector" mechanism, but with more flexibility, but the sparse vector operations were never implemented in the emulated vector processor, and the sparse mechanism was not developed.

However, the **MU6V** proposal had suggested that it might be possible to build a vector processor capable of executing the full set of vector arithmetic instructions directly on sparse vectors stored in compressed form. If so, the result would be a performance improvement on sparse vector arithmetic, compared to scalar processor implementations, at least as great as that provided by conventional vector processors for dense vector arithmetic.

The aims of the author's research were to investigate the feasibility of efficiently supporting sparse vector arithmetic in a vector processor, and, if feasible, to develop an architecture for a machine incorporating sparse vector instructions: the Edinburgh Sparse Processor. ESP was intended to be a single vector processor machine, although it was expected that, if the design was successful, an MIMD parallel version, built by replicating the ESP architecture, would give even higher performance on large sparse problems. The development of a prototype of the machine has been supported by High Level Hardware Ltd., of Oxford, and the UK Science and Engineering Research Council.

Specific objectives of the research described in this thesis were:

- to identify the vector and matrix operations which must be implemented efficiently to support chosen target applications;

- to identify mechanisms for storage of sparse vectors and matrices of a wide range of sizes and densities, and for computation with those vectors and matrices, including mechanisms for all standard vector operations, plus any other operations identified as necessary for the target applications; to examine the feasibility of hardware support for those mechanisms, and to evaluate their resulting efficiency;

- to develop a vector processor architecture capable of high performance on the target applications, incorporating suitable sparse vector mechanisms and supporting a full vector instruction set, based on that developed for **MU6V**, plus any additional instructions identified as required for the target applica-

tions, and with each instruction able to operate efficiently, where appropriate, with dense vector or sparse vector operands, or a mixture of the two;

- to evaluate the technical feasibility of the new architecture, by designing an implementation using off-the-shelf components, and to examine the effectiveness of the architecture by analysis, by simulation, and by building and testing a prototype.

Chapter 2

Sparse Matrix Algorithms

2.1 Introduction

The principal chosen target application for ESP is Linear Programming (LP). This is an application of great commercial interest, for example in the manufacturing, transport and communications industries, and is also of importance as a sub-problem of non-linear optimisation problems. Many LP problems are large and sparse, and the availability, from a local research group, of a variety of real problems of different characteristics assists in the evaluation of the architecture and of the machine. However, in developing the architecture of ESP, the simpler problem of the direct solution of sparse systems of linear equations by Gaussian elimination was also targetted, as this problem is both in itself an important application area, and is a sub-problem of the most commonly used LP algorithm.

This chapter presents a general description of the methods currently used to solve these sparse matrix problems, concentrating on the operations which must be performed at matrix and vector level.

2.2 Direct solution of linear equations

There are two classes of method for solving

$$\mathbf{Ax} = \mathbf{b}$$

– direct solution methods, and iterative techniques (the latter are considered in section 2.3 below). Direct methods for the solution of linear equation systems are based on the *Gaussian elimination* algorithm. The basic principle is to reduce the matrix $\mathbf{A}$ (or some permutation of it) to upper triangular form by subtracting multiples of each row from the rows below it, each time zeroing out a column in the lower triangle. The same subtractions may be performed on the right hand side $\mathbf{b}$, as the solution proceeds, leaving a system of the form

$$\mathbf{Ux} = \mathbf{b}'$$

which is solved by back substitution of the elements of $\mathbf{x}$. Alternatively, the sequence of subtraction multipliers may be stored, yielding an explicit factorisation $\mathbf{LU} = \mathbf{A}$, which may be used to solve the equations for multiple right hand sides, by forward-substitution to solve $\mathbf{Ly} = \mathbf{b}$, then back-substitution to solve $\mathbf{Ux} = \mathbf{y}$. (The related *Gauss-Jordan* method diagonalises the original matrix in one calculation, by applying the row subtractions to rows above the diagonal as well as below it. It is not widely used in practice, as it involves more arithmetic operations than the Gaussian method.)

An intermediate stage in Gaussian elimination is illustrated in fig. 2–1. The principal operation required is vector subtraction:

$$\mathbf{A'}_{i\bullet} \leftarrow \mathbf{A'}_{i\bullet} - s_i * \mathbf{A'}_{p\bullet}$$

where $\mathbf{A}'$ is the current, partially triangularised, version of $\mathbf{A}$. Using this operation repeatedly, the lower part (*ie* the part below the diagonal) of a column of $\mathbf{A}'$ is zeroed by subtracting suitable multiples of row p from all the rows i beneath it. s_i is a scalar chosen for each row i so that the subtraction zeroes the pth element

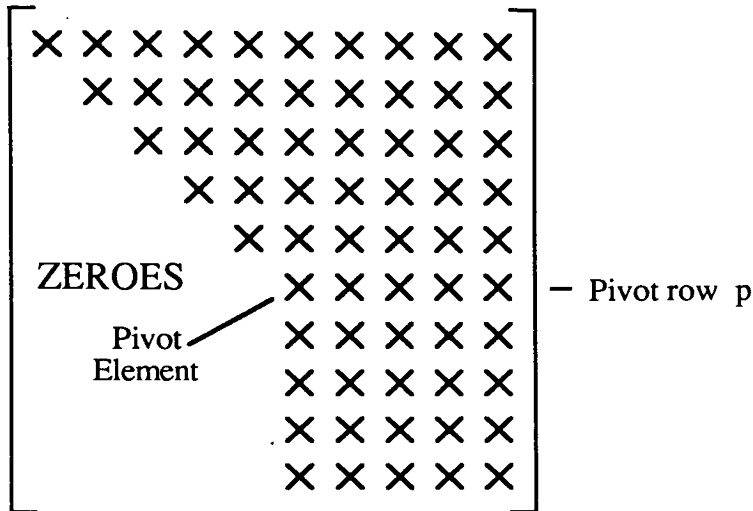

Figure 2–1: Gaussian elimination

of the row. $\mathbf{A}'_{p\bullet}$ is known as the *pivot row*, and $\mathbf{A}'_{\bullet p}$ is the column of which the lower part is being zeroed, and is called the *pivot column.* Element a'_{pp} is called the pivot element, and s_i is given by:

$$s_i = a'_{ip}/a'_{pp}$$

2.2.1 Choice of pivot element

In the basic Gaussian elimination algorithm described above, the pivot elements are the diagonal elements of the matrix, and are used in order from top to bottom. This is often unsatisfactory, however, because when a diagonal element is used as the pivot its value may be relatively very small compared to the elements below it in the pivot column, and this will necessitate subtraction of very large multiples of the pivot row. Because of the limited precision of floating point number representations, the size of such subtracted row element multiples may swamp the original matrix entries, and the resulting error in the calculation will render it useless. It is therefore normal to select the element to be used as pivot at each stage of the elimination, either by exchanging rows to replace the diagonal pivot element with the largest element in the pivot column (this operation is known as *partial pivoting*), or by exchanging rows and columns to make the pivot

the largest element in the submatrix remaining to be triangularised (*full pivoting*). The overall effect is the same as applying row and column permutations to **A** *before* factorisation, and then pivoting down the diagonal, but the pivot choices must be made *during* factorisation, as the values of the relevant elements change during the calculation. In practice, partial pivoting almost always provides sufficient numerical stability, and full pivoting is not normally used.

In the special cases where the matrix **A** is symmetric and positive definite, or diagonally dominant, pivoting can proceed down the diagonal, without permutation, as the factorisation will always be numerically stable.

2.2.2 Gaussian elimination on sparse matrices

Sparse systems of linear equations arise for example from engineering models, where structures to be modelled often consist of (or may be approximated by) many parts sparsely interconnected, from optimisation problems, and from difference methods for solving partial differential equations. All of these problem areas can generate very large matrices, of order 10^4 or larger, but the problems differ in the typical patterns of distribution of the non-zeroes through the matrix.

2.2.3 Sparse matrices with regular structure

Sparse matrices arising from partial differential equation solution by finite differences, and from many finite element methods, generally have a regular, banded structure. The band is relatively dense, and there are no non-zeroes outside it. If Gaussian Elimination proceeds using pivots on the diagonal, no fill-in will occur outside the band. Partial pivoting will result in, at most, a doubling of the width of the upper half of the band, without affecting the width of the lower half.

Obviously, only the parts of the rows within the band need be processed, and it is efficient to store and process the band alone, as a set of *dense* vectors, using the vector arithmetic mechanisms of an ordinary vector processor.

2.2.4 Sparse matrices with irregular structure

Matrices arising from engineering or optimisation problems often have an irregular distribution of non-zeroes, reflecting the inherent structure of the original problem (see fig. 1–1). Algorithms exist to permute matrices to band structured matrices with minimised band width - the resulting matrices may then be solved using dense vector operations on the band. However, it is often not possible to permute the non-zeroes into a dense enough band to make this method efficient.

In such a case, it is better to work on the matrix in its sparse form. To solve large problems of this kind, the non-zeroes, and information about their positions, must be stored in compressed form, but in such a way that the elimination and pivot choice steps of the Gaussian elimination algorithm can proceed efficiently.

The elimination step (*ie* the subtraction of multiples of a single pivot row from other rows in the matrix) here involves the subtraction of one sparse vector from another ($\mathbf{x} \leftarrow \mathbf{x} - \mathbf{y}$), and will generally involve *fill-in* of the vector $\mathbf{x}$ – if the positions of the non-zeroes in $\mathbf{y}$ do not coincide with the non-zeroes in $\mathbf{x}$, the updated vector $\mathbf{x}$ will contain more non-zero elements after the subtraction. The storage scheme for the vectors must be able to deal with this.

If the matrix is sparse, it is only necessary to subtract the pivot row from those rows in the matrix with a non-zero in the pivot column – the storage scheme must therefore allow fast identification of the position of the non-zeroes in each column of the updated matrix, as the elimination proceeds, or the time spent searching for these will dominate the solution time. In order to calculate the scalar multiple of the pivot row to be subtracted, the actual values of the non-zeroes in the pivot column must also be rapidly accessible.

2.2.5 Pivot choice in sparse Gaussian elimination

The overall computation time will depend on the total number of non-zeroes processed, and this number is increased by fill-in of the as yet unfactorised part of the matrix, caused by previous pivot row subtractions. The amount of fill-in depends

on the chosen sequence of pivots, because the more non-zeroes there are in the chosen pivot row, the more fill-in is likely to occur in each row from which the pivot row is subtracted, while the more non-zeroes there are in the pivot column, the more such rows there will be. A second requirement, the need to minimize fill-in, must therefore guide the choice of pivots in the sparse case, in addition to the requirement to choose pivots which do not upset the numerical stability of the calculation.

The most common criterion for pivot choice to minimise fill-in is known as the *Markowitz criterion* [36]. It consists of choosing as pivot the element in the unfactorised sub-matrix with the minimum product of number of non-zeroes in its row, and number of non-zeroes in its column. To find such an element efficiently requires rapid assessment, at each stage of the factorisation, of the number of non-zeroes currently in each row and column of the matrix.

The usual way of combining the requirements of numerical stability and minimal fill-in is known as *threshold pivoting* [11]. The element to be used as the pivot is chosen on fill-in grounds, using the Markowitz criterion, and its magnitude is then checked against all other elements in its row (or alternatively, all other elements on or below the diagonal in its column). If the selected pivot is too much smaller than the largest element in its row, it is rejected, and another candidate chosen on fill-in grounds.

2.2.6 Pivot choice on sparsity grounds only

In special cases, for example where the matrix is symmetric and positive definite, *any* permutation may be applied to it before factorisation, without risking numerical instability. It is therefore normal practice to determine a permutation which will minimise fill-in, before factorisation starts. (This can be done before factorisation because the amount of fill-in depends on the choice of pivots and the non-zero structure of the original matrix, but, ignoring the rare possibility of cancellation of a matrix entry to zero during a subtraction step, not on the *values* of the non-zeroes.) Finding the optimal sequence of pivots is provably NP-complete, but

Figure 2–2: Typical pattern of non-zeroes in the factors $\mathbf{L} \backslash \mathbf{U}$ of a sparse matrix. In this case, the original matrix was symmetric, and $\mathbf{U} = \mathbf{L}^{\mathbf{T}}$ (from [16]).

heuristic methods based on a graph representation of the non-zero structure are fast and work well [11]. Factorisation can then proceed without further checking of pivots on sparsity or stability grounds.

2.2.7 Size and sparsity of typical problems

In [17], Duff *et al.* list a set of typical problem matrices, which they used to compare different implementations of Gaussian elimination. These vary in order from 156 to 5300, while the number of non-zeroes ranges from 371 for the order 156 matrix, to 21842, for the matrix of order 5300, with most matrices averaging fewer than ten non-zeroes per row. As Gaussian elimination proceeds, fill-in accumulates, so that the resulting factors **L** and **U** become increasingly dense towards the bottom right-hand corner (see Fig. 2–2). The amount and pattern of fill-in obviously depend on the structure of the original matrix, but, typically, the unfactorised submatrix might be 25% full after 70% of the pivots have been used, and 50% full after 80% of the elimination steps are done [15].

2.3 Iterative solution of linear equations

If $\mathbf{x}_0$ is an inaccurate solution to $\mathbf{Ax} = \mathbf{b}$, then the accurate solution is

$$\mathbf{x} = \mathbf{x}_0 + \mathbf{A}^{-1}(\mathbf{b} - \mathbf{A}\mathbf{x}_0)$$

If $\tilde{\mathbf{A}}^{-1}$ is a sufficiently close approximation to $\mathbf{A}^{-1}$, then the iterative sequence

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \tilde{\mathbf{A}}^{-1}(\mathbf{b} - \mathbf{A}\mathbf{x}_n) \tag{2.1}$$

will converge to an accurate solution.

The trick is to choose $\tilde{\mathbf{A}}^{-1}$ to minimize the computation involved in evaluating (2.1), while achieving fast convergence. If $\mathbf{A}$ is partitioned into its diagonal $\mathbf{D}$, the part below the diagonal $\mathbf{A}_L$, and the part above the diagonal $\mathbf{A}_U$, so that $\mathbf{A} = \mathbf{A}_L + \mathbf{D} + \mathbf{A}_U$, then the *Jacobi* method chooses $\tilde{\mathbf{A}}$ to be $\mathbf{D}$, so that (2.1) may be re-written

$$\mathbf{D}\mathbf{x}_{n+1} = -(\mathbf{A}_L + \mathbf{A}_U)\mathbf{x}_n + \mathbf{b}$$

The right-hand side uses (most of) the original matrix, and takes full advantage of its sparsity; $\mathbf{Dx} = \mathbf{y}$ is trivial to solve.

The *Gauss-Seidel* method uses $\tilde{\mathbf{A}} = \mathbf{A}_L + \mathbf{D}$, so that

$$(\mathbf{A}_L + \mathbf{D})\mathbf{x}_{n+1} = -\mathbf{A}_U\mathbf{x}_n + \mathbf{b}$$

Again, full advantage is taken of the sparsity of $\mathbf{A}$; $(\mathbf{A}_L + \mathbf{D})\mathbf{x} = \mathbf{y}$ is solved by forward-substitution.

Both the above methods are certainly convergent if $\mathbf{A}$ is diagonally dominant, while Gauss-Seidel also converges if $\mathbf{A}$ is symmetric and positive-definite. The convergence of the Gauss-Seidel method may be improved by *relaxation* - in this

variant, $\tilde{\mathbf{A}}$ is chosen to be $\omega^{-1}(\omega\mathbf{A}_L + \mathbf{D})$ with ω a factor chosen by experience and dependent on the problem type.

An alternative choice for $\tilde{\mathbf{A}}$ is to use an approximation to the **LU** factorisation of **A**, $\tilde{\mathbf{A}} = \tilde{\mathbf{L}}\tilde{\mathbf{U}}$ [16]. The factors $\tilde{\mathbf{L}}$ and $\tilde{\mathbf{U}}$ are generated by setting to zero any matrix entries smaller than a chosen threshold encountered during the Gaussian elimination of **A**, and are thus much sparser than accurate factors. Evaluating the product term on the right hand side of (2.1) is done by forward- and back-substitution using the sparse approximate factors.

Other, more sophisticated iterative methods, such as the *conjugate gradient* method [9] may be used; these methods also take good advantage of the sparsity of **A**, while converging faster than the simpler methods above. However, one often cannot be sure in advance that iteration will converge satisfactorily, and in practice this has restricted the use of iterative methods to special cases where convergence is known to be good and where a good initial approximation is known. In any case, these iterative algorithms are computationally relatively straightforward, and the sparse vector operations required to implement them are included in the set required to be supported by ESP for other applications which have been considered in detail. For these reasons, iterative methods are not considered further.

2.4 Linear Programming

2.4.1 The simplex method

The general Linear Programming (LP) problem consists of finding a set of values for a number of variables (represented here as an n-element vector $\mathbf{x}$) which maximizes a given linear function of those variables (known as the *objective* function):

$$f = \mathbf{c}^T\mathbf{x}$$

subject to linear constraints:

$$l_i \leq \mathbf{A}_{i\bullet}\mathbf{x} \leq u_i \qquad 1 \leq i \leq m \tag{2.2}$$

$$l_{m+j} \leq x_j \leq u_{m+j} \qquad 1 \leq j \leq n \tag{2.3}$$

Each constraint inequality defines a pair of hyperplanes in the n-dimensional $\mathbf{x}$ space, between which any solution must lie. The inequalities together define a convex region *(simplex)* in which the solution must lie (if the problem is *feasible*, *ie* there are values of $\mathbf{x}$ which satisfy all the constraints, such a region will certainly exist, and in a well-behaved problem it will be bounded). The maximum value of the objective function will be attained at one (or more than one) corner of the simplex. The *simplex method* consists of first finding a corner of the simplex, then moving around the corners step by step, improving the value of the objective function at each step, until no further improvement is possible.

The simplex method usually involves converting the problem set out in (2.2) and (2.3) into a form in which all the inequality constraints refer to a single variable only. This is done by introducing a unique extra variable into each of the inequalities in (2.2), so that the inequality,

$$l_i \leq a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n \leq u_i$$

becomes:

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n + x_{n+i} = 0$$

$$-u_i \leq x_{n+i} \leq -l_i$$

The newly introduced variables are known as *logical*, or alternatively *slack* or *surplus* variables.

Applying this transformation to each inequality converts the problem set out in (2.2) and (2.3) into the standard form for the simplex method:

Maximize $f = \mathbf{c}^T\mathbf{x}$, subject to:

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{2.4}$$

$$l_i \leq x_i \leq u_i \qquad 1 \leq i \leq n+m$$

With the introduction of the logical variables, $\mathbf{x}$ and $\mathbf{c}$ are now $n+m$ long (the elements of $\mathbf{c}$ corresponding to the logical variables are zero). $\mathbf{A}$ is m by $n+m$, and includes the m by m identity matrix.

There are m independent equations in (2.4), so if n components of $\mathbf{x}$ are fixed, (2.4) uniquely determines the remaining m components. If $\mathbf{x}$ is partitioned into n components which are fixed (the 'independent' variables), written $\mathbf{x}_I$, and the remainder (the 'dependent' variables), $\mathbf{x}_D$, equation (2.4) and the objective function may be rewritten:

$$\mathbf{A}_D\mathbf{x}_D + \mathbf{A}_I\mathbf{x}_I = \mathbf{b}$$

$$f = \mathbf{c}_D^T\mathbf{x}_D + \mathbf{c}_I^T\mathbf{x}_I$$

$\mathbf{x}_D$ and f may be determined from $\mathbf{x}_I$:

$$\mathbf{x}_D = \mathbf{A}_D^{-1}\mathbf{b} - \mathbf{A}_D^{-1}\mathbf{A}_I\mathbf{x}_I \tag{2.5}$$

$$f = \mathbf{c}_D^T\mathbf{A}_D^{-1}\mathbf{b} - (\mathbf{c}_I^T - \mathbf{c}_D^T\mathbf{A}_D^{-1}\mathbf{A}_I)\mathbf{x}_I \tag{2.6}$$

If an initial feasible solution (*ie* a solution which satisfies all the constraints) $\mathbf{x}$, and partition $\mathbf{x}_D \mid \mathbf{x}_I$, are chosen such that each of the n independent variables in $\mathbf{x}_I$ is at one of its bounds, this will correspond to a corner of the feasible simplex, and the value of $\mathbf{x}_D$ will indicate the distances from the remaining bounds. Moving to an adjacent corner of the simplex corresponds to setting one of the $\mathbf{x}_D$ components to a bound, while allowing one of the $\mathbf{x}_I$ components to come off its

bound, and thus involves altering the $\mathbf{x}_D \mid \mathbf{x}_I$ partition by making one dependent variable independent, and *vice versa*.

Each iteration of the simplex method involves choosing an independent variable to move off a bound, and a dependent variable which will reach a bound. The equations (2.5) and (2.6) must then be updated to reflect the new partition. The value of $\mathbf{A}_D^{-1}\mathbf{b}$ can easily be updated at each step, while the *standard simplex* method also updates $\mathbf{A}_D^{-1}\mathbf{A}_I$ at each step. However, for problems of significant size ($n > 100$), the *revised simplex* method is substantially faster. This involves storing $\mathbf{A}_D^{-1}$ in a form which is easy to update as the $\mathbf{x}_D \mid \mathbf{x}_I$ partition changes, and from which $\mathbf{c}_D^T\mathbf{A}_D^{-1}$ and (the required part of) $\mathbf{A}_D^{-1}\mathbf{A}_I$ is calculated at each step.

2.4.2 Computational steps in the revised simplex method

In the revised simplex method (also called the *product form of inverse (PFI) simplex method*), the inverse matrix $\mathbf{A}_D^{-1}$ is stored as a list of matrices whose product is the inverse matrix. Each of these matrices has the form of the unit matrix, with one special column:

$$\begin{pmatrix} 1 & 0 & \cdots & \eta_1 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & \eta_2 & \cdots & 0 & 0 \\ & & & \cdot & & & \\ & & & \cdot & & & \\ & & & \cdot & & & \\ 0 & 0 & \cdots & \eta_{m-1} & \cdots & 1 & 0 \\ 0 & 0 & \cdots & \eta_m & \cdots & 0 & 1 \end{pmatrix}$$

These matrices are usually known as the *η-matrices*, and the interesting column of the matrix as the *η-vector*.

A single iteration of the simplex method replaces one column of $\mathbf{A}_D$ with a column from $\mathbf{A}_I$, and applying the old inverse matrix to this new $\mathbf{A}_D$ would result in a unit matrix with one filled column. The inverse matrix can therefore be

updated by adding one extra η-matrix to the front of the list, chosen so as to reduce the extra column to a single entry, value 1, without affecting the other columns. The η-matrix list thus grows by one entry per iteration of the algorithm.

Choosing the variables to update

At each iteration, to choose the independent variable to be moved off its bound, the effect on the objective function of changes in the value of each independent variable must be known. This is given by the term in brackets in equation (2.6) above. At each iteration $\mathbf{A}_D^{-1}$ changes, and $\mathbf{c}_D^T\mathbf{A}_D^{-1}$ must be calculated anew. Its value is given by:

$$\mathbf{c}_D^T\mathbf{E}_j\mathbf{E}_{j-1}\cdots\mathbf{E}_1$$

where the $\mathbf{E}$ are the η-matrices described above.

If k is the column position of the vector η within the matrix $\mathbf{E}$, then $\mathbf{c}^T\mathbf{E}$ is simply the vector

$$(c_1, c_2, \ldots c_{k-1}, (\mathbf{c}.\eta), c_{k+1}, \ldots c_m).$$

Thus $\mathbf{c}_D^T\mathbf{A}_D^{-1}$ may be calculated via a sequence of scalar products, the result of each of which updates a single element of $\mathbf{c}$. The result is called the *price* vector π, and this calculation, which must be redone at every iteration, is called the *BTRAN* operation.

Assessment of the effect on the objective function of a change to a single independent variable now requires a scalar product of the price vector with the corresponding column of $\mathbf{A}_I$ (refer to equation (2.6) above). The independent variable chosen to move from its bound is the one for which the rate of increase in the objective function is largest, although when solving a large problem, it is usually sufficient to make the choice from a subset of the independent variables at each iteration (some versions of the algorithm, *eg DEVEX* [21], use slightly

different criteria for choosing the variable). The calculation of the required scalar products is known as the *pricing* operation.

To calculate how far the chosen independent variable can move off its bound, it is necessary to ascertain which of the dependent variables reaches a bound first, as the value of the independent variable changes.

The effect on $\mathbf{x}_{\hat{D}}$ of changes in a single independent variable is given by a single column of $\mathbf{A}_D^{-1}\mathbf{A}_I$ (refer to equation (2.5) above). Once the independent variable to be changed has been chosen at the pricing step, this column can be calculated from the corresponding column in the original matrix A, by:

$$\mathbf{E}_j\mathbf{E}_{j-1}\cdots\mathbf{E}_1\mathbf{A}_{\bullet i}$$

As matrix multiplication is associative, this can be evaluated from right to left, and the result of pre-multiplying a vector **a** by the specially structured matrix **E** is given by:

$$\mathbf{a} + a_k\eta$$

where k is the position of the interesting column η within the matrix **E**. The calculation of the required column of $\mathbf{A}_D^{-1}\mathbf{A}_I$ can therefore be implemented as a sequence of scaled vector addition steps. This calculation is known is the *FTRAN* operation.

Once the *FTRAN* operation is complete, the resulting vector gives the rate of change of each dependent variable with change in the chosen independent variable, and, as the current values of the dependent variables are known, it is easy to ascertain which reaches a bound first. The step in the independent variable is then fixed, and all the dependent variables may be updated. The dependent variable which is now on a bound becomes 'independent', while the independent variable which was moved off a bound becomes 'dependent'. The result of the *FTRAN* operation is also used to generate the new η-matrix to update the representation of $\mathbf{A}_D^{-1}$, because it corresponds to the result of applying the existing $\mathbf{A}_D^{-1}$ to the

new column introduced to $\mathbf{A}_D$ when the $\mathbf{x}_D \mid \mathbf{x}_I$ partition is updated. The extra η-matrix, which is added to the front of the list (*ie* becomes $\mathbf{E}_{j+1}$ in the notation used above), effects the subtraction steps required to clear out this column to a single 1.

The Re-invert Step

The product form of the inverse $\mathbf{A}_D^{-1}$ described above represents of a list of Gauss-Jordan elimination steps, each with its pivot chosen as described above. After a large number of iterations of the revised simplex method, this representation of the inverse matrix becomes prone to excessive fill-in, because the order of pivot choice ignores sparsity. Many implementations of the simplex method therefore use a slightly modified representation of the inverse matrix $\mathbf{A}_D^{-1}$, in which the inverse is still represented as a product of elementary matrices of the form of the η-matrices described above (thus the *BTRAN* and *FTRAN* steps remain as described above) but where those elementary matrices are initially generated by directly factorising the matrix $\mathbf{A}_D$, choosing pivots by a method which minimises fill-in and ensures stability, such as the threshold method described in section 2.2.5 above. This product form of the inverse is updated at each iteration (the exact method used to update the factors differs between implementations), and as the updates proceed, the number of non-zeroes in the η columns grows. It is therefore worthwhile to regenerate the inverse from time to time, by re-inverting the current matrix $\mathbf{A}_D$. This *re-invert* step comprises a Gaussian elimination operation on the very sparse $\mathbf{A}_D$, and is typically carried out every 100 or so iterations.

2.4.3 Size and sparsity of typical LP problems

The matrix $\mathbf{A}$ in a typical commercial problem might have 500 rows and 1500 columns [37] - the three to one shape is typical for large problems. Problems with more than 1000 constraints are very common, while some have 10000 or more. However, in a typical large problem, there are only a small number (perhaps 6

to 10) non-zeroes in each column, thus a model with a few thousand constraints would only be around 0.1% full.

At the very start, therefore, the η vectors are correspondingly sparse, but fill-in caused by the application of the product-form inverse results in the columns of $\mathbf{A}_D^{-1}\mathbf{A}_I$ typically being around 20% full for most of the computation. This is therefore the density of most η vectors added after a re-invert. The re-invert step will typically reduce the η density to around 1%.

During *BTRAN*, the original vector $\mathbf{c}_D^T$, which is usually very sparse, fills to reach a density of around 20% or more during the *BTRAN* operation – the final density of the price vector is problem dependent. During *FTRAN*, the column of $\mathbf{A}$ being updated fills rapidly to about 20% full [37].

Although in theory the number of iterations required to solve an LP problem can be exponential in the size of the problem, in practice the number of iterations required is usually roughly proportional to problem size, typically a small factor (1 to 10) times the number of constraints [11], *ie* many thousand iterations on a large problem. In practical implementations, it is found that the most computationally intensive parts of the revised simplex algorithm are the *BTRAN*, pricing and *FTRAN* steps - taking perhaps 30%, 35%, and 20% respectively of the total solution time for a large sparse problem, while the re-invert step might consume 5% of the time [37,19].

2.5 Summary

From the descriptions given above, a set of vector and matrix operations which must be efficiently supported by the new machine can be identified. For the Gaussian elimination algorithm, efficient subtraction of a multiple of one sparse vector from another is required, and the implementation must be able to handle the fill-in which occurs. In addition, for pivot choice by the Markowitz criterion, there must be a way of rapidly finding the number of non-zeroes in each row and in each column of the updated matrix. Rapid access to the non-zeroes in the chosen

pivot row is required, in order to check the chosen pivot (if using the threshold pivoting method), and rapid access to the non-zeroes in the chosen pivot column is needed to calculate the scalar multipliers for the subtraction step.

For fast implementation of Linear Programming, the machine must also support efficient implementation of the *BTRAN*, pricing and *FTRAN* operations.

If mechanisms can be developed which support all of the above operations efficiently, in addition to sparse versions of the standard set of vector arithmetic operations provided by a vector processor, then the resulting machine could be expected to support well a wide range of problems involving sparse vectors and matrices. In practice, it is also likely that the use of a machine offering new facilities will develop in unexpected ways, as programmers exploit the new features to solve problems in ways unanticipated by the designers of the architecture.

Chapter 3

Implementing Sparse Matrix Computation

3.1 Introduction

One of the aims of the ESP development was to develop an architecture for a processor supporting a general set of vector instructions, such as might be found on many vector processor machines, but with the facility to work with sparse vector, and mixed sparse/dense vector operands. A second aim was to ensure that the architecture supports all the operations required for the Gaussian elimination and Linear Programming algorithms described in chapter 2, and does so in a balanced way – that is, there should be no 'holes' in the support for the algorithm, which would result in the time for one step dominating the execution time. This requirement has led to the development of new vector instructions, and to hardware to support not just vector operations, but operations on complete matrices.

Close examination of the operations which need to be performed on vectors during sparse matrix computation reveals that there are three basic suboperations which need to be implemented efficiently:

1. Access to all of the non-zero indices, and often the values as well, in sequence. Sometimes the sequence is required to be in order of ascending index.

2. Access to the value of a single element, given its index.

3. Insertion of new non-zero elements, and updating of the value of existing non-zeroes.

This chapter examines the ways in which a variety of data structures for sparse vectors can support these suboperations so as to provide efficient implementations for the required vector and matrix operations. These data structure operations may be implemented in software, on conventional scalar and vector processors, or by using special-purpose hardware additions to standard processor architectures.

Large sparse matrix problems require matrix storage in compressed form – a suitable choice of data structure depends on a balance between the need to minimise the memory space used, and the requirement for fast implementation of vector and matrix operations. All representations save space by discarding the zero values, and, because the memory location of a non-zero in the vector no longer implies its index position, information about the index positions of the non-zeroes must be represented in some other way.

Existing implementation techniques, both software and hardware, are discussed in this chapter, and other options are examined.

3.2 Array storage for vectors

The standard data structure for storing a *dense* vector is what will be referred to here as the *array*[1] – a vector with n elements, whose elements are each held in m memory locations, is stored in mn consecutive locations starting at the location

---

[1]Many publications concerned with vector processing use the term *vector* to refer to both the mathematical entity involved in the computation, and its concrete representation in the memory of the computer. This is confusing when dealing with sparse vectors, for which a variety of concrete representations is possible, even for the same vector at different times within a single program. The term *vector* will here be used exclusively

with address A. The ith element of the vector starts at location $A + m(i - 1)$. Elements may be accessed sequentially in order of index, simply by using an address counter, and because such accesses go to consecutive memory locations, processors can provide increased memory bandwidth for given memory technology by prefetching data from memory to reduce access latency, and by interleaving several banks of memory to increase access bandwidth. Vector arithmetic operations on arrays are implemented by vector processors as single instructions, and prefetching and interleaving are among the factors on which the increased performance of such processors depends. Vector arithmetic using array data structures is implemented on *scalar* processors by tight loops of code, and again the hardware may allow advantage to be gained from the sequential nature of the memory accesses, if the processor has a predictive data cache which accesses blocks of consecutive data words across interleaved banks of memory.

The array data structure is also ideal for operations which require 'random' access to individual vector elements via given indices, although on most computers such randomised accesses cannot proceed at the same rate as consecutive accesses. This is partly because the addresses may not be available sufficiently in advance to allow pipelining of memory access to reduce access latencies (although some recent vector architectures do support pipelined 'random' indexed access into arrays – described in section 3.4.2 below), but more significantly because consecutive accesses now fall into 'random' memory banks, and some of the memory bandwidth increase provided by the bank interleaving is lost.

Some vector processors provide registers capable of holding complete arrays, or sections of large arrays. Both consecutive and indexed access to such registers is very fast. Scalar processors with large caches may be able to store the vector operands for a series of vector operations entirely within the cache, and will gain a similar performance advantage to that provided by vector registers.

---

to refer to the more abstract mathematical entity, while the term *array* will be used to refer to a contiguous block of equal-sized pieces of memory.

While it is of course possible to store sparse vectors in arrays, this is obviously not an efficient use of space. Compressed forms of sparse vector storage are considered in the following sections. There are, however, two situations in which array storage is commonly used in sparse matrix software. A single array is often used to hold a de-compressed copy of a sparse vector while operations are performed on it – examples are given in the sections below. The second situation arises because computers incorporating conventional vector processors provide a greater vector element computation rate for vectors stored as arrays and processed by the vector processor, than for vectors stored in some compressed form, whether these compressed vectors are processed by the vector or the scalar processors. The ratio of the two rates depends on the machine architecture, and in particular on whether the vector processor itself provides any support for operations on compressed vectors, or whether all such operations must be carried out by the machine's scalar processor. This speed difference means that for sparse vectors whose density is above a certain threshold, computation is faster on the vector processor with the vectors stored as arrays, even though many of the arithmetic operations performed are on values of zero, and are therefore wasted. For the **CRAY-1**, for example, the threshold density is around 20% [16]. As a result, sparse Gaussian elimination programs often include provision for switching from a compressed form of vector storage to array storage, when the density of the remaining unfactorised sub-matrix reaches the threshold value.

3.3 The order vector mechanism

The "order vector" method for storing sparse vectors was developed for the **CDC CYBER-205** computer [12], and is supported by hardware in the vector processor of that machine. Sparse vectors are each stored in two arrays. The *data* array holds the non-zero values from the vector, in ascending order of vector index position. The *order* array (known as an "order vector" in CDC terminology) identifies the positions of the non-zeroes within the vector. It is an array of single-bit elements (stored 64 to a memory word) – the bits correspond one-to-one with the elements

of the vector, with a bit set to zero if the corresponding vector element is not present in the value array (*ie* is zero), and set to one if the element is in the value array. Thus, by scanning the order array and the value array, the non-zero values and positions may be identified. The amount of storage space saved by this mechanism depends on the sparsity of the vector, and is limited by the use, in the order array, of one bit for every vector element, zero or not. For vectors whose values are represented as 32-bit quantities, the order vector mechanism reduces the space required by a factor approaching 32 for very sparse vectors, while vectors of 64-bit values are compressed by up to 64 times. This compression factor does not seem large when compared with density figures of 0.1% or less for vectors in large sparse matrix problems – in space terms, the order vector method is efficient only for vectors whose density is above 1% or so.

The order vector data structure is not efficient for 'random' access to vector elements by index – although one may quickly index into the appropriate bit of the order array to determine whether a vector element is present in the value array, determining the value of an element which is present involves a scan through both the order and value arrays. The data structure is much better suited to those operations which always require a scan of the vector, such as the vector arithmetic operations (vector add, scalar product, *etc*), and the **CYBER-205** provides a set of such vector instructions to operate directly on sparse vectors stored in this fashion.

Instructions such as vector add, which have two input and one output operand, contain fields to identify the three value and the three order arrays. The vector processor reads the input value and order arrays from memory and uses the order arrays to identify the way in which the value streams should be aligned and combined. For a vector add operation, for example, the output order array is a bit-wise OR of the two input order arrays, and the output value array contains values copied from one of the two input value arrays if only one of the corresponding input order array bits was a one, and a sum of values from both input arrays if both bits were one. An element-wise vector multiply on the other hand, produces an output value only where the two input order arrays have ones in the

same position. The output order array is a bit-wise AND of the two input order arrays.

In addition to a range of vector operations on sparse vectors stored in this way, the **CYBER-205** provides instructions which support copying of vectors stored as arrays to order vector form, and *vice-versa.*

Measured in terms of output values generated by the vector arithmetic pipeline per unit time, the vector instructions which take order vector operands are slower than corresponding vector instructions with array operands, because of the overhead of fetching and examining the order arrays. As the operand vectors become more sparse, the order vector arithmetic rate becomes relatively slower, because the order arrays are mostly empty of ones, and the arithmetic pipeline is starved while the order arrays are scanned. More relevant, however, is a comparison between the time for a complete vector operation on sparse vectors stored as arrays, and the time for the same operation on the same sparse vectors using the order vector method. The array implementation keeps the arithmetic pipeline busy, but it wastes most of its operations on processing zeroes. For sparse arrays of the densities found in large Linear Programming problems (0.1 to 1%) the order vector method will be more than ten times faster than the array method, but nevertheless the rate of useful arithmetic operations, even using the order vector method, is many times lower than the peak rate at these low vector densities. In terms of vector instruction execution time, the order vector mechanism seems appropriate for vectors with densities between 1% and perhaps 20%.

3.3.1 Handling fill-in in the order vector method

When two vectors stored in order vector form are added, the number of non-zeroes in the result vector can range up to the sum of the numbers of non-zeroes in the two original vectors – the exact number depending on the extent to which the original vector non-zero positions coincide. In the **CYBER-205** order vector implementation, it is the programmer's responsibility to ensure that enough memory space has been reserved in advance for the result value array. All such instruc-

tions must have three *distinct* vector operands, for example $\mathbf{z} := \mathbf{x} + \mathbf{y}$; operations which directly add one vector to another (*ie* $\mathbf{x} := \mathbf{x} + \mathbf{y}$) are not possible, because, depending on the relative positions of the non-zeroes within the two vectors, the writing of the output value array might overtake the reading of the input value array of $\mathbf{x}$.

These considerations mean that code which uses order vector operations must be integrated with code for dynamic management of memory allocation for the value arrays. The latter code must handle reclaiming of arrays which are replaced with larger ones, and the resulting tendency towards fragmentation of the memory area used will require periodic compression of the space used by copying all the value vectors. When the vectors involved are very sparse, the time for the memory management operations, which must be performed by the scalar processor, will dominate the vector instruction execution time. The memory management overhead has meant that the order vector mechanisms provided in the **CYBER-205** have not in practice been used by writers of sparse matrix software, and no subsequent vector machine has incorporated these mechanisms. The order vector mechanisms become very inefficient, in both space and time, for vectors with a density of less than 1%, and without the special vector processor hardware of the **CYBER-205**, the order vector data structure is more costly in both space and time, for *any* vector density, than alternative compressed data structures described below. It as not surprising, therefore, that there have been no reported implementations of this method in software.

3.4 The index/value array mechanism

In the index/value array mechanism, the non-zero values of a sparse vector are again stored compressed into an array, but their original vector index positions are indicated not by an order array, but by an *index* array. The ith component of the index array is an integer specifying the position in the vector of the non-zero whose value is the ith component of the value array. In some variants of this data

structure, the entries in the arrays are always maintained in ascending order of index, while in other variants the arrays do not need to be kept in this order.

Assuming that the index array is an array of 32-bit integers, the amount of space required to store a sparse vector using the index/value array mechanism is clearly more than that required by the order vector method, for vector densities above 3%. However, at lower densities, the index/value array method becomes relatively by far the better, requiring about half the memory space of the order vector method for a vector of density 1%, and less than a tenth the memory space at a vector density of 0.1%.

Like the order vector mechanism, the index/value array mechanism does not efficiently support access to a vector element by index – to find such an element the index array must be searched to determine whether, and if so where, the index occurs in it. Such a search is most simply implemented by a sequential scan, but for vectors with more than a few non-zeroes, stored in ascending order of index, a binary search or interpolation search may be faster. Operations such as vector addition, element-wise multiplication, and scalar product can however be implemented efficiently, in two rather different ways.

3.4.1 In-phase scan implementations of arithmetic

If the index and value arrays are maintained in ascending order of index, arithmetic operations may be implemented by synchronised scans of the index arrays of both vectors. For operations such as add, the index arrays are merged, simultaneously merging the two value arrays; if an index appears in both input arrays, the corresponding values are added. Element-wise multiply, and scalar product, require the intersection of the two index sets to be determined, with corresponding arithmetic operations. Each of these operations may be implemented in software on scalar processors, but the vector hardware provided by conventional vector processors does not support this type of operation.

The problems caused by the occurrence of fill-in are exactly those discussed above (section 3.3.1) – as in the case of the order vector mechanism, operations

which can cause fill-in, such as vector add and subtract, must place their result in arrays distinct from those containing the two input operands, and complex memory management algorithms are required.

3.4.2 Scatter/gather implementations of arithmetic

An alternative method may be used for vector arithmetic operations on sparse vectors stored as index/value arrays, and this second method does not require that the arrays be kept in ascending order of index. The idea is to copy one of the operands into array format before performing the operation. For example, a scalar product between two vectors $\mathbf{x}$ and $\mathbf{y}$ may be implemented on a scalar processor by first converting $\mathbf{x}$ into array format: starting with a suitably-sized array containing all zeroes, the entries of the value array of $\mathbf{x}$ are copied into the positions specified by the index array of $\mathbf{x}$ (this operation is often called *scatter*). The value and index arrays of $\mathbf{y}$ may then be scanned, with the corresponding elements of $\mathbf{x}$ picked up by indexed accesses into the full array version of $\mathbf{x}$, and the scalar product accumulated.

On a machine incorporating a vector processor, the order of computation is likely to be slightly different: once the new form of $\mathbf{x}$ is produced, the entries in it corresponding to the non-zeroes of $\mathbf{y}$ may be collected into a compressed array of values, by using $\mathbf{y}$'s index array to address the array version of $\mathbf{x}$ (this operation is known as *gather*). The answer is then computed by a scalar product operation (available on vector processors as a single `MULTIPLY-AND-ACCUMULATE` instruction, or via linked execution of element-wise `MULTIPLY` and `ACCUMULATE` instructions) between the array of gathered values and the value array of the vector $\mathbf{y}$.

The **CDC CYBER-205** incorporates vector instructions which directly implement the scatter and gather operations. Each instruction specifies two source and one result operand. For the `SCATTER` instruction, the source operands are the two arrays containing the sparse vector non-zero values and indices, while the result operand is the target array (entries in the target array whose indices are not in the source index array are unchanged, so the target should normally have

been preset to all zeroes). The source operands of the GATHER instruction are the index array and the source array, while the result operand is the value array to hold the gathered values. The number of elements processed by these instructions is equal to the number of members in the index array, and the instructions process elements at a slower rate than other vector instructions, due to the irregular pattern of full vector accesses, which prevents advantage being taken of memory bank interleaving. However, the SCATTER and GATHER instructions operate many times faster than equivalent code loops executing on the **CYBER-205**'s scalar processor, and similar instructions have been included in most subsequent vector processor designs. On such a machine, the scalar product between two vectors **x** and **y** stored in index/value array form is calculated with three vector instructions – a SCATTER of **x**, a GATHER using the index array of **y**, followed by a MULTIPLY-AND-ACCUMULATE on the gathered value array and the value array of **y**. An additional SCATTER operation is then normally required to zero out the temporary storage array for future use.

Vector multiplication

Operations producing vector results are more complex. A vector processor implementation of an element-wise multiply between **x** and **y** would start by proceeding as before, with the MULTIPLY-AND-ACCUMULATE instruction replaced by MULTIPLY. The result is a value array of the same length as **y**'s value array, but potentially containing zeroes, due to zeroes picked up from **x** by the GATHER instruction. To place the result in standard index/value form, this value vector must be scanned for zeroes, and these zeroes must be deleted, with the corresponding elements deleted from a copy of the index array for **y**, to form the index array of the result. The first step might be implemented, for example on the **CYBER-205** vector processor, using a COMPARE operation to identify the non-zeroes of the array result of the MULTIPLY – this generates an array of single bits, of similar form to the order array discussed in the previous section. The COMPRESS instruction takes two source operands – a bit array, and an array of 32- or 64-bit values – this instruction may be thus used twice, to remove the unwanted entries in the value array result

of the MULTIPLY instruction, and in the index array for $\mathbf{y}$, to produce the value and index arrays of the final result. Other vector processors which support scatter and gather, such as the **IBM 3090VF**, also have instructions corresponding to the **CYBER-205** COMPARE and COMPRESS. On a *scalar* processor, the gather, multiply, compare and compress operations could be combined into a single loop, and the relative speed of scalar and vector implementation would depend on the number of non-zeroes in the operands, the relative scalar/vector arithmetic speeds of the machine, and the time taken to start up each vector operation.

Vector addition

A vector addition operation is more complex still, because, where the element-wise multiply result has non-zeroes in some or all of the $\mathbf{y}$ non-zero positions, the add result has non-zeroes in all the non-zero positions of $\mathbf{y}$, plus, in general, extra non-zero positions from $\mathbf{x}$. A typical scalar processor implementation, described in [16], of $\mathbf{x} := \mathbf{x} + \mathbf{y}$ involves first scattering $\mathbf{y}$ into an array. The index array of $\mathbf{x}$ is then scanned, fetching the corresponding elements from the array version of $\mathbf{y}$. If an element is non-zero, it is added to the corresponding element of the value array of $\mathbf{x}$, and the non-zero value in the array is reset to zero. After this step, all the elements of the result in positions of non-zeroes in the original vector $\mathbf{x}$ are correct, but there may remain non-zeroes in other places in $\mathbf{y}$ to be added to the result. The index array of $\mathbf{y}$ is therefore scanned next, and the corresponding values fetched from $\mathbf{y}$'s array representation. If the value fetched is zero, this corresponds to a non-zero of $\mathbf{y}$ which has already been added into $\mathbf{x}$ in the previous step. If the value fetched is non-zero, it must be at a different index position from any non-zero in the original $\mathbf{x}$, and since we do not need to keep the result index and value arrays in ascending order of index, the new non-zero and its index can simply be appended to the end of the arrays representing $\mathbf{x}$. As was the case with previous vector addition examples, the program must ensure that sufficient memory space has been allocated for the value and index arrays of $\mathbf{x}$ to expand to accommodate the new non-zeroes.

These steps may be implemented on a vector processor with SCATTER, GATHER, COMPARE and COMPRESS instructions as a sequence of eight vector instructions. Again, the relative speed of scalar and vector implementations depends on the numbers of non-zeroes involved, the relative scalar/vector arithmetic speeds, and the vector instruction start-up times.

3.4.3 Comparison of in-phase scan and scatter/gather methods

Although the in-phase scan methods for arithmetic on index/value arrays are rather more straightforward than the scatter/gather methods, in scalar processor implementations there is little difference in the speed of the two in most cases. However, there are particular cases in which the scatter/gather methods are much faster. As an example, consider a sequence of scalar products between a single vector $\mathbf{y}$ and a number of different vectors $\mathbf{x}_i$. If the vector $\mathbf{y}$ is the one scattered, this needs to be done once only at the start of the sequence, and each scalar product proceeds by considering only the elements of the full array version of $\mathbf{y}$ whose non-zeroes appear in the index array of $\mathbf{x}_i$. This is potentially faster than the in-phase scan method, which would need to look at every entry in $\mathbf{y}$'s index array for each new scalar product, as well as each entry in the index array of $\mathbf{x}_i$. If the $\mathbf{x}_i$ are as dense as, or denser than $\mathbf{y}$, the advantage is small, and may be outweighed by other differences in speed due to the different detailed implementations of the in-phase scan and scatter/gather methods, but if the $\mathbf{x}_i$ are substantially sparser than $\mathbf{y}$, the scatter/gather method will certainly be substantially faster. This performance difference only occurs for a sequence of operations where one vector operand remains the same throughout the sequence, and where each operation can work by considering the index array of just one of the two operand vectors (*eg* scalar product or element-wise multiply). (The *BTRAN* and pricing steps in LP provide good examples of such a sequence of operations.)

Apart from such special cases, the relative speeds of the two methods will depend on their detailed implementation. On a vector machine with instructions

to support the scatter/gather methods, these will generally be faster, but only if vectors contain enough non-zeroes for the increased element processing rate of the vector instructions to outweigh the vector instruction start-up time. The mechanisms described in this section are used in practice in many sparse matrix programs [19,16,32].

The memory management overhead required to deal with fill-in during a computation such as Gaussian elimination, using index/value array storage, is the same whichever implementation is used for the individual vector operations.

3.5 The index/value list mechanism

The biggest difficulty in implementing sparse matrix code using the index/value array method described above is the problem of handling fill-in caused by vector addition or subtraction operations. The task of memory management is complex, involving constant checks that the allocated space for value and index arrays is large enough to hold the potential result of the next addition, allocation of new array spaces as required, and periodic copying of the spaces in use in order to reclaim garbage space in large enough blocks to be useful.

One way of simplifying this task is to store the non-zero indices and values of a sparse vector not in arrays, but in linked lists – usually, for each vector, a single list of records each of which contains the value and index of a single non-zero. Fill-in can now be handled by linking new records into the lists. Memory management is simpler, as it is not necessary to determine, before an addition operation, how much space the result will require, and garbage collection does not require periodic rearrangement of space in use – any space which becomes free can be linked onto a free list.

The detailed algorithms for implementation of vector arithmetic on linked list representations of vectors are exactly analogous to the in-phase scan and scatter/gather methods described above, and because of the simpler memory management, complete sparse matrix code using linked lists is simpler than that using

index/value arrays. However, there are disadvantages to the linked list method. Firstly, the amount of memory space required for linked lists is greater than that of the corresponding index and value arrays, because of the space required for links. Secondly, although the speed of arithmetic operations using the linked list method is not significantly different from the speed of the same operations using index/value arrays when the operations are implemented in *scalar* code, vector processors which support scatter/gather type operations support them only on index and value array representations, not on linked lists. When such a vector processor is available, and the arrays are long enough that vector instructions are faster than scalar code, the array method is preferable. Finally, the linked list method may suffer from 'thrashing', when used in a virtual memory environment, if the records in each list become too randomly distributed through memory.

Linked list storage methods have been used in practice in a number of sparse matrix software designs [16], and in fact, index/value arrays and lists are the only data structures in widespread use for sparse vector computation on conventional scalar and vector machines. It seems unlikely that better data structures, able to confer a significant performance advantage on such machines, remain to be discovered. However, a new architecture which includes hardware support for a sparse vector data structure might be able to use a data structure which would be inefficient on a conventional machine, but highly efficient on the special hardware. One such data structure, requiring a special type of associative memory, has been proposed by J.T. O'Donnell, and others were considered during the design of the ESP architecture.

3.6 Associative memory storage

In [38], J.T. O'Donnell proposes a design for a special purpose memory system for storing sparse vectors in compressed form, which nevertheless supports fast access to single vector elements by index, and also solves the memory management problem associated with fill-in.

The storage format is similar to the index/value array format described above, with vector non-zero elements stored in increasing index order (although in O'Donnell's proposal, the index fields of some of the non-zeroes need not be explicitly stored). Fill-in is handled by arranging for the memory contents to be shiftable through the memory array. In this way, a new non-zero can rapidly be inserted into an existing vector, by shifting the entire contents of the memory beyond the insertion point up by one word. Of course, this means that all vectors stored in memory beyond the vector being updated move position, so vectors cannot be accessed using their address, and instead the entire memory array is made associative, and the start of each vector is labelled with a unique vector identifier which is found by parallel associative matching throughout the memory array when an access to the vector is attempted.

The associative matching hardware is also used to support fast access to a vector element by index. However, because only some of the indices are explicitly stored in the data structure, the associative matching hardware within each memory word must, in this case, *arithmetically* compare the word's contents against the test word and return a `less than` indication as well as an `equal` indication.

The stored format of a small sparse vector is illustrated in figure 3–1. Each memory word in the array can contain a vector identifier, an index, or a value, and each word therefore incorporates a three-valued type tag to distinguish these. Associative matching is performed on the word/type tag pairs. Each memory word also has associated with it three binary flags, which are used by the memory system during memory access. Each kind of memory access, `read element`, `write element`, `return next non-zero`, *etc*, is implemented in a small number of steps which include associative match steps and flag test and set steps. The number of steps involved in each kind of access is constant, independent of the number of vectors stored, and the number of non-zeroes in the vectors.

The manipulation of memory word flags and the required priority encode of the `less than` match results from each word are implemented by a logic tree connected to every cell in the memory array. The size of the tree grows linearly with the

Tag	VI	IX	VA	VA	IX	VA	IX	VA
Data Word	Identifier for vector A	23	Value of A[23]	Value of A[24]	36	Value of A[36]	51	Value of A[51]

(Tags are: VI - vector identifier; IX - index; VA - value)

Figure 3–1: The format of a sparse vector with four non-zeroes, in O'Donnell's proposed memory system

total size of the memory array, while the propagation delay grows logarithmically with memory array size. The time taken to execute each step of a memory access is therefore small, even for large memory arrays, and is independent of the number and size of stored vectors. Thus, for any particular size of memory, each kind of memory access takes constant time, regardless of vector size and density.

O'Donnell's memory system certainly appears possible – a shiftable memory array can easily be designed, and the necessary associative match hardware incorporated with each word. However, by the time the shift control, match circuitry, and the logic tree are included, it seems inevitable that each memory word will occupy several tens of times the silicon area required for the same size word in a standard dynamic memory device.

A further practical problem arises from the need to be able to shift the whole array. If each memory IC were designed as an array of complete words, three word wide buses would be required on the device – one to shift in a word, one to shift out, and one for data input and output from the processor. As double-precision arithmetic is required in many sparse problems, each word must be 64 bits, plus two for the type tag, and so 198 pins would be required for the three data buses alone. Although the bits of a word could be split between two or more chips, it is certainly necessary for all the bits of the word on which the associative match is performed to be in the same device. The vector identifier and index could be limited to 24 bits, and these bits of the word stored with the type tag in one device, with the remaining 40 memory word bits split between perhaps two

further memory devices. Three 26 bit data buses would then be required on the first device. It is clear that, with control signals and logic tree connections also required, each memory chip will need to be packaged in a pin grid array or similar large multi-pin package.

A typical numerical workstation might contain 16Mbytes of memory – enough for most sparse LP problems. This might be implemented using 128 chips packed onto a few 10 cm^2 of circuit board. Because of the much increased memory cell size, an implementation of O'Donnell's memory system would require perhaps 4000 chips to handle similar sized problems. Because the of the large pin-out, each device would be in a much larger package than a dynamic RAM chip, and might occupy 25 or more cm^2 of circuit board space, giving a total of 10 m^2 of board space, without allowing for additional logic tree and control circuitry.

Such practical costs might be justified if the architecture could provide a speed up of a couple of orders of magnitude, but it is not clear that, on typical complete sparse matrix problems, the proposed architecture will confer any performance benefit. Certainly, access to a vector element by index is relatively fast, and memory management is handled automatically, but consecutive access to all the non-zeroes of a sparse vector would be several times *slower* than from index/value arrays on a conventional machine, because of the extra delays in the more complex access method. If there is some performance benefit, it seems certain to be too small to justify the large engineering costs.

3.7 Other data structure candidates

The vector operations which must be supported by a sparse vector data structure involve the following suboperations on individual vector elements:

1. Access to each non-zero index and/or value, one after another – some vector operations require these accesses to return the non-zeroes in ascending order of index.

2. Access to the value of a single vector element (zero or non-zero), given its index.

3. Alteration of a single vector element with given index. This may involve adding a new non-zero, or changing an existing non-zero to zero.

The first suboperation forms part of most vector arithmetic operations (add, scalar product, *etc*). The second is required, for example, during Gaussian elimination, to find the values of the non-zeroes in the chosen pivot column, and during the Linear Programming *FTRAN* operation. Alteration of a single element within an existing vector is required during the *BTRAN* operation, while the writing of a complete vector is required for any vector arithmetic instruction producing a vector result.

Data structures which support these suboperations are required in many application areas, and have been studied extensively (see for example [1,31]). The sparse vector application differs from most in that the first suboperation is required often (*eg* for the row subtractions in Gaussian elimination). Also frequent is the rewriting of a complete vector (the result of a vector instruction), while the individual element lookup and insert suboperations are less frequently used. The two data structures discussed above (sections 3.4 and 3.5), an array of key/data (*ie* index/value) pairs, and a linked list of key/data pairs, are both used in other applications also. These structures support the first suboperation above efficiently, but both are very slow for access to an element by key. Other data structures in common use in applications requiring indexed lookup are tree-based representations such as the *B-tree*, and the *hash table.*

3.7.1 Tree structures

B-trees [1] have the property that elements can be retrieved by key in a time logarithmic in the number of key/data pairs in the structure. Insertion of new elements is also logarithmic, although the worst case insertion time has a large constant factor due to the need to copy data at each level of the tree. Elements

can also rapidly be accessed in key order. However, a vector machine supporting vector instructions operating directly on sparse vectors stored as B-trees would require complex access hardware, as access to an element involves travel down the tree, and, much more problematically, writing a complete vector (for example, the result of a vector add operation) will be slow, as the writing of some of the elements requires copying of many of the previously written elements, to maintain the tree's balance. Tree-based representations were therefore rejected at an early stage in the design of ESP.

3.7.2 Hash table based structures

Hash tables support fast access to elements by key (although performance degrades when the table storage space is nearly full), but there is no way rapidly to access the elements in the table in key order. However, a combination of a hash technique with array or linked list storage of the indices of the non-zeroes in a vector results in a data structure which is fast for all three suboperations listed above. Two separate memories are required, each with different memory access hardware – one holds vector indices, the other values. The indices may be stored in arrays or linked lists, and accesses to them proceed as for the index/value arrays and lists described above. To fetch the corresponding values, a second access is required, to the value memory, which is organised as a hash table. The address of each value is computed by a hash function on the vector identifier and the index. If that function is guaranteed to generate a different address for each index within a single vector (a simple function would be the sum of the index and an integer derived from the identifier), then the data stored at the computed address would be the value itself, plus the vector identifier, so that collisions can be recognised. A rehash function is required, and empty and deleted locations are marked, to support collision handling. A suitable hash function can be computed quickly using combinational logic, and hardware can also handle rehashing when required. Accesses to values would usually be very fast, but subject unpredictably to additional cycles due to collisions.

Such a system would support vector arithmetic operations such as add and scalar product in either of the two ways described for index/value arrays above (in-phase scan or scatter/gather), with each index fetched first, followed by a corresponding value fetch. However, the hash system also supports fast access to a single vector element, by index – if the value is not found in the value memory, it is zero.

3.8 Data structures for whole matrices

Many parts of algorithms for sparse matrix computation involve operations on *vectors* – either rows or columns of the matrices involved – and any of the data structures described above is suitable, with the matrix stored either as row vectors or as column vectors. However, there is one critical step of the Gaussian elimination algorithm, described in section 2.2.5 above, which requires information about both the row and column structure of the matrix. In Gaussian elimination by rows, the natural way to store the matrix is as row vectors. However, at each elimination step, the pivot row need only be subtracted from rows with a non-zero in the pivot column – and in a sparse matrix these will be a small fraction of the rows in the unfactorised sub-matrix. If every row must be examined to discover whether its entry in the pivot column is zero, this search will dominate the actual subtraction steps. It is therefore necessary that the structure of the matrix be stored in such a way that the positions of the non-zeroes in each column can be quickly determined, while continuing to support fast subtraction between rows.

This has usually been done by storing, separately from the row vectors (which in a typical implementation would be stored as index/value arrays or linked lists), an array or linked list for each column, containing a list identifying the positions (but not the values) of the non-zeroes in that column. These lists must be updated as elimination proceeds, and the columns fill in. Alternatively, a two-dimensional linked structure has sometimes been used, where each record contains the value of

a single non-zero, its row and column position, plus links to the next non-zeroes in its row and in its column.

3.9 A vector storage mechanism for ESP

Each of the vector mechanisms described above was considered as a candidate for implementation, with suitable hardware support, in ESP. Two of these mechanisms, the associative memory system and the B-tree storage method were eliminated early in the design. Although both support all the basic suboperations in a balanced way, the overheads involved in the relatively complex access mechanisms would make the memory system several times slower than a standard memory. The associative memory system is also technically impractical, for reasons discussed above.

The order vector method (section 3.3) has been tried before, in the **CYBER-205**, which provides vector instructions which operate directly on sparse vectors stored in that form. As discussed above, it is not efficient, either in time or space utilisation, for very sparse vectors such as are routinely found in Linear Programming problems. A further problem is the need for the programmer to surround each vector operation by code which deals with memory management, to handle vector growth due to fill-in. The **CYBER-205** order vector mechanism was not a success, and has not been repeated in more recent vector processor designs.

This leaves three mechanisms – index/value arrays, index/value lists, and a hash-based system. Most recent vector machines incorporate some hardware support for sparse vector operations using index/value arrays, in the form of scatter and gather instructions, and on such machines, some form of index/value array storage is normally used to implement sparse vector software. However, a sparse vector arithmetic operation, such as add, still requires several vector instructions, because the scatter/gather implementation of vector arithmetic involves a sequence of separate steps. The in-phase scan arithmetic method, on the other hand, has

only a single step (although the iterated operation is more complex) and so it is an obvious candidate for hardware support. Index comparison hardware on the input to the processor's arithmetic unit would allow index/value pairs fetched from arrays in memory to be matched by index (the mechanism is described in more detail in section 4.2). This hardware would increase the effective arithmetic pipeline length, but would not reduce the arithmetic rate at all. Vector operations such as add and scalar product could then be implemented as single instructions operating directly on sparse vectors stored as index/value arrays, and these instructions would be several times faster than current hardware-supported scatter/gather techniques. However, the memory management problem remains – as vectors fill in they must be copied into larger arrays, and the software must arrange for this to be done when necessary.

Using index/value *lists*, the required memory management operations are simpler, and are carried out *within* a vector operation, as each element is processed. Current vector processors do not support scatter and gather operations on linked lists, because their memory addressing hardware only supports (within a single vector instruction) access to locations of an array. However, it would not be difficult to add to the memory addressing hardware the necessary registers to support the reading from memory of complete vectors stored as index/value linked lists, within single vector instructions. If a list of free records is maintained, the memory access hardware can also support the writing of a vector in index/value list form, and so scatter and gather operations can now be supported on index/value lists. With the index matching hardware described in the previous paragraph, vector arithmetic instructions using the in-phase scan method could operate directly on vectors stored as index/value lists. If the result vector is written into records taken from the free-list, fill-in is not a problem, and the only additional memory management operation required is to reclaim the space occupied by the previous copy of a vector which has been updated (*eg* by a vector to vector add instruction), by adding it back on to the free list. This is an operation which can itself be implemented in hardware.

Thus if hardware can be designed to support the reading and writing of vectors

stored as linked lists, and to support efficient garbage collection of old vectors, then the index/value list method becomes much easier to use than the index/value array technique – no memory management software is required. If additional hardware is used to support index matching at the arithmetic pipeline input stage, single vector instructions can implement vector arithmetic directly on sparse vectors stored as linked lists.

One potential problem that remains to be considered is that accesses to an array are always to consecutive memory locations, whereas accesses to a linked list are to scattered locations. As a result, array access can take advantage of memory bank interleaving techniques, and is therefore faster than list access could be, even if the overhead of link following could be eliminated. This difficulty can be overcome by storing index/value lists as linked lists of short arrays of fixed length. Access within the short array can now take advantage of bank interleaving, and if the link to the next short array is stored at the *start* of each array, the address of the next array is available sufficiently in advance that there need be no gap in the reading of elements from one short array to the next. Storing vectors as lists of small arrays also reduces the storage overhead of the links.

If the overheads associated with linked list processing can be eliminated in this way, a storage method based on index/value lists is superior to the index/value array method, because memory management is handled almost entirely by the hardware, within the vector instructions. However, the fundamental drawback of both methods remains – fetching a single element by index from a sparse vector is a very slow operation, requiring a search down the list of indices.

This drawback can potentially be removed by using the hashing technique described in section 3.7.2. In this implementation, the non-zero indices of a sparse vector would be stored using a linked list of short arrays, as above. Values would be stored in a separate value memory, at addresses generated by a hash function of the vector identifier and index. The memory management problem is solved as in the previous case, by writing the index list for the output vector of a vector instruction into space taken from a free list, and reclaiming the space used by out-of-date copies of vectors. The hash-accessed value memory does not require

memory management. The hash calculation can be pipelined with the reading of indices and values, and so does not reduce memory bandwidth, although it does increase the memory access pipeline length, and thus the start-up time for a vector read or write operation. Collisions in the value memory, however, do reduce the memory bandwidth, and the reduction become progressively worse as the value memory fills up. This problem could be solved at the expense of larger memory for any given problem, but there is a more fundamental difficulty with the hash storage technique. The addresses in value memory of the non-zeroes of any sparse vector will, by their nature as hash function results, be scattered through memory in an irregular way, and as a result, bank interleaving will not be effective. Because of this, in any given memory technology, the memory bandwidth for consecutive access to the non-zeroes of a vector will be several times less using hashed storage than for the index/value lists described above.

To be balanced against this reduced bandwidth is the very much increased speed of access to single vector elements. In the Gaussian elimination target application, the main use of single element access is to read the non-zeroes in the pivot column, to calculate the subtraction multipliers. There is one single-element read per vector subtract operation. Extracting a single element from an index/value list requires, at most, a scan to the end of the list, and so will take, at most, no longer than the subsequent subtraction operation, and, on average, less than half the time required for the add. The slow single element access operation will therefore slow Gaussian elimination by less than one third. The lack of memory interleaving in the hash table method would cause a greater performance degradation – the subtraction operations could be slowed down by a factor approaching the number of memory banks in an index/value list implementation.

The Linear Programming target application also makes use of single element access. In the *BTRAN* operation, a single element of a vector is updated at each step, while in *FTRAN*, a single element of a vector must be read at each step. Again, each step also requires a vector operation (scalar product for *BTRAN* and vector add for *FTRAN*), and so the performance penalty of the vector operation for the hash storage method would probably outweigh any gain for the single

element access. In any case, there is a better way of improving the performance of the *BTRAN* and *FTRAN* operations, which can be applied in a system based on index/value lists. Both operations involve many steps (hundreds or thousands for a large problem), but within the *BTRAN* or *FTRAN* calculation, it is the same vector which is updated at each step. That vector is one operand of each scalar product (*BTRAN*) or vector add operation (*FTRAN*). The performance of both these operations can therefore be improved substantially by storing that vector in an array. The vector to be updated is scattered into the array at the start of the *BTRAN* or *FTRAN* operation. Single element lookup and update are now fast – an indexed access into the array. A scalar product between the array and a sparse vector in memory, or the addition of a sparse vector into the array, proceeds at the rate at which the sparse vector index/value elements can be fetched from memory, with corresponding indexed lookups into the array.

Overall, the slower fetching of complete vectors in the hashed implementation is unlikely to be compensated by its advantage in speed of single element access, for the target ESP applications. It was therefore decided to proceed with a design using a form of index/value linked list storage for sparse vectors, and incorporating an indexed addressing mechanism for access to arrays, to improve the performance on operations like *BTRAN* and *FTRAN*. For the reasons discussed in section 3.8, the design also incorporates hardware to handle storage by column of the non-zero pattern of a matrix consisting of sparse row vectors.

Chapter 4

Sparse Matrix Mechanisms in ESP

ESP supports two formats for storage of vectors – the **array** form and the **list** form. Whether a vector is stored in array or list form, it is accessed indirectly via a 56-bit descriptor, which includes a pointer to the start of the vector, plus information about the number of non-zeroes in the vector.

4.1 The array form

This is the standard array mechanism (section 3.2). For operations requiring indirect access into a vector, it is preferable to store that vector as an array, and ESP provides instructions to convert between array and list forms of vectors. For some vector/vector arithmetic operations, such as *add*, there is no advantage to using the array mechanism, unless the vector is 100% dense; if the vector is sparse, the list storage mechanism is always more efficient.

4.2 The list form

Linked List Methods – General Strategy

This is a variant of the index/value list mechanism (section 3.5). The lists are maintained in order of increasing index, and the arithmetic pipeline of ESP includes hardware to support arithmetic operations via the in-phase scan method described in section 3.4.1. Thus vector/vector operations such as $\mathbf{c} \leftarrow \mathbf{a} + \mathbf{b}$ are implemented by streaming the lists **a** and **b** into the vector processor's Arithmetic Unit (AU), using additional hardware in the AU input stage to align **a** and **b** elements with equal indices. In the *add* instruction, illustrated in figure 4–1, if the first index in the **a** list is n, and that in the **b** list is m, then if $n < m$ the first element output is the first index/value pair from **a**. This element is taken into the AU from the list **a**, while the list **b** is unchanged. Similarly for $m < n$, while if $n = m$, the output element is the sum of the elements at the front of the **a** and **b** queues, both of which are taken into the AU simultaneously, for addition.

A different form of input vector index matching is required for vector/vector multiplication operations, for example the *scalar product* operation $\mathbf{c} \leftarrow \mathbf{a}.\mathbf{b}$, as in this case a multiply step is only required if non-zeroes appear at the same index position in *both* input streams. In this case, the arithmetic unit input hardware discards non-matching elements in the input streams.

As has already been noted, the amount of space needed to store a sparse vector in compressed form is not usually known at compile time. Nor is it in general known at run time, even at the start of the operation producing the vector. In the *add* example above, the output list **c** may be as short as the longer input list or as long as the sum of the two input list lengths, depending on the extent to which the positions of non-zeroes in the input vectors coincide. In some operations (*eg* the compression of a vector from array storage format to list format) the range of possible result list lengths may be anything from zero to the length of the original array vector. For the list storage format to be useful, therefore, the amount

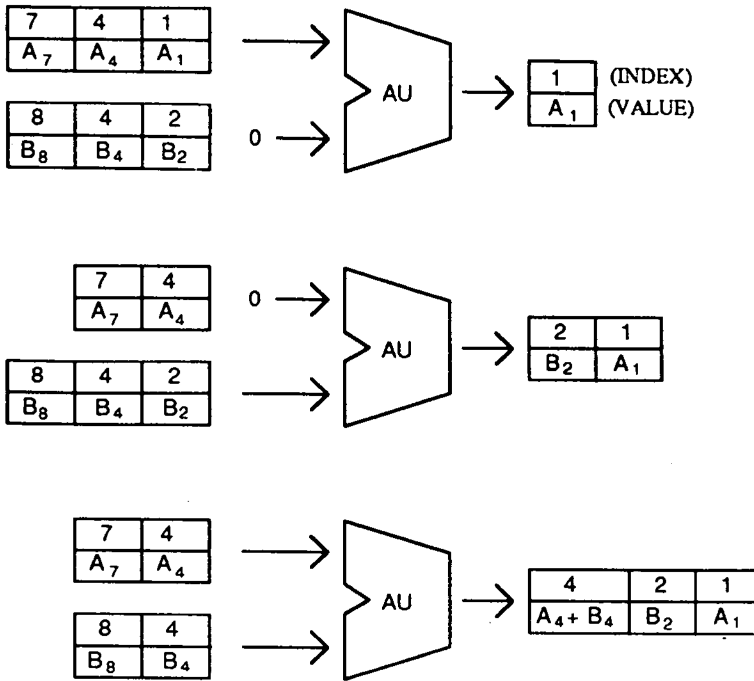

Figure 4–1: An ADD operation on list vectors

of memory space allocated to a vector must be dynamically and automatically variable, and to achieve the required performance, the allocation of space must be performed in hardware. The ESP hardware maintains a pool of free memory space, allocating space from the pool to vectors as required. The index/value list comprising a single vector resides in one or more *linked blocks* of memory locations. If the block allocated to hold the result at the start of an operation turns out to be too small, another block can be linked onto it. The unused portion of the last block allocated may be returned to the free pool. The hardware which fetches list vectors from memory must now be capable of following the links between blocks.

Space may be freed by explicit de-allocation (under program control) of the memory used by temporary vectors which are no longer required. However, more often, it will become free automatically. To see why this is so, consider an operation of the type $\mathbf{a} \leftarrow \mathbf{a} + \mathbf{b}$, and suppose vector **a** has non-zeroes at index positions 100 – 199, while **b** has non-zeroes at index positions 0 – 99. If the result vector is written into the memory blocks already occupied by **a**, then the first 100 output elements will overwrite the 100 non-zeroes of the original **a**. The first few of these

will be in the arithmetic unit input pipe, but most will not yet have been read from memory, as the AU must deal with all 100 non-zeroes of **b** before using up any from **a**. As a result, input elements will have been corrupted before they are read. To avoid this problem, when a vector appears as both an input and result of an operation, the result vector must always be allocated new space, and the original space used by that vector is returned to the free pool automatically at the end of the operation, by a hardware de-allocate operation.

4.2.1 Implementing linked lists in a single-level memory environment

These list structures can be implemented in a simple way by treating memory as a pool of fixed size (small) blocks, each with a single link field. The free space is a linked list of unused blocks. As a vector operation produces its result, that result is written into the first locations on the free list, following links as required. The result vector's descriptor is updated to hold a pointer to the start of the vector list. The unused part of the final block in the result vector is wasted; this is the reason for using small blocks. Reclaimed space is linked onto the start or end of the free list. The hardware required to support these operations is simple, and the operations of allocating new space and reclaiming old space are both very fast, each requiring only the updating of processor pointer registers, and the alteration of two links in memory.

Matrix codes tend to use many operations of the type $\mathbf{a} \leftarrow \mathbf{a}+\mathbf{b}$. For example, every elimination step in Gaussian elimination causes a vector to be re-written, usually with a small increase in density, and for reasons explained above, these re-writing steps involve the allocation of new space for the updated vector, and the reclaiming of space previously used. As space from vectors is reclaimed and later used again by vectors of different length, the blocks on the free list will become thoroughly mixed. As a result, the blocks used to store any vector will become randomly distributed throughout the whole memory space. In a single-level memory environment this may not matter, but in a hierarchical memory

environment it is very likely to lead to thrashing of the paging/cacheing system. Although the prototype ESP does have only a single-level memory, it was felt to be important to investigate the operation of memory management techniques for sparse vector storage which would work effectively in a hierarchical memory environment, *ie* which would, as far as possible, preserve locality of reference.

4.2.2 Implementing linked lists in a hierarchical memory environment

In this variant of the previously described implementation, the free list remains a linked list of blocks of free memory, and allocation of new space for a result vector proceeds as above, except that blocks may now be of any length. Any space in the last block allocated (to a result vector) which remains unused at the end of the operation is left, as a smaller block, on the front of the free list. The key to maximising locality of reference lies in ensuring that the blocks on the free list remain as large as possible, so that the space allocated to a new vector consists of a small number of large blocks; this requires a more complex de-allocation algorithm. In general, a vector to be de-allocated itself consists of a list of blocks, and the de-allocation algorithm must check, for each of these blocks, whether is is *adjacent in memory* to a block (or blocks) already on the free list, and if it is, must merge the blocks. A simple way of achieving this merging de-allocation is used in the prototype ESP; this involves maintaining the blocks in the vector lists and in the free list in order of ascending memory address (*ie* links from block to block are always *forward* through the memory address space). The de-allocation algorithm can then merge the two sorted lists of blocks into a single sorted list by a synchronised scan of the two lists in the obvious way, and at the same time can merge adjacent blocks by checking the end address of each block in the merged list against the start address of the next block.

In ESP, vector elements are held in memory as an index/value pair, in a single memory word of 88 bits (64 bits for the value, and 24 for the index). A list vector is held in a sequence of blocks of consecutive memory words, the final word in the

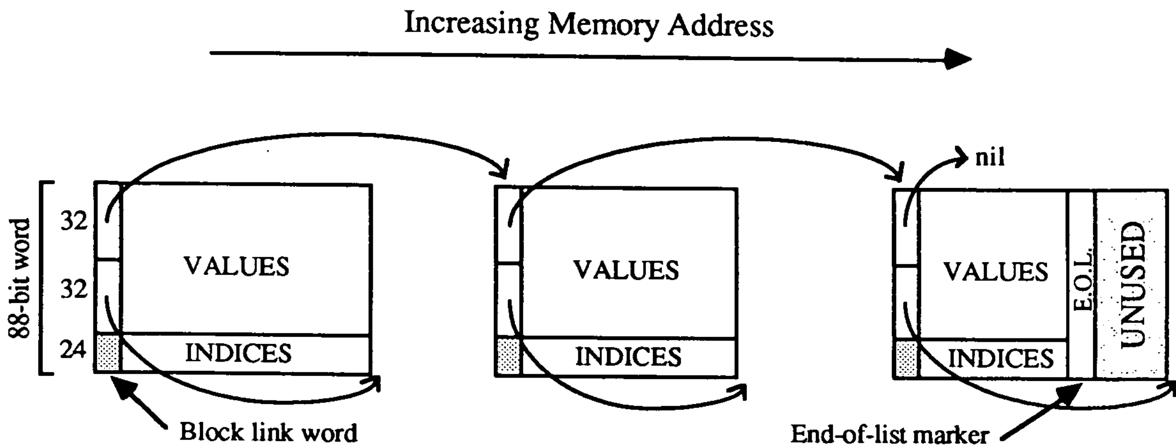

Figure 4–2: A simple structure for a linked list vector

list being a special end-of-list marker. The simplest way to link such blocks would be to make the first word of each block a *block link word* containing *two* pointers (each of which is a 32-bit virtual memory address), called here the *external* pointer and the *internal* pointer (see Fig. 4–2). The external pointer holds the address of the first word of the next block in the list, and is there to maintain the list linkage. The external pointer in the last block of a list holds the special value **nil**. The internal pointer holds the address of the first word *after* the end of the current block, and is there because the de-allocation algorithm needs to know the size of blocks to perform concatenation of adjacent blocks (the de-allocation algorithm is simpler to implement in hardware if these two pointers, rather than the external pointer and a block length count, are stored). Because the blocks are in order of increasing memory address, all pointers point forwards through the address space.

The link word is at the *start* of the block to ensure that the transfer of operand elements between memory and the arithmetic unit can be made as efficient for these list structured vectors as it is for vectors stored in the usual array form. Providing that blocks are above a certain minimum size, there is time to emit the start address of the next block to the memory sufficiently far in advance to avoid a gap in the address generator→memory→AU pipeline. The writing of result elements back to memory may also be effectively pipelined.

Many vector processing systems use interleaved banks of memory to achieve the memory bandwidth required to run the arithmetic unit at full speed and in

ESP, within a block of a list vector, interleaving will work effectively. However, even though the link address is known well in advance, if the first word of the next block falls into the wrong bank, there will be a hiatus in the interleaving. To avoid this, it is sensible to restrict all blocks to starting in a particular bank, and all pointers are thus multiples of the number of banks. For example, in an eight-way interleaved memory, to allow full use of interleaving and pipelining, pointers should be restricted to be a multiple of 16 (rather than eight, to allow sufficient time to send the address of the next block to the memory system).

The algorithm for de-allocating vector space, described below, needs only to access block link words, not the words containing index/value pairs. Because of the alignment of blocks onto the memory banks, all these link words are in the *first* memory bank. To allow de-allocation to proceed as fast as possible, and perhaps concurrently with other accesses to vector memory, the bandwidth of that bank should ideally be higher than that of the other memory banks. The prototype ESP in fact provides this extra bandwidth by providing a completely separate memory to hold the block link words, as illustrated in figure 4–3. Here, the words in the link memory are 32 bits wide, which is large enough to hold one pointer only. In ESP, there are four interleaved banks of 88-bit index/value vector memory, and there is one word of link memory per four words of vector memory. The internal and external pointers of a list vector block which starts at main memory address a are at link memory addresses $a/4$ and $a/4+1$ respectively. Since every block uses two link memory words, the minimum block size is 8 words. Of course, if blocks are large, large amounts of the link memory will be unused, and so the dual memory system introduces an overhead of wasted memory area. However, this overhead is more than compensated for by the main memory bus bandwidth gained, and by the simpler bus arrangements which result from the separation of the two memory types.

The free list is of identical structure to a list vector, and a pointer to the start of it is maintained in a register. Vector descriptors contain, in addition to information about vector size, a pointer to the current position of the first word of the first block in the vector list.

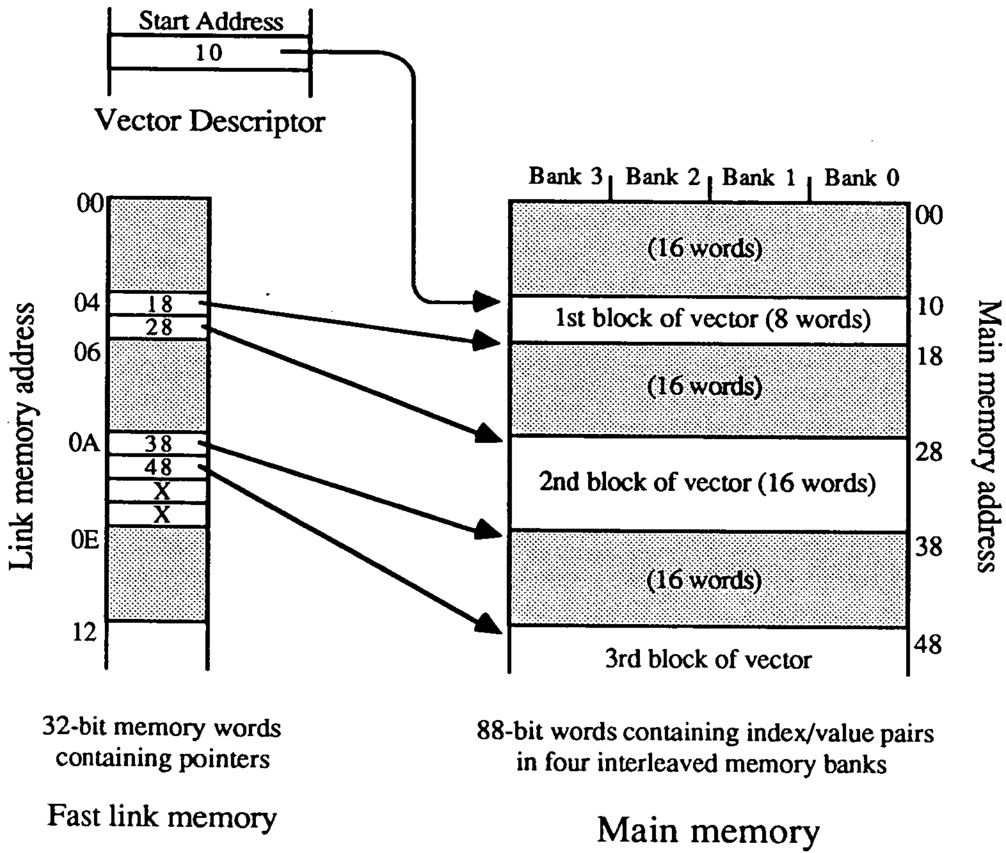

Figure 4–3: The list vector structure used in ESP

4.2.3 Writing list vectors

As the AU produces index/value pairs as the results of a list vector operation, these are always written into the free list. The hardware for performing the allocation of space for a list vector contains four registers: a *write address pointer*, a *block address pointer* which holds the address of the start of the block currently being written into, plus an *internal pointer* and an *external pointer* register, which hold copies of the link words associated with that block. At the start of a vector operation which produces a list vector result, the block address pointer and the write address pointer registers are loaded from the free list pointer register, and the internal pointer and external pointer registers are loaded by accessing the link memory, using the block address pointer. Index/value pairs produced by the AU during the operation are written into consecutive words, with the write address pointer incremented after each write, until the write address pointer is equal to the internal pointer register (see figure 4–4). The current block is now full, and the external pointer register is copied to both the block address pointer and the write address pointer, to prepare for writing to the next block on the free list. The internal and external pointer registers are then updated by loading from link memory, using the new block address pointer. In this way, writing continues through the blocks on the free list. (Hitting the end of the free list, signalled by a **nil** external pointer, aborts the current vector operation and causes a trap.)

Eventually, the AU will signal the end of the list of index/value pairs, by producing an element with the form of the special end-of-list marker. This is written in the normal way. The write address pointer is now forced to the next 8-word boundary, as all blocks must be a multiple of 8 words in length, and a comparison is performed between it and the internal pointer register – this determines how many words remain unused in the current block. If there are no unused words, the free list pointer is simply updated from the external pointer register so that the next complete block is now at the front of the free list. The external pointer word (in link memory) of the last block in the vector just written (which is pointed to by the block address pointer) is updated to contain the **nil** pointer, to signal that it is the last block in the list vector. Alternatively, if there

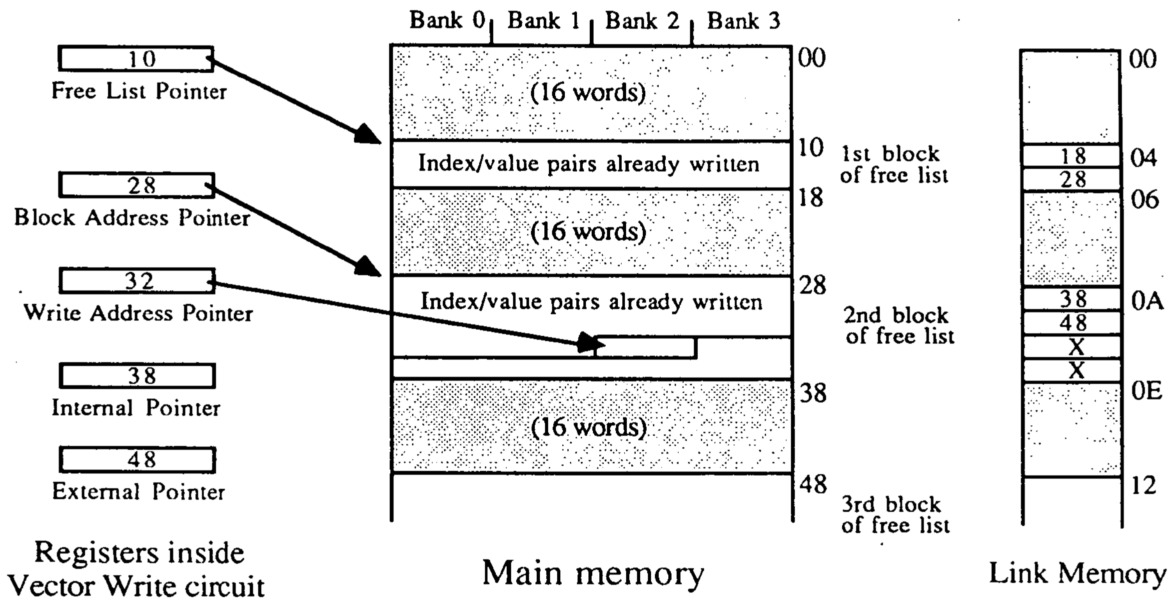

Figure 4–4: Writing a vector in list form

are unused words in the current block, the unused portion of the block is reclaimed for the free list. This involves writing the external and internal pointer register contents into the first two link memory locations pointed to by the write address pointer, to set up the remaining portion of the block as a new block at the start of the free list, and copying the write address pointer into the free list pointer. Finally, the last block in the vector just written has its external and internal pointers updated, using the address in the block address register, to place the **nil** pointer value in the external field, and the write address pointer in the internal field, to indicate the correct block length.

4.2.4 Freeing list vector space

The de-allocation hardware, which reclaims vector space for the free list, requires two pointer registers (`A` and `B`), plus a small number of working registers. The algorithm merges two ordered lists of blocks – the free list, and the vector list being de-allocated. At the start, the free list pointer register is updated to point to the lower of the two list starting addresses. `A` also points to this address; `B` points to the other list. During the algorithm, the pointers `A` and `B` proceed along

the two lists. The external pointers are adjusted to merge the two lists, while the internal pointers are examined to check for contiguous blocks, and updated to merge such blocks.

1. If the first block on list `B` is behind (in memory) the second block on list `A`, `A` is moved to point to the second block on the list.

2. If the first block on `B` is between (in memory) the first two blocks on `A` (but is not contiguous with the first `A` block), the external pointer of the first block on `A` is updated to point to the first `B` block, and the `A` pointer is advanced to the next `A` block. Pointers `A` and `B` are now exchanged.

3. If the first block on `B` is between the first two blocks on `A`, *and* is contiguous with the first `A` block, it is merged with that block. If it exactly fills the memory space between the first two `A` blocks, it is merged with both of them. The pointer `B` is updated to point to the next block on its list.

The algorithm terminates when one list is exhausted, and at this point, all blocks are linked, in order of increasing memory address, onto the list pointed to by the free list pointer. The maximum number of algorithm iterations will equal the total number of blocks on both lists which occupy memory locations between the lower starting address and the lower end address of the two lists.

This de-allocation algorithm is a potential bottleneck in the system, as a de-allocate must be performed after most vector operations. How long this takes depends on the number of blocks on the free list and on the vector list being disposed, and on the start and end positions of both lists; in the worst case, the algorithm must examine every block on both lists to complete the de-allocation.

However, note that the list restructuring remaining to be performed at any time during de-allocation takes place *beyond* the locations pointed to by `A` and `B`. Since the new free list pointer is set up at the *start* of the de-allocate, writing the result of the next vector operation into the free list can commence almost immediately after any pending de-allocate has started, and can continue concurrently with the

de-allocate, subject to the condition that each free block used by the write must start at a memory address *less than* the value of pointer A (which in the particular algorithm used, is itself always less than B). If this condition fails, the write must be delayed until it is again satisfied. In this way, so long as the de-allocation of vectors normally takes less time than writing them, de-allocation need not necessarily delay the processor.

An alternative, more complex, de-allocation strategy has also been considered; this takes a time proportional to the number of blocks in the vector being de-allocated, independent of the number of free list blocks. It is based on a memory management algorithm in [31]. In this case, blocks contain linking information at *both* ends; this consists of an external pointer, and an internal pointer, at each end of the block. (Because four words in link memory are used in every block, the minimum block size in this case is 16 words.) Each of the two internal pointers points to the other end of the block. The external pointer at the front of the block points forward to the next block on the list, while that in the at the end of the block points back to the previous block on the list. The top bit of each external pointer contains a tag bit which indicates whether the block is on the free list or is part of a vector (this restricts the pointer size to 31 bits). Lists need no longer be maintained in order of memory address, and by examining the tag bits, it is possible for the de-allocate algorithm to perform merging of adjacent blocks, and complete the de-allocate in time proportional to the number of blocks being de-allocated. Although the operation of de-allocating a single block is more complex, in some cases this method will be much faster than the algorithm described above. Since the lists are unordered, however, any stage of the de-allocate may affect the first block on the free list (by merging another block with it), and so it is no longer possible to start another write operation until the de-allocate is complete. The prototype ESP incorporates memory management circuitry flexible enough to experiment with both algorithms.

4.3 Indirection and vector registers

List vectors do not waste store, and they provide immediate identification of the non-zero positions of the vector. An operation like the Gaussian elimination step $\mathbf{A}_{i\bullet} \leftarrow \mathbf{A}_{i\bullet} - s_i * \mathbf{A}_{p\bullet}$ will execute efficiently using list vector storage and a single ESP vector instruction, which operates by streaming operand elements into the arithmetic unit in the manner described above. This is also true of other vector/vector operations, such as multiply, on vectors in list form, if both vectors are of roughly equal sparsity. However, in, for example, the *pricing* step of the Linear Programming algorithm (section 2.4.2), one vector (the price vector) is much denser than the vectors into which it is multiplied. To execute the multiplication by streaming in two list vectors would be inefficient, as most of the elements of the price vector would be discarded because there is a zero in the corresponding position of the column vector. If one vector is several times denser than the other, the scalar product operation is more efficient with the denser vector stored in *array* form, using ESP's *indirect* access mode. Using this mode, the sparser vector (stored in list form) moves into the vector unit element by element, and the index fields of the elements are used to offset into the denser vector (stored in array form), using indexed addressing. Obviously, access to the elements of the denser vector is slower than streaming a vector out of memory, because the elements accessed are in non-consecutive locations, and memory bank interleaving will be interrupted, and so this method of access is only preferable where the sparsity of the two vectors differs by a factor of four or more.

Whether it is worthwhile expanding a vector stored in list form to array form for an operation like the scalar product will depend how many such operations will be performed on the vector before it must be recompressed to list form. In the case of the LP pricing step, many hundreds of scalar products will be performed with the same price vector, so it is clearly worthwhile having the price vector in array form. Performance on list vector/array vector operations can be further increased, if the same array vector is to be operated on many times over, by providing a fast

access vector register near the arithmetic unit, to hold the array vector operand. This reduces the memory bandwidth required, and, through the use of fast memory technology for the register, avoids problems related to the failure of memory bank interleaving.

The usefulness of such a vector register is even clearer in the case of the Linear Programming *BTRAN* and *FTRAN* steps. *BTRAN* also requires a scalar product of a sparse vector (the η-vector) with a vector which is (for most of the *BTRAN* steps) more dense, and then requires that one element of that denser vector be replaced with the result of the product. This final step requires access to an element of the vector with specified index, and is clearly very inefficient on a list vector. However, it can be carried out with ease if the vector is stored in array form in a register. The result of the complete series of *BTRAN* steps is the price vector, which is thus conveniently in the register ready for the pricing step.

FTRAN requires the addition of a sparse η-vector (scaled) to a vector which for most of the *FTRAN* steps is denser than the η-vector. The scaling factor to be applied to the η-vector is an element of the denser vector, specified by its index value. It is therefore useful to store the vector being updated in array form in the register, to allow rapid indexed access to these scaling elements.

ESP contains a single, large (32K element) vector register, which may be partitioned into any number of smaller sections, and thus used to store many, shorter, vectors.

4.4 The Sideways List Unit

The need for keeping track of both the row structure and the column structure of the matrix, during Gaussian elimination using threshold pivoting, was discussed in section 3.8. To support this, an extra facility has been added to the vector processor in ESP. Known as the *Sideways List Unit (SLU)*, this maintains lists of non-zero *indices* (but not values) by column, as the elimination proceeds row

by row (ignoring cancellation of non-zeroes during subtraction steps to form new zeroes – something which is rare during the solution of most types of problem).

The total number of entries in these column lists is the number of non-zeroes currently in the matrix, and is equal to the number of entries in all the row vectors. Thus as many words are required to store the SLU lists, as for the rest of the data in the problem. However, the size of word required is only 56 bits (24 bits for the index and 32 bits for the list link), while each row element occupies one word of 88-bit vector memory. Although it seems conceptually neater to have a single memory for all vector data, using main vector memory for the SLU lists increases the vector memory bandwidth requirement, and complicates the memory bus structure. A simpler solution, adopted in the prototype machine, is to provide a separate SLU memory, with the same total number of words as the vector memory, but only 56 bits wide. This memory is written to by the SLU alone, and is only read by the scalar processor. It is not used at all for applications which do not perform dynamic pivot choice on sparsity grounds.

For each column in the matrix, a list of column non-zero positions is kept in the separate SLU memory as a linked list of single memory locations, each holding a non-zero position (24 bits) and a link to the previous word on the list (32 bits), as illustrated in figure 4–5.

The SLU non-zero lists are updated during Gaussian elimination in the following way. A single Gaussian Elimination step consists of the operation $\mathbf{A}_{n\bullet} \leftarrow \mathbf{A}_{n\bullet} - s_n * \mathbf{A}_{p\bullet}$. Whenever the arithmetic unit produces a non-zero in an index position in the result vector $\mathbf{A}_{n\bullet}$ which contained a zero in the input copy of $\mathbf{A}_{n\bullet}$, that element is a new non-zero in the matrix, and its index, i, (its column position in the matrix) is passed to the SLU. The SLU contains a register holding the current row number n (this register is loaded from a field in the vector instruction), and two registers for each column of the matrix, one (the *count register*) holding a count of the number of non-zeroes in the column, and the other (the *address register*) holding, for each column, a pointer to the last non-zero position record which was added to the list of non-zero positions in that column of the matrix. On receiving a new index i from the arithmetic unit, the SLU increments

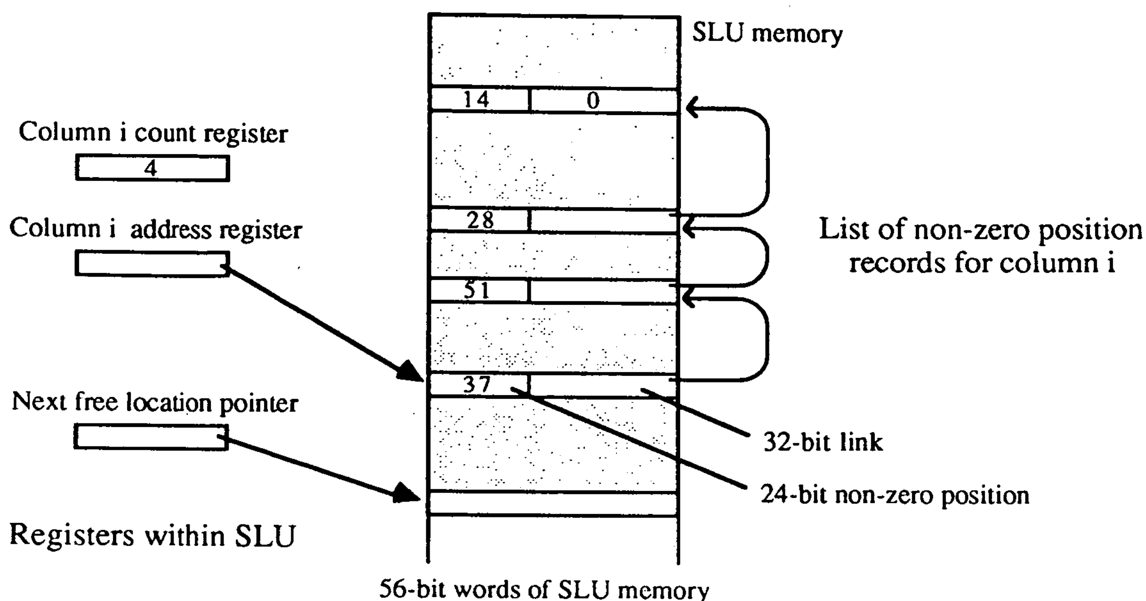

Figure 4–5: An SLU non-zero position list

the non-zero count for column i, and adds a non-zero position record for the row number n onto the non-zero list for column i, by writing the row number (from the row register), together with a link to the previous non-zero on the list (from the address register for column i), into the next free word in the SLU memory. It then updates the address register for column i, to point at the word just written.

The information maintained by the SLU is accessed by ESP's scalar processor during pivot choice. The new non-zero counts for all the columns in which there were non-zeroes in the previous pivot row (these are the only columns which can have had extra non-zeroes added during the elimination steps with that pivot row) are read from the SLU to enable pivot choice by the Markowitz criterion. The address of the end of the non-zero position list for the chosen pivot column is then read from the SLU by the scalar processor, and the scalar processor can then determine the position of all the pivot column non-zeroes by reading the list directly from the SLU memory, following the links, until a link address of zero is found. The *values* of the non-zeroes must be found by using the vector pipeline to search down the relevant rows until the correct column index is reached (using a vector instruction which extracts from a vector the element with a specified index).

Before the elimination steps start, the SLU must be fed the positions of the non-zeroes in the original matrix, and this is achieved by a special vector instruction which zeroes all the count and address registers in the SLU and sets the SLU memory pointer to the integer value one, followed by vector instructions which stream each row vector of the matrix through the vector processor without modification, but adding all elements onto the non-zero position lists maintained by the SLU.

4.5 Summary

This chapter has described the general mechanisms used in ESP to store, access and compute with sparse vectors and matrices. The following chapter describes the ways in which these mechanisms have been integrated into a complete computer architecture.

Chapter 5

The Architecture of ESP

5.1 General structure

The innovative part of ESP is the vector processor, in which are implemented the mechanisms described in the previous chapter. These are intended to confer, for irregular sparse matrix problems, the performance advantages associated with the use of conventional vector processor architectures on non-sparse problems.

The hardware required for fetching and operating on vector data is very different from that required for general purpose operations on typical scalar data types. For this reason, vector processors are built as specialised *co-processors*: they do not implement any scalar instructions or program flow control operations, and they contain no instruction fetch hardware [25,28]. Instead, the main, scalar, processor controls program execution, and passes to the vector processor any vector instructions encountered in the instruction stream.

This co-processor arrangement may be distinguished from the commonly occurring provision of multiple operational units within a CPU data-path by the relative looseness of the coupling between the main processor and the co-processor. In contrast to, for example, an additional closely-coupled floating point arithmetic unit within the data path of a CPU, a vector co-processor will usually provide for:

- fetching and storing its own data from/to memory;

- an independent register bank which may not be directly accessible by the scalar processor data path;

- rather looser synchronisation between the scalar processor and the vector unit – the scalar processor will usually be able to continue execution of instructions encountered after a vector instruction, even though the vector instruction has not completed execution. The extent of this decoupling varies from machine to machine. In the **CRAY-1**, the scalar and vector processors share a single floating-point arithmetic unit, limiting the extent to which scalar instructions may proceed after a vector instruction is started, whereas on the **CYBER-205**, once a vector instruction is started, subsequent scalar instructions may execute until one is encountered which accesses main memory.

Because synchronisation between the scalar and vector processors is enforced on the time-scale of short instruction sequences, all the performance advantages of the vector architecture will be lost unless the scalar processor is sufficiently fast. For this reason, care has been taken to provide a fast scalar processor in ESP. In addition, the interconnection between the scalar and vector processors is necessarily complex, to provide support for rapid instruction and data transfer, and synchronisation. This precluded use of a ready-built computer as the scalar part of the machine, and the performance requirement meant that the scalar processor could not be built from one of the more common microprocessor parts available at the time of design, and has been implemented using a relatively uncommon fast microprocessor chip set.

It would have been possible to build ESP as a stand-alone super-computer. However this would have entailed the provision of various I/O sub-systems, including disk and terminals, and of a full operating system including filing system *etc.* A preferable alternative was to connect the machine to an existing computer or network of computers which can provide most of this functionality. One option would have been to integrate ESP into the Departmental Ethernet network, running TCP/IP protocols, and to write handlers for NFS file service protocols,

standard remote terminal protocols, and so on. This in itself represents a great deal of work, however, for a new machine for which everything from assembler upwards must be written from scratch. To avoid this, it was decided to operate ESP as a back-end processor to one computer, a workstation, on the Departmental network. The connection between the host workstation and ESP can then be a dedicated link, with much simpler protocols than those needed for a direct ESP/network connection. Front-end programs running on the workstation download code and data to ESP, and start and stop ESP execution. Input and output to and from programs running on ESP is via the workstation. The workstation connection has been made as fast as possible by using a multi-wire parallel link, and, at each end of the link, a control microprocessor handles transfer of data and control information.

The general structure of the machine is shown in figure 5–1. Triple buses (two read buses and one write bus) connect the vector processor to the vector memory, to provide the required vector processor/memory bandwidth. Dedicated buses connect the scalar processor to the program memory and the scalar data memory, while a further 32-bit wide bus allows the scalar processor access to all other memories in the machine. Vector instructions are also transferred, via this bus, from the scalar to the vector processor. The same bus allows the control processor full control over the entire machine.

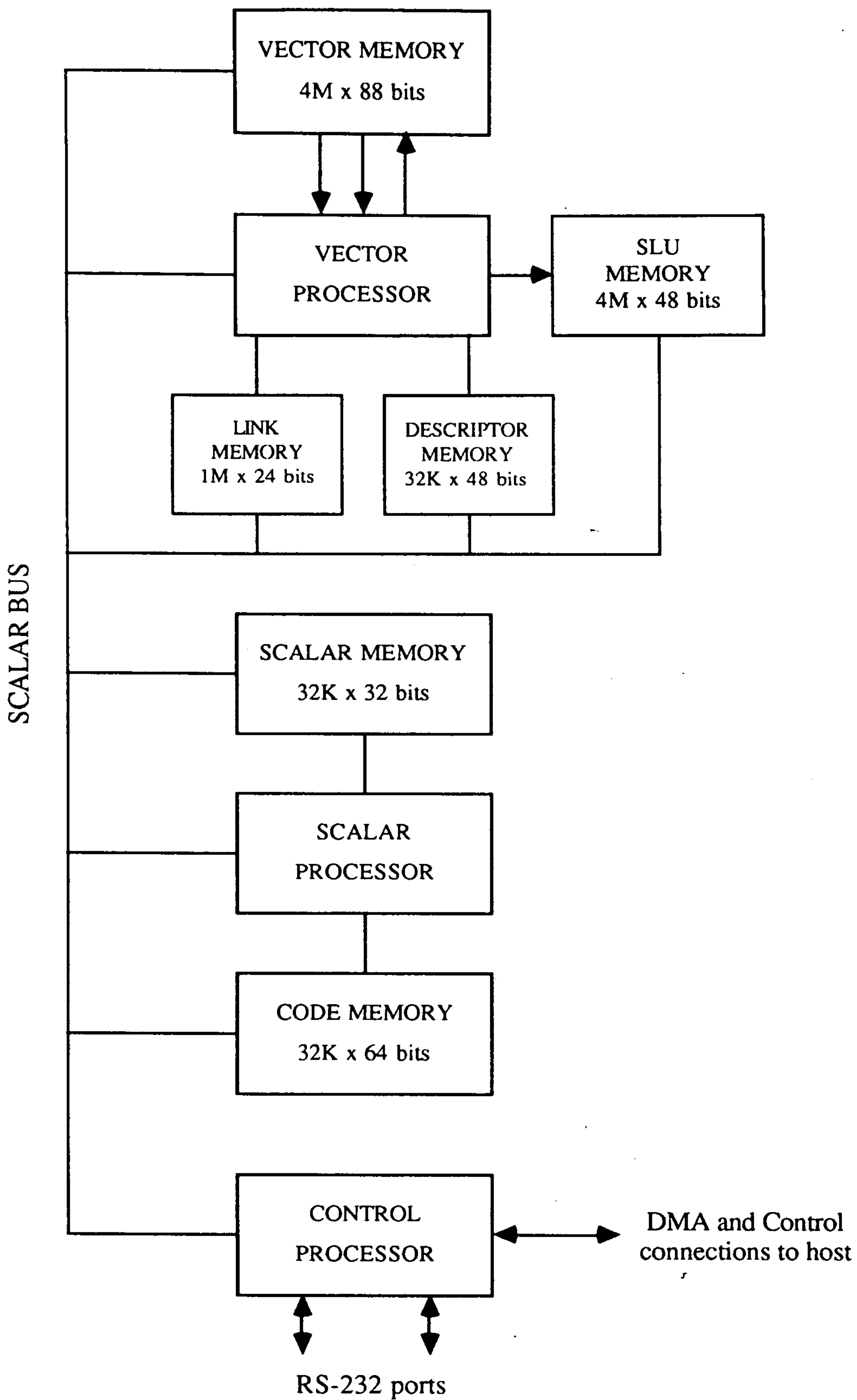

Figure 5–1: The general structure of the Sparse Processor

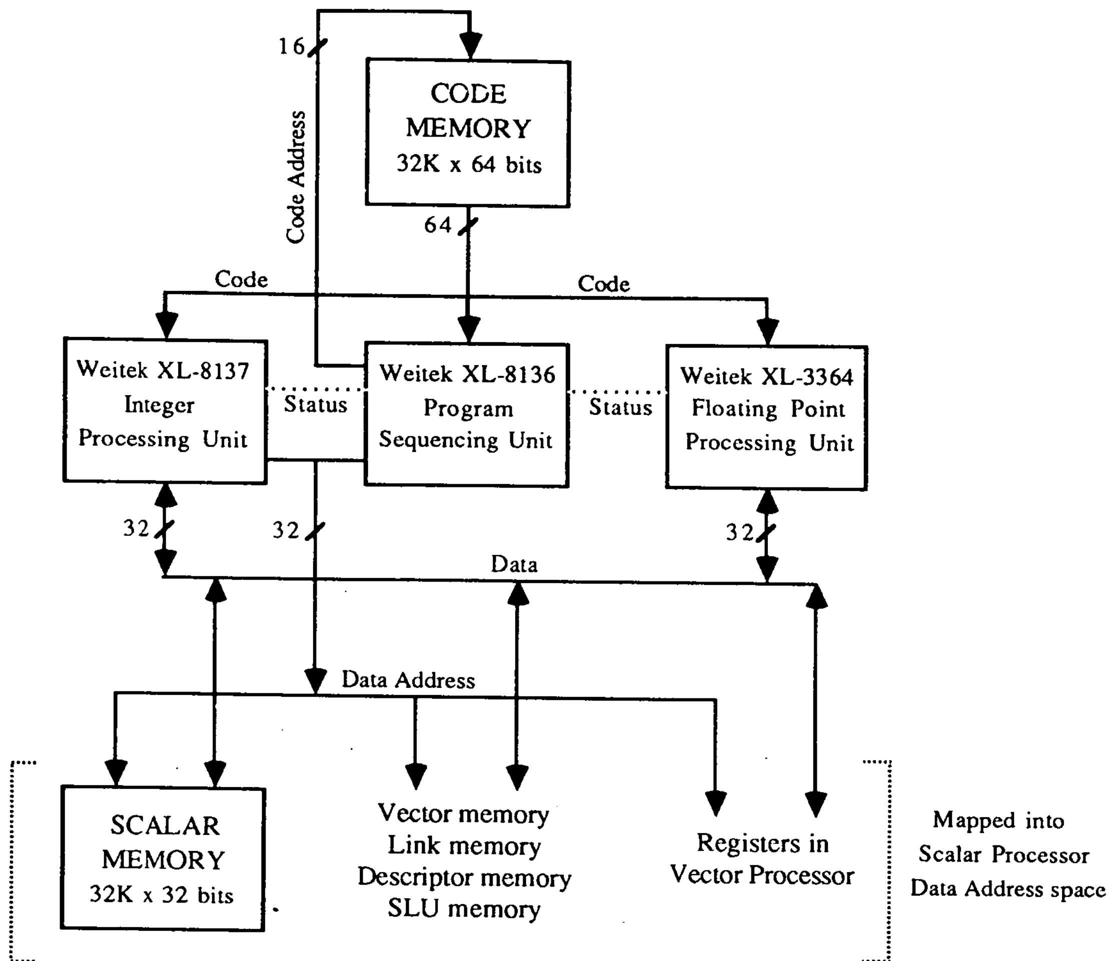

Figure 5–2: The architecture of the scalar processor

5.2 The scalar processor and associated memories

The scalar processor architecture is illustrated in figure 5–2. The 3-chip **Weitek XL-8364** processor set forms the basis of the scalar processor. The chip set implements a "Harvard" architecture, with separate code and data memory spaces – the code memory is 64 bits wide, and in the prototype, consists of 32K words. One 64-bit instruction can be fetched every clock cycle. The code memory effectively contains both scalar and vector instructions – the latter are passed to the vector processor for execution, in a manner described in section 5.4.1 below. The scalar processor chip set is described in detail in the Weitek manuals [48,49,50].

The data memory space of the scalar processor is 32 bits wide. There is a small (32K word) scalar memory, which may be accessed by the scalar processor in a single clock cycle. In addition, the much larger vector memory, the link memory, the descriptor memory, the Sideways List Unit memory, and a number of registers within the vector processor, are mapped into the scalar data address space, to allow access by the scalar processor (and by the control processor).

5.3 The vector processor

The architecture of the vector processor is illustrated in figure 5–3 (the link and descriptor memories are omitted, for clarity). It comprises instruction decode and control circuits, plus a pipelined data path consisting of parts which carry out each of the operations identified in chapter 4, as follows:

- the *Vector Read (VR)* circuit fetches vectors held in either list or array form from vector memory, following links between blocks of list vectors, as appropriate. The VR circuit also supports indexed (*ie* non-sequential) access into array vectors.

- the *Index Match (IM)* unit is responsible for matching the two input vector streams, and supplying the implicit zeroes where required, to provide the arithmetic circuits with a stream of index-matched value pairs.

- the *Arithmetic Unit (AU)* contains integer and floating point add and multiply pipelines, plus a bank of registers for scalar input and output operands.

- the *Vector Output (VO)* circuit deletes new non-zeroes, formed by cancellation in the Arithmetic Unit, from the output vector stream before it is passed to the Vector Write unit or to the Vector Register.

- the *Vector Register* is a 32K by 64-bit register, which can hold one or more vectors in full form. The register feeds into the vector pipeline at the Index Match stage, and is itself fed from the Vector Output stage.

- the *Vector Write (VW)* circuit writes vectors back into main memory, either into the pre-allocated, fixed size, space for an array vector, or into space from the free list, for a list vector. Indexed writing into array vectors is also supported.

- the *Garbage Collection (GC)* circuit merges, into the free list, list vector spaces which are no longer required.

- the *Sideways List Unit (SLU)* maintains counts of non-zeroes in each column of a matrix during Gaussian elimination by row, and keeps updated lists, in the dedicated SLU memory, of the positions of the non-zeroes in each column.

The vector processor pipeline is divided into three independently controllable sections, for reasons explained in section 5.3.2. The first section comprises the *Vector Read* circuit. The second section, the *Arithmetic Section*, consists of the IM, AU, VO and vector register parts. The third section, the *Vector Write* section, consists of the VW, GC and SLU parts.

5.3.1 Vector instructions

Vector Types

The two basic vector storage forms described in chapter 4, **array** and **list**, are supported. Array vectors are each stored in a single block of memory words, the start address of which is specified in the *descriptor* of the vector (see below). The value field of the first memory word holds the value of the first element of the vector, the second word holds the value of the second element of the vector, *etc.* The index field of the first memory word holds 1, that of the second word holds 2, *etc.* The number of elements in the vector is given in the **non.zero.count** field of the descriptor. However, the total number of memory locations used in storing the vector is (non.zero.count + 1), rounded up to the next multiple of four. This is a consequence of the way the four memory banks are accessed. The start address of a vector is always a multiple of four. The first non.zero.count words hold the

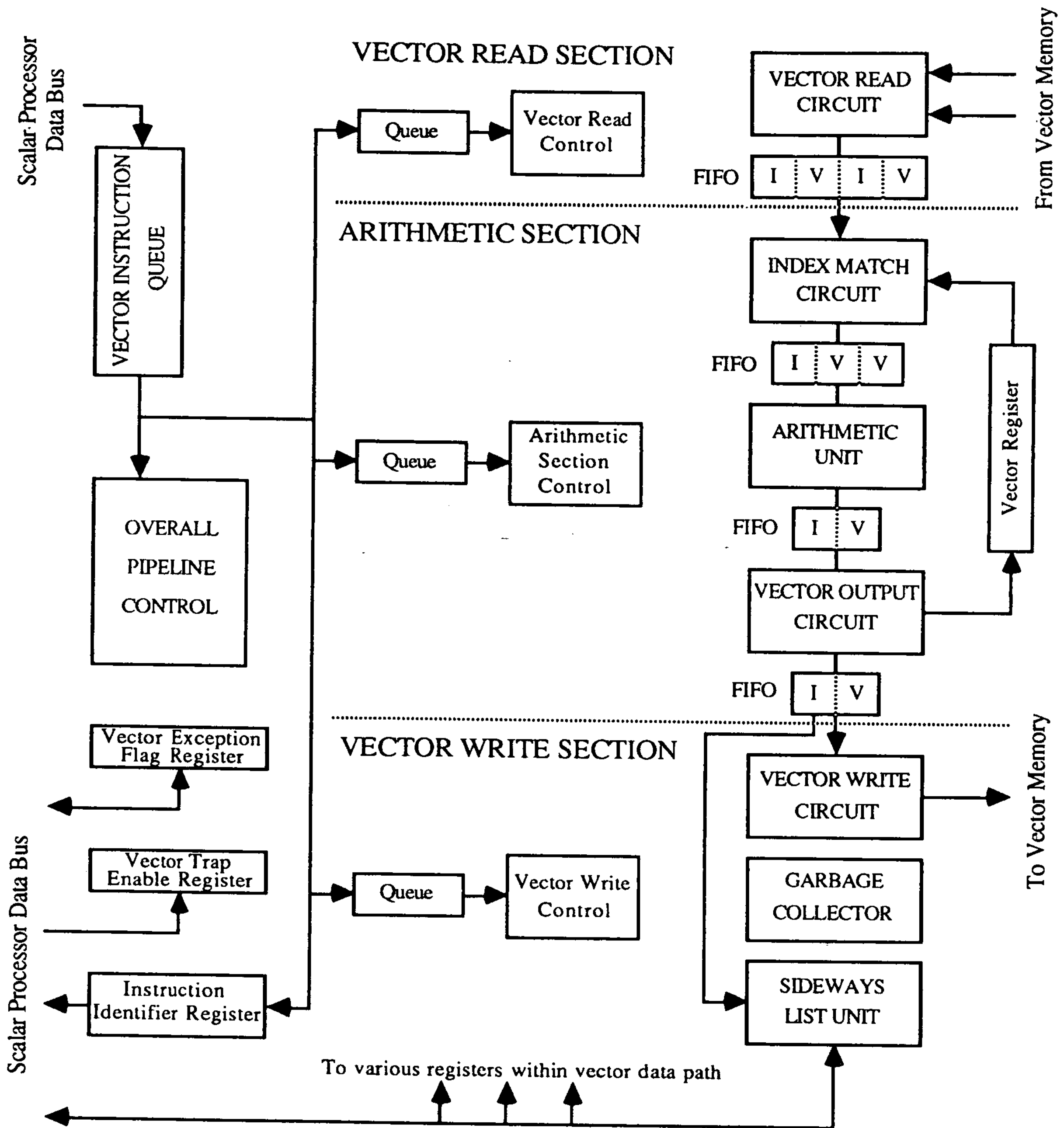

Figure 5–3: The architecture of the vector processor

vector elements and their indices (1 to non.zero.count). The next memory word holds an **end-of-vector** marker, which is identified by an index field of all ones (the value field is not defined). Any words remaining to bring the number of words to a multiple of four are undefined.

A list vector may be spread over several non-contiguous memory blocks, each of which is some multiple of eight memory words in length. Elements whose value is zero do not have to be stored explicitly. The start address of the vector (and of each linked block of elements) is a multiple of eight. After the last non-zero vector element, there is an end-of-vector element defined as above, and any remaining words to bring the total length to a multiple of eight are undefined.

The following data types are supported for both array and list vectors:

- 32-bit IEEE floating point
- 64-bit IEEE floating point
- 32-bit signed integer
- 32-bit logical

Vector Descriptors

All vectors are accessed by reference to a vector descriptor. Descriptors are stored in a separate descriptor memory (capable, in the prototype, of storing 32K descriptors), which is accessible to the Vector Read and Write circuits, and to the scalar and control processors. Contained in the descriptor are the following fields:

- Vector start address (32 bits)
- **non-zero.count** (24 bits) – for an array vector, this is the number of vector elements; for a list vector it is the number of elements explicitly stored in memory (which will normally all be non-zeroes, although there is no reason why zeroes should not be stored also).

Vector Instruction Formats

Each vector instruction contains some or all of the following 32-bit words (see Appendix figure A–1):

- A control word comprising fields specifying the operation of VR, IM, AU, VO, VW, GC and SLU (always present)
- A 24-bit integer **index.count** word (always present)
- A 32-bit **instruction.identifier** field (see section 5.4.2 below – always present)
- For each vector operand (input or output) held in vector memory, a 15-bit integer giving the address of the vector descriptor in descriptor memory.
- If the vector register is used as an input and/or output operand of the instruction, a 32-bit **register.offset**
- If a scalar input operand is required, the operand itself as an immediate value of one or two 32-bit words
- If the SLU is to be used, a word containing the 24-bit **row.number** to be loaded into the SLU row register at the start of the instruction.

The presence or absence of each of the optional fields may be determined by examining the value of the control word.

index.count specifies the notional number of vector elements to be processed by the instruction (the real number of elements processed can be much less, if the operands are sparse vectors stored in list form). The **register.offset** specifies the index position within the 32K-element vector register at which a vector register operand starts. Provision of this value allows the 32K-element vector register to be partitioned into any number of smaller registers.

Scalar input operands are always provided as literal values in the instruction, rather than via more normal addressing modes. Although this may appear to prevent the use of any values other than compile-time constants, in fact, because of the way vector instructions are passed from the scalar to the vector processor no such restriction occurs (more details are given in section 5.4.1 below). Scalar *output* operands, *eg* the sum produced by a vector element accumulate instruction, are always passed back to the scalar processor via a hardware queue, the *Scalar Result Queue*, readable by the scalar processor.

Further details of the vector instruction set are given in section 5.6 and in Appendix A.

5.3.2 Instruction decode and control

The sparse vectors which arise in some important sparse matrix problems (*eg* Linear Programming) typically have very small numbers of non-zeroes, perhaps 6 to 10, even though the matrix order may be many thousand. In many vector operations, the number of pipeline clocks required for the data to pass any point in the pipeline is equal to the number of non-zeroes, and the small numbers involved mean it is very important to minimise the *start-up* time of these operations. During the initial design study, it soon became clear that the total number of pipeline stages through the Vector Read, Arithmetic Section, and Vector Write stages was likely to be at least fifteen, and probably over twenty. If only one vector instruction could execute in the pipeline at any time, the instruction start-up time could not be less than the length of the pipeline, and so performance on problems with very sparse vectors would be a small fraction of the available arithmetic bandwidth.

To deal with this problem, it was decided that the pipeline would be split into independently-controllable sections, so that, potentially, as many instructions as there were pipeline sections could be executing at once. The start-up time is then limited by the pipeline length of the longest section. Obviously, the more independently controllable sections, the shorter the start-up time is likely to be, but the more complex the control hardware. On balance, it was decided to partition

the pipeline into three sections: the Vector Read (VR) section, the Arithmetic Section (AS) (the Index Match circuit, Arithmetic Unit, and Vector Output circuit), and the Vector Write (VW) section (the Vector Write circuit, the Garbage Collection circuit, and the Sideways List Unit). To the author's knowledge, this method of reducing start-up time has not been employed in any previous vector processor design. It has some similarity to the control arrangement in a pipelined *scalar* processor, where a control pipeline runs in parallel with the data pipeline, so that each stage in the processor pipeline is executing part of a different instruction. The difference between such an arrangement and that in ESP is that the ESP architecture further decouples the pipeline sections by using data queues between the sections, and separate instruction queues for each section. Thus it is possible for one section of the pipeline to run ahead of the next section by more than one instruction. Similar decoupling mechanisms have been proposed for the different parts of a *scalar* processor (instruction fetch, operand fetch and data manipulation), and in that context the arrangement has been termed a *decoupled architecture* [44].

The vector pipeline control circuits process a short queue of vector instructions, passing the front instruction in the queue into the VR control circuits as soon as the data stream for the previous instruction leaves VR (*ie* the last data item enters the IM input queue). Similarly, the instruction is forwarded to the control circuits for AS, and then VW, as the previous instruction clears those sections of the pipeline. The queues in the data paths between each pipeline section, handshake signals between the control circuits for each of the three sections and the overall pipeline control circuit ensure that instructions and associated data flow through the sections correctly.

5.3.3 The Vector Read circuit

For each vector instruction, the Vector Read circuit may access from vector memory zero, one or two vector operands (according to the VR mode specified in the vector instruction). The VR circuit is also responsible for fetching the start addresses

of the vector operands from the vector descriptors held in the vector descriptor memory. The descriptor addresses are specified in the vector instruction.

The Vector Read circuit output path comprises a pair of 88-bit paths to the input queues of the Index Match circuit in the Arithmetic Section. Each 88-bit output path may carry a stream of index/value pairs – most VR modes produce one or two index/value output streams. (A vector operand stream consists of a stream of index/value pairs, in which the indices are in ascending order, but not necessarily consecutive, and is terminated by an end-of-vector element which has an index field of all ones.)

Modes which produce two index/value streams include straightforward access from memory of two list vectors, two array vectors, or one list and one array vector. Also supported is *indexed access* using one array and one list vector, with the indices of the non-zeroes in the list vector used to index into the array vector to read the corresponding elements.

Modes which produce one index/value stream are straightforward access to a single list or array vector.

There is also a VR mode which produces as output nothing at all ('null mode'). Null mode is used in instructions with no vector input operand, for example the instruction which fills the output vector operand with repeated copies of a scalar value – in null mode, the Vector Read circuit does nothing.

5.3.4 The Index Match circuit

The Index Match circuit (the first stage of the Arithmetic Section of the pipeline) receives from VR one or two streams of index/value pairs, or nothing at all. It is connected to the next stage of the pipeline, the Arithmetic Unit, by a 152-bit wide data queue which may carry one 24-bit index and two 64-bit values. It passes to the Arithmetic Unit a stream of index/value/value triples, or of index/value pairs, or just of indices.

The functions of IM are:

- To match up pairs with the same index from the two input streams, generating extra zero values, or discarding unmatched input elements, as required.
- To fetch values from the vector register if appropriate.
- To fill out the operand stream so that all indices are explicitly present in the stream if the input vectors were list type, and the instruction requires them to be converted to array type.
- To cause the output data stream to terminate after the index being passed to AU reaches the index.count value specified in the instruction.
- Where appropriate, to flag values passed to AU as *new non-zeroes* to support operation of the Sideways List Unit.

5.3.5 The Arithmetic Unit

The arithmetic unit accepts a stream of index/value/value triples, index/value pairs or indices only, from the Index Match circuit, and may also accept a single scalar input operand supplied as part of the vector instruction. It performs various arithmetic or logical operations on these, and produces as output a stream of index/value pairs, passed to VO via a queue, and/or one or two single scalar output values, passed to the Scalar Result Queue.

The Arithmetic Unit contains separate floating-point add and multiply pipelines, and a bank of 32 64-bit registers, and is able to compute one double precision floating-point add, and one multiply, per clock cycle. This allows operations such as scalar product to use up one pair of input values per clock cycle, with one multiply and one add operation completed each cycle. The functions supported include straightforward vector arithmetic, accumulate operations (*eg* add up elements of vector), a range of searching operations to identify particular elements of a vector, and more complex operations such as scalar product, scaled subtract (for a Gaussian elimination step), and operations to support Linear Programming steps such as *BTRAN* and *FTRAN*.

5.3.6 The Vector Output circuit

The Vector Output stage is the final stage of the Arithmetic Section. It receives as input a stream of index/value pairs from the Arithmetic Unit. It may output a stream of index/value pairs to the Vector Write section of the pipeline (via a queue), and/or write the vector result to the vector register.

Its functions are:

- Optionally to compare the value of incoming I/V pairs against a *drop* value in a register, and to delete the I/V pair if the absolute value is less than the drop value. This operation is valid for floating point values only, and only the exponent field of the value is involved in the comparison.
- To write values into the vector register if required.
- To pass index/value pairs onto the Vector Write unit, if the output vector is to be written to memory.

5.3.7 The Vector Write circuit

The function of the Vector Write circuit is to write the index/value pairs received from the Arithmetic Section back into memory. If the specified output vector is an array vector, this is written to memory in the standard way. If the specified output vector is a list vector, output elements are written into space from the free list, and at the end of the write operation, the descriptor is updated to point at the new version of the vector, while the start address of the old version of the vector is passed to the Garbage Collection circuit. The Vector Write circuit also supports *indexed access* to an array vector; in this case, the index/value pairs coming from AS are written into the array vector using indexed addressing.

The Vector Write circuit is responsible for accessing descriptor memory at the start of the instruction, to fetch the contents of the output vector descriptor, using the descriptor address specified in the instruction. If the output vector is a list

vector, VW also updates the descriptor at the end of the instruction, to ensure that it points at the newly allocated memory space, and to bring up to date the field containing the count of the number of non-zeroes in the vector.

5.3.8 The Garbage Collection circuit

Garbage collection is performed on list vectors which are no longer required, usually as result of an update operation such as $\mathbf{a} \leftarrow \mathbf{a} + \mathbf{b}$, which (in common with all vector instructions producing a list vector result) writes its result into newly allocated space from the free list. If the output vector of a vector instruction is a list vector, the space occupied by the old version of that list vector is retrieved by the Garbage Collector. In addition, vector instructions exist for explicitly collecting an unwanted list vector. (The mechanism used for garbage collection is described in section 4.2.4.)

5.3.9 The Sideways List Unit

The principle of the Sideways List Unit was described in section 4.4. The SLU supports three operations:

- Save the positions of flagged new non-zeroes, output by the Arithmetic Section, in the relevant lists (the new non-zeroes are detected and flagged by the Index Match circuit).
- Save the positions of all non-zeroes output by the Arithmetic Section in the relevant lists.
- Initialise the SLU registers (the *count* and *address* registers are loaded with zero, and the SLU memory address register with 1).

The second and third operations are used when initialising the non-zero position lists for a sparse matrix before elimination starts. If the first or second

operations are specified, the instruction will also contain a word specifying the row number to be loaded into the SLU *row* register at the start of the instruction.

The function of the Sideways List Unit is to keep track of the positions of the non-zeroes in a matrix by *column*, when the rows of the matrix are stored as list vectors in vector memory. Normally, the non-zeroes stored in the row vectors, and the non-zeroes stored in the SLU will correspond exactly. However, if the Vector Output circuit option which drops output vector elements whose values are below a small threshold is enabled, the SLU non-zero lists may still contain an entry for any matrix element thus dropped. This is not a serious problem – in most Gaussian elimination applications, cancellation of non-zero matrix elements to almost zero is rare, and so few elements would be dropped in this way. The extra, false, non-zero positions flagged by the SLU would be too few in number to substantially affect the success of the Markowitz method in minimizing fill-in, and if any such element were selected as pivot by the Markowitz method, it would be discovered to have zero value during the threshold check, and abandoned.

5.4 The interface between the scalar and vector processors

5.4.1 Instruction transfer

Vector instructions are effectively part of the main ESP instruction stream, in the 64-bit wide scalar processor code memory, and are passed to the vector processor by the scalar processor as they are encountered. However, as the entire 64-bit instruction field is used to encode scalar instructions, rather than distinguish vector instructions from scalar instructions by using an extra, 65th, tag bit, ESP uses a different mechanism to include vector instructions in the code. To generate a vector instruction, code is included which causes the scalar processor to write a small number of 32-bit words to a fixed scalar data address, at which is mapped the Vector Instruction Queue (VIQ). The words written are the words which make

up the vector instruction, and will usually be literal values specified in the scalar instruction stream, although some of the words which make up the vector instruction may be calculated at run time by the scalar processor (for example, most scalar input operands to vector instructions would be calculated at run time). Vector instructions entering the 64-word (equivalent to 10 average vector instructions) Vector Instruction Queue await execution by the vector processor. Although this mechanism requires the scalar processor to execute a sequence of instructions to build up a single vector instruction, the overhead is not great, and using the scalar processor to generate the values of scalar operands, rather than requiring the vector processor to decode and execute scalar operand addressing mechanisms, simplifies the design of the vector processor. A similar mechanism was used in the **Burroughs BSP** [8], an array processor in which the control (scalar) processor constructed each array instruction, and entered it into a queue for execution by the processor array. The arrangement was found to support high utilisation of the processor array.

5.4.2 Synchronisation between the scalar and vector processors

Providing a short queue for vector instructions waiting to execute decouples the execution of instructions in the scalar and vector processors, allowing the scalar processor to 'run ahead'; however, a synchronisation mechanism is required to deal with data dependencies between scalar and vector instructions, and data dependencies between different vector instructions.

The synchronisation mechanism chosen for ESP involves the labelling of each vector instruction with a 32-bit integer, the *instruction identifier*, supplied as one word of the multi-word vector instruction. By reading a 32-bit data word from the Instruction Identifier Register, which is mapped to an address in the scalar data memory space, the scalar processor may read the instruction identifier of the most recent vector instruction completed by the vector processor. The programmer or compiler can insert an explicit check and wait loop where necessary; for example, before a issuing a vector instruction which accesses a vector produced by

a previous vector instruction which may not yet be complete, or before scalar code which requires a particular vector operation to be complete. The execution time overhead of this check is unlikely to be any greater than the overhead introduced by an attempt to check for such dependencies automatically. To perform the check automatically would in any case be extremely difficult, when dependencies may be between a (full) vector output operand of an executing vector instruction, and a single element of that vector used as an input operand for a scalar instruction (perhaps accessed by indirection through an address register in the scalar processor). The possibility of such dependencies is presumably the reason for the interlock, in vector processors such as the **CYBER-205**, which prevents any scalar instruction which accesses memory from starting while the current vector instruction is incomplete. The explicit dependency check mechanism in ESP has the advantage that checking and interlock is performed only when required.

5.4.3 Data transfer between the processors

The vector memory, the link memory, the descriptor memory, and the Sideways List Unit memory are all mapped into the scalar processor data memory address space, so that the scalar processor may access any part of each.

The scalar processor may also access a number of registers inside the vector processor – these are also mapped into the scalar processor data memory space. In addition to the Vector Instruction Queue and Instruction Identifier Register described in the previous section, the scalar processor can read scalar results of vector instructions from the Scalar Result Queue, can access the *count* and *address* registers in the Sideways List Unit, and can read and initialise various registers in other parts of the vector pipeline, for example, the *drop value* register in VO, the *free list pointer* in VW, *etc.*

5.4.4 Vector exceptions

Various types of exception can arise during execution of a vector instruction. For example, an arithmetic overflow may occur. After a vector instruction encounters

an exception, processing of the instruction continues, so as to clear all the data for that instruction from the pipeline. Processing of subsequent vector instructions continues normally. Any vector instruction exception causes a bit to be set in a *vector exception flag register*, which may be read by the scalar processor. Bits in this register are 'sticky' – they remain set even though subsequent vector operations complete successfully – and they must be explicitly reset by the scalar processor writing to the register. Associated with the vector exception flag register is a *vector trap enable register* – when an exception flag is set, if the corresponding bit in the vector trap enable register is set, the scalar processor is interrupted. The vector trap enable register is loaded by the scalar processor. The scalar processor interrupt handler will normally be able to identify which vector instruction caused the exception, although in some cases, when several vector instructions are queued for execution, exact identification of the problem instruction may be difficult.

5.5 The control processor

The control processor provides the interface between ESP and the host workstation. It contains a small general purpose microprocessor system able to interpret commands passed from the host, and to transfer data at high bandwidth between the host and the various memory spaces of ESP. It is also able to start, stop, interrupt, and single step the ESP hardware, and can also interrupt the host processor.

In addition, the control processor supports two standard RS-232 terminal ports, and runs a simple monitor program allowing a user at a terminal to download ESP programs and data (via the other terminal line), and to run a number of hardware tests, including single stepping the machine through a program, independently of the host workstation.

5.6 The instruction set

The scalar instruction set of the machine is fully determined by the choice of the **Weitek XL-8364** as scalar processor [48,49,50].

The vector instruction set is defined in detail in Appendix A. The decomposition of the vector pipeline into three asynchronously controlled sections is reflected in the instruction format – each pipeline section (Vector Read, the Arithmetic Section, and Vector Write) is controlled by a different field of the instruction. In addition, the Arithmetic Section instruction field consists of three subfields, controlling the Index Match circuit, the Arithmetic Unit, and the Vector Output circuit, while the Vector Write instruction field includes subfields to control the Garbage Collection circuit and Sideways List Unit. These instruction fields are as far as possible orthogonal, except for the constraint that the kind of operand stream generated by the instruction field for each part of the pipeline must match the required input stream for the instruction field for the next pipeline part. This orthogonality means that the majority of vector instructions will operate quite happily on list vectors or on array vectors, or on one of each, and that one array operand can optionally come from the vector register rather than memory. Output vectors can be lists or arrays (the latter written to memory or to the vector register), independent of the kind of input vectors.

The instruction fields controlling the Vector Read and Vector Write sections, and the Index Match and Vector Output parts of the Arithmetic Section, select from the different data handling and format conversion operations performed by those parts of the pipeline, which are detailed earlier in this chapter. The heart of the vector pipeline is the Arithmetic Unit, and this part of the pipeline has the largest instruction field set. The Arithmetic Unit part of the instruction set is based on the instruction set for MU6V [26,47]. The set may be subdivided into four subsets:

Data movement operations These include the Arithmetic Unit *no-op*, used

when copying a vector, *fill* instructions, used to fill the output (array type) vector with copies of a supplied scalar or with an incrementing integer, *select* instructions, which locate or extract vector elements whose values satisfy a specified arithmetic comparison against a supplied scalar, and *search* instructions, which find extreme values in a vector.

Arithmetic operations These include the normal vector/vector operations: element-wise add, subtract, multiply and divide. Reverse subtract and reverse divide are also provided, because the two-operand fetch options of the Vector Read circuit are not fully symmetrical. There is a variety of options for scaling one of the two vector operands using an additional scalar operand, including, for example, the scaled vector subtract operation required for Gaussian elimination. Instructions for add, subtract, multiply and divide of a single vector by a scalar operand are also provided. Finally, *accumulating* arithmetic instructions, which produce a scalar, rather than a vector result, are provided. These include the scalar product instruction, and instructions for adding up the elements of a vector, and for accumulating the sum of the squares of the elements of a vector.

Logical operations Bit-wise *and*, *or*, and *exclusive or* operations between two logical type vectors are supported, plus the logical inversion of the bits of a single vector operand. The *shift* instructions shift each word of a single vector operand, and there are accumulating instructions which produce a scalar result from a bit-wise *and*, *or* or *exclusive or* of every word in a vector.

Special operations These are unusual vector instructions which have been added to support specific operations in the target applications – operations which are not normally implemented by a single instruction, but which are time critical. Two such instructions have been identified so far, to support the Linear Programming *BTRAN* and *FTRAN* operations. The *BTRAN* instruction takes three operands: an array vector held in the vector register (this is the vector being updated), a sparse vector from memory (the η-vector), and an integer (the column position of the η-vector). The instruc-

tion forms the scalar product of the two vector operands, then writes the result to the element of the vector register indexed by the integer operand. The *FTRAN* instruction takes the same triple of operands. It first reads the value of the element of the vector register indexed by the integer operand, then scales the sparse vector operand by this value and adds the scaled sparse vector to the vector register.

The instruction decode and control circuits for each part of the vector pipeline have been implemented using programmable logic components, to allow some scope for alterations to the instruction set in the light of experience with the machine. In particular, it may turn out that there are other parts of the target applications which can be made considerably faster by adding new special vector instructions like those already provided for *BTRAN* and *FTRAN*.

Chapter 6

The Implementation of ESP

6.1 Introduction

The implementation of a prototype of the architecture described above has been underway since 1991. The implementation has been developed by the author and R.W. Thonnes; detailed circuit design has been carried out by R.W. Thonnes, and construction, by P.J. Lindsay. The detailed design and construction are not yet complete.

6.2 Basic design decisions

The timescale and budget available for the building of the prototype have precluded the use of custom VLSI; ESP has therefore been designed from off-the-shelf components. However, the uncertainties in the architectural details of the vector processor, and the relative lack of experimental results from simulation studies at the start of the design, meant that it would be desirable to be able to make changes to the detailed architecture in the light of experience with the machine. This has affected both the choice of components and the constructional technique. By using general purpose parts for implementing data path registers and processing elements, rather than single-function components, and by extensive use of

programmable hardware in control circuits, the detailed function of each part of the machine may to some extent be modified in the light of experience in commissioning it. In particular the **Integrated Device Technology IDT49C402** 16-bit wide general-purpose data path device, which contains an ALU and 32 16-bit registers, has been used in several of the functional units, and the **Altera** range of erasable programmable logic devices (EPLDs) has been used in many parts of the design, including in particular the **EP448** programmable microcode sequencer/store [2], for controlling operation of the data path devices. The choice of wire-wrap construction means that it is also possible to make more substantial hardware changes if necessary.

Early design decisions were made for some parts of the machine where the choice of suitable components was limited. In particular, it was straightforward to choose a component to form the Arithmetic Unit of the vector pipeline. The relative advantages of the VLSI parts available for IEEE standard floating-point arithmetic are discussed in section 6.10.2 below – the part chosen was the **Weitek WTL-3364** [48]. Use of this component limits the clock speed of the vector pipeline to a maximum of 10MHz.

Another early decision was to choose the 3-chip set **Weitek XL-8364** [48,49, 50] as the scalar processor, mainly for reasons of performance, but partly because it uses the same floating point unit as that chosen for the vector processor. This has helped to reduce time spent on both hardware and software development. The fastest version of this chip set available is an 80ns version, from which it is possible to build a system running at 10MHz, using a memory system capable of 100ns cycle time and 80ns access time. In practice, 120ns is more likely to be the fastest clock speed achievable, once external logic has been added.

The third design decision taken at an early stage was to implement the vector memory using four banks of dynamic RAM. Static RAM would have been prohibitively expensive for the size of store required. The fastest dynamic RAM components available at that time had cycle times in the range 120ns to 150ns, which roughly matches the clock speed likely to be achievable in the vector pipeline. Each pipeline cycle may use two vector input operands and produce one vector

output operand, so it is necessary to read two words from, and write one word to, memory every cycle. Allowing extra memory bandwidth for scalar processor accesses, it is clear that providing four memory banks matches the memory bandwidth to that of the processors. The highest density memory components available at reasonable cost were 1M by 8-bit modules, and so each bank is constructed from eleven of these, to give a total of 4M words of 88-bit wide vector memory.

In the light of these decisions, the design of the rest of the machine assumed a main clock period of 120 to 150ns, with the possibility that some parts which might be built from faster components (*eg* the Vector Read and Index Match circuits) might be clocked from a synchronous clock running at twice that speed.

6.3 Physical partitioning of the machine

Initial estimates indicated that the required circuit board area was approximately 4000 cm^2. The largest easily-available standard card frame, *triple Eurocard (9U)*, provides for boards up to approximately 30 cm square, and wire-wrap prototyping cards of this size, with a complete 0.1" grid of holes, and power and ground colander planes, were available within the Department. Each of the available prototyping boards can be fitted with three 96-way connectors on the back edge, plus three 64-way connectors on the front edge. Allowing 36 connector pins for power feed to the board via the back edge connectors, 444 pins remain available, and a proportion of these is required for ground return wires for critical signals. The circuitry was partitioned between seven such boards - the main criterion for deciding the partition being minimisation of inter-board connections.

The parts of the machine are divided between the seven circuit boards as follows:

1 The control processor and ESP end of the host interface connection

2 The scalar processor with associated code memory and scalar data memory

3 The vector Arithmetic Section (comprising the Index Match circuit, the Arithmetic Unit, and the Vector Output circuit) plus the Vector Instruction Queue, and the Overall Pipeline Control circuit

4 The memory controller, comprising Vector Read and Vector Write circuits, Garbage Collector, plus link memory and vector descriptor memory

5 The Sideways List Unit

6, 7 A pair of boards carrying the vector memory, two banks per board.

The relative positions of, and connections between, the seven boards are illustrated in figure 6–1.

The seven boards are mounted, in the order shown, in a standard triple-Eurocard card frame inside a 19" cabinet, in the lower half of which is the machine's power supply (which can provide 60A at 5V). Connection between the boards is effected via three printed circuit backplanes running across the back of the card frame, designated **A** (top), **B** (middle), and **C** (bottom), plus ribbon cable connections between connectors mounted on the front edges of some of the boards – there are three positions for these front edge connectors, designated **D** (bottom), **E** (middle), and **F** (top).

The backplane printed circuits incorporate ground planes, and provide an electrically quieter connection than the ribbon cable connections at the front of the boards. Each backplane however runs the full width of the card frame, whereas each front edge connector position can carry different signals between different pairs of boards. The backplanes were therefore used for the common connections to all boards, and for more time critical connections. Running across most of backplane **A** is the *scalar bus*, which provides for transfer of all data and code to and from the host, via the control processor, for transfer of data between the scalar processor and vector processor registers, vector memory, and SLU, link and descriptor memories, and for transfer of vector instructions from the scalar processor to the vector processor. Running across the remainder of backplane **A**,

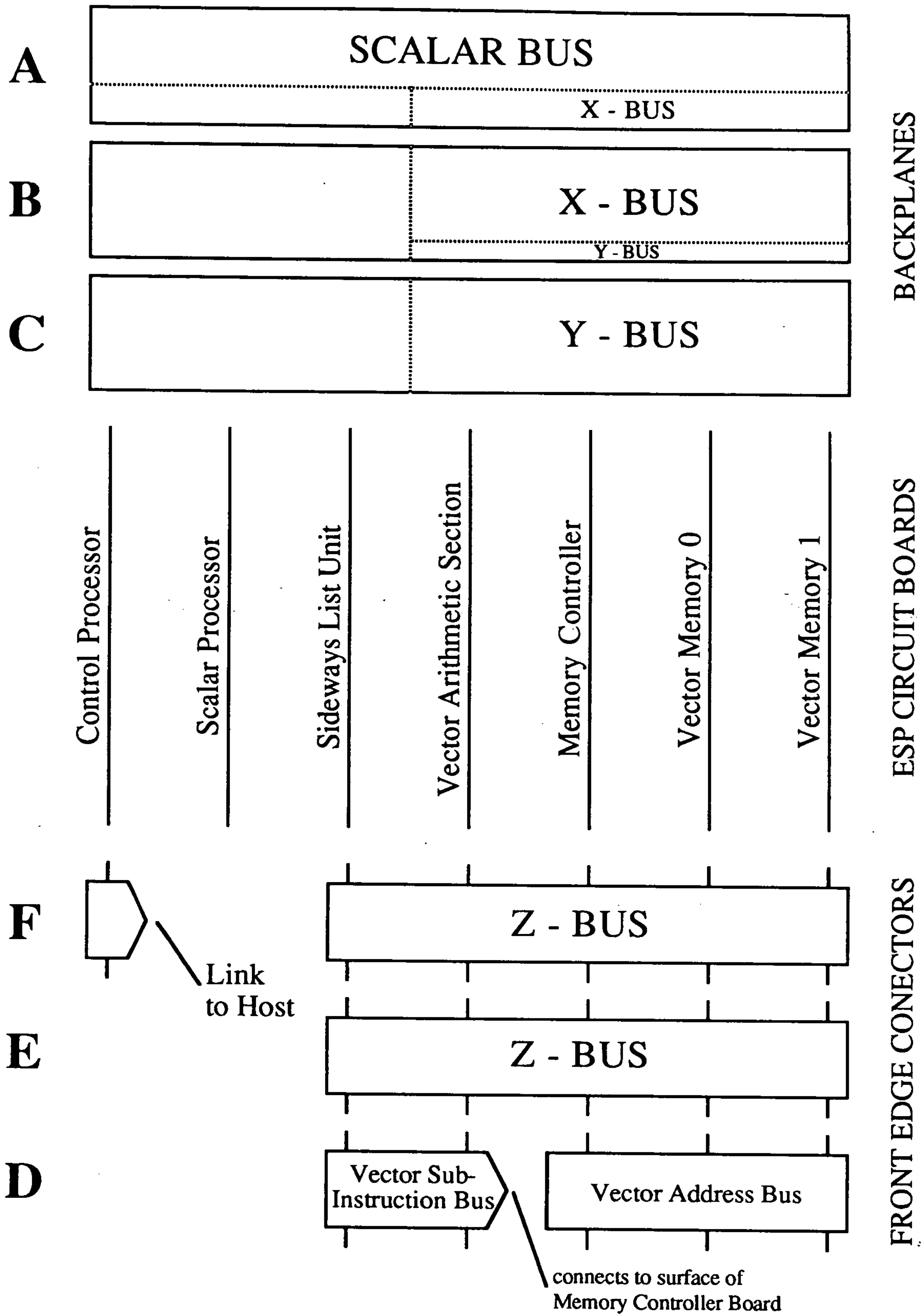

Figure 6–1: Interconnection between the seven circuit boards

together with the whole of backplanes **B** and **C**, are the two vector Arithmetic Section input buses, designated the **X** bus and the **Y** bus. These carry index/value pairs from the vector memory, under control of the Vector Read section, to the vector Arithmetic Section.

Connecting the memory controller to the vector memory boards, in the **D** connector position at the front of the boards, is the *vector address bus*, which carries vector addresses and read/write request control signals from the memory controller to the memory boards. The **E** and **F** connector positions are used for the vector Arithmetic Section output bus (the **Z** bus), which carries index/value pairs from the vector Arithmetic Section to the memory and to the Sideways List Unit, under control of the Vector Write section on the Memory Controller board. In the **D** position, a ribbon cable link runs from the Arithmetic Section board to the Sideways List Unit board, and to the Memory Controller board, via a connector on the board surface, carrying vector subinstructions and synchronisation signals.

Running from the control processor board to the host interface board in the host workstation is a 4m long ribbon cable, for data and program transfer, and control purposes.

6.4 The host interface board

A high-bandwidth interface is required from the host workstation, via the ESP control processor, to ESP's memories. The current host is a SUN **3/60** workstation with VME bus slots for high-bandwidth I/O, and to provide the necessary functionality at the host end, an interface board based around a **68010** microprocessor (chosen because of the level of local support for and experience with **68010** design) has been designed for one of the VME slots. A block diagram is shown in figure 6–2. 64Kbytes of EPROM and 64Kbytes of RAM are provided, plus two RS232 ports, and interfaces to the SUN VME bus and to the cable connecting to ESP. The board is clocked at 8MHz, and constructed on a 6U high wire-wrap prototyping board, so as to match the standard VME format.

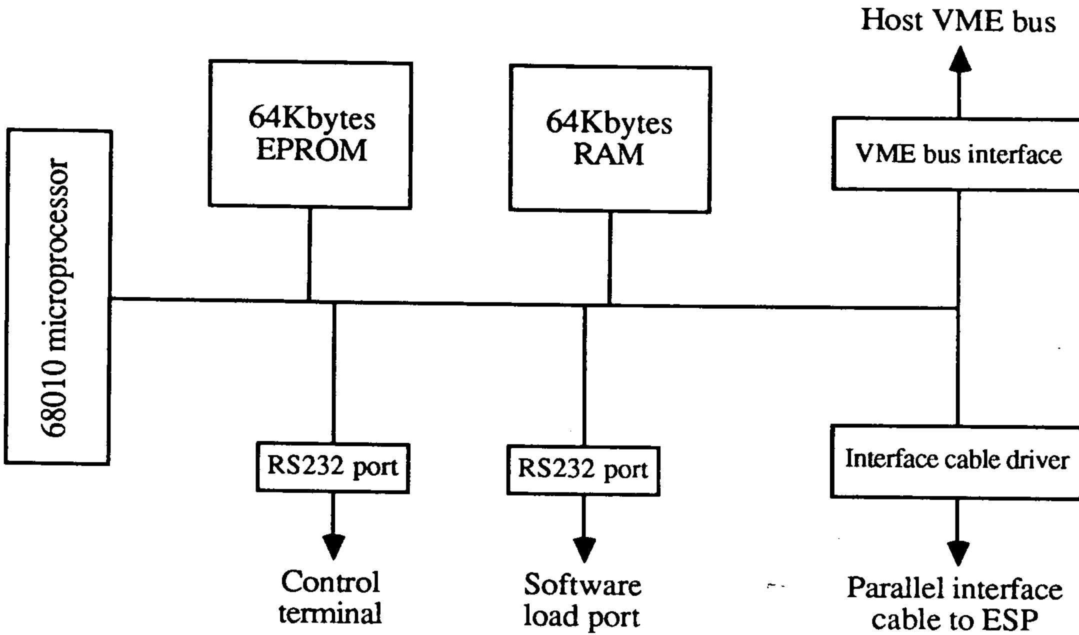

Figure 6–2: The host interface board

The interface board processor is able to access an area of the host workstation's physical memory, and thus blocks of data can be transferred from a buffer in host memory, across the cable connecting the interface board to the control processor board in ESP. This cable provides a 16-bit parallel data path, and can transfer one 32-bit word every 2.75μs. A parallel path was chosen so that the speed of transfer over the cable would be able to keep up with the rate at which the interface board could access data in the host memory.

A simple loader program is held in EPROM. During development, the loader is controlled via one of the RS232 ports, and the main software for the board is loaded into RAM via the other RS232 port. Software may be started by a simple command from the control RS232 port. In the final configuration, all the software for this board will be loaded automatically from the workstation.

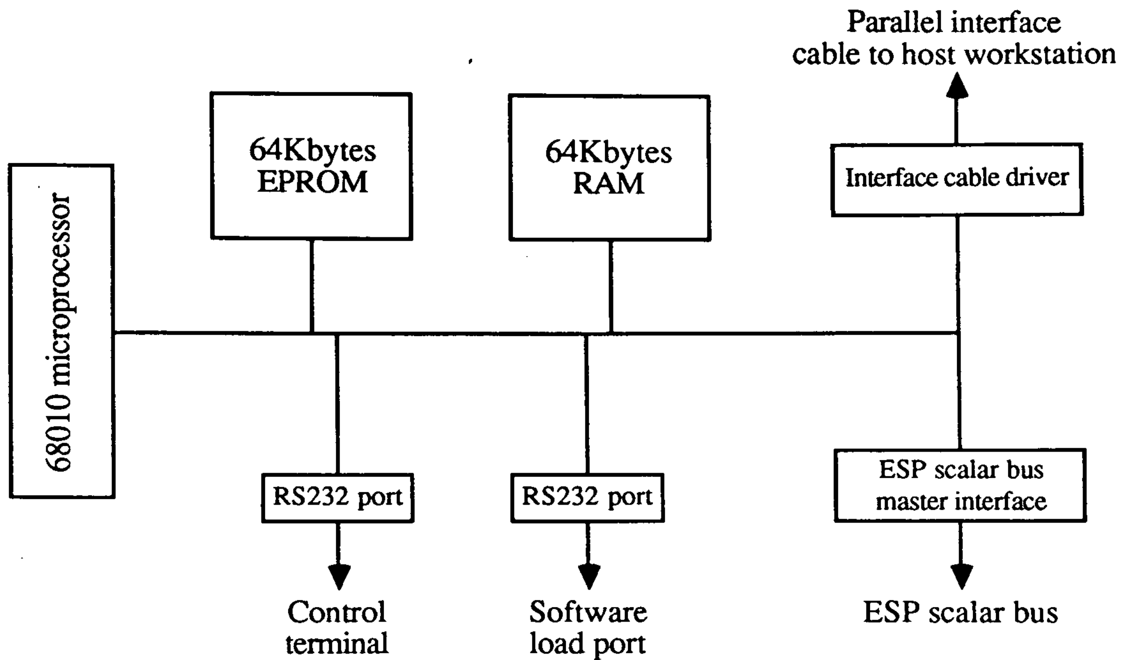

Figure 6–3: The control processor

6.5 The control processor

The functions of the control processor are: to support the transfer of programs and data to/from the host workstation, to support control by the host of ESP program execution, to support interrupts to the host by the ESP program, and to allow local testing of ESP, separately from the host.

The obvious way to provide this level of functionality is to use a small microprocessor system, and again the processor chosen was a **68010** running at 8MHz, provided with 64Kbytes of EPROM and 64Kbytes of RAM, plus two RS232 ports. Again, a simple loader in EPROM allows loading of software into RAM from an RS232 port.

A block diagram of the complete control processor is shown in figure 6–3. The interface to the ESP scalar bus is described in the following section. The main ESP clock is generated on the control processor board, and runs at 8MHz.

6.5.1 The scalar bus

The control processor must be able to access all the memories of the machine: the program memory, the scalar data memory, the vector memory, the link memory, the descriptor memory and the Sideways List Unit (SLU) memory. All these except the SLU memory (which can be initialised by the vector processor) need to be initialised from the control processor, before an ESP program can run. Also, data may need to be transferred back to the host from several of these memories at the end of a program run. In addition, it is important that the control processor be able to access all ESP's memories, including the SLU memory, and many control and status registers, for debugging purposes.

In order to be able to create and read vector and matrix data, ESP's *scalar* processor also must have access to the vector, link, descriptor and SLU memories, and must also be able to access a number of registers and queues within the vector processor (*eg* the Vector Instruction Queue, the Instruction Identifier Register, *etc*). Since control processor access to memory will normally be before and after program runs, while scalar processor access will be during program execution, it was decided to use a single bus, called the *scalar bus*, to support access by both processors.

The scalar bus runs across the entire width of the backplane, so that it can connect to memories and registers on any of the seven boards. By default, the scalar processor is bus master, while the control processor may request bus mastership. Bus arbitration takes place on the scalar processor board, and control processor requests take priority over scalar processor accesses. Into the scalar bus address space are mapped all the memories of the machine, plus a number of control and data registers within both the vector and scalar processors. In addition, a register on the control processor board is mapped into this address space, so that programs running on the scalar processor can return status to the control processor.

6.6 The host interface software

The software supporting the host interface consists of three parts: software running on the host workstation itself, software running on the host interface board processor, and software running on the ESP control processor. The simple loaders for the control processor and the host interface board processor are written in **68010** assembler, while the rest of the software is written in C.

In the initial stages of development, all data transfers are initiated by the software running on the host. Control operations, such as starting and stopping ESP's scalar processor, are implemented with single word data transfers to control registers mapped into ESP's scalar bus address space. Thus, the only operation which needs to be supported by the interface software is the transfer of a specified size block of data between the host program and a specified area in the ESP scalar bus address space. 'Interrupts' from ESP to the host are implemented by having the host program regularly poll the ESP status register – at a later stage of development, a proper ESP to host interrupt mechanism will be added.

6.7 The scalar processor

The scalar processor is built around the **Weitek XL-8364** microprocessor, which comprises a 3-chip set. The 3 devices are: the *Program Sequencer Unit (PSU)*, the *Integer Processing Unit (IPU)* and the *Floating Point Unit (FPU)*. Each unit is normally controlled from a different field in the instruction word (although there is some overlap between the PSU and IPU control fields), and the whole instruction is 64 bits wide. 32K words of 64-bit wide instruction memory are therefore provided – this is built from 80ns static RAM, so that one instruction may be fetched per clock cycle. 32K words of 32-bit wide scalar data RAM are provided (also built with 80ns static RAM).

A block diagram of the scalar processor is given in Figure 6–4.

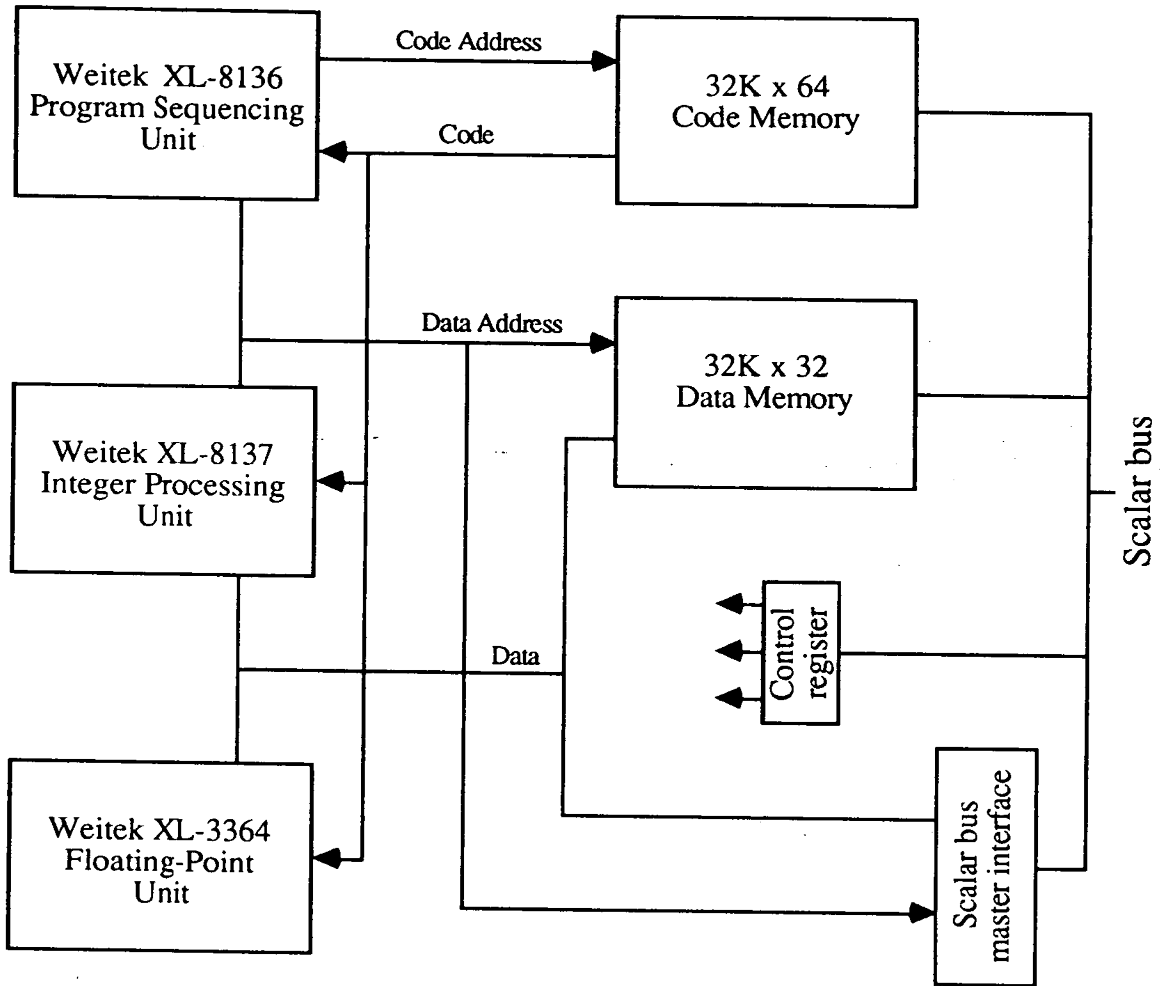

Figure 6–4: The scalar processor

Vector instructions are not explicitly present in the instruction stream – they are generated by scalar processor code which writes a vector instruction as a short sequence of 32-bit words, into the Vector Instruction Queue, via the scalar bus (see section 5.4.1). No special hardware is needed on the scalar processor board to support this.

6.8 The memory controller

The memory controller comprises the Vector Read, Vector Write, and Garbage Collection circuitry, plus the two memories used by these circuits, the link memory and the descriptor memory (see Figure 6–5). Accesses to the link memory may come from the Vector Read circuit (if one or two list vectors are being read), the Vector Write circuit (if a list vector is being written), and the Garbage Collection circuit. The descriptor memory is accessed by the Vector Read circuit at the start of a vector operation, and by the Vector Write circuit, at the start of an operation which produces an array vector result, or at the end of an operation which produces a list vector result (to update the start address and non-zero count). Arbitration circuitry is therefore required for both these memories.

Arbitration is also needed for access requests to the vector memory itself – the Vector Read and Vector Write circuits will often both be accessing vector memory during a vector instruction. In addition, requests to access vector memory may be made via the scalar bus, by the scalar processor or the control processor. To simplify the arbitration, all these requests are channelled through the memory controller board. A vector address bus connects the memory controller to the two vector memory boards, and transfers one memory request per clock cycle. Eight distinct kinds of memory request may be made:

R4X, R4Y Read four consecutive memory locations starting at the given address, into the memory board output queues for the X-bus and Y-bus respectively.

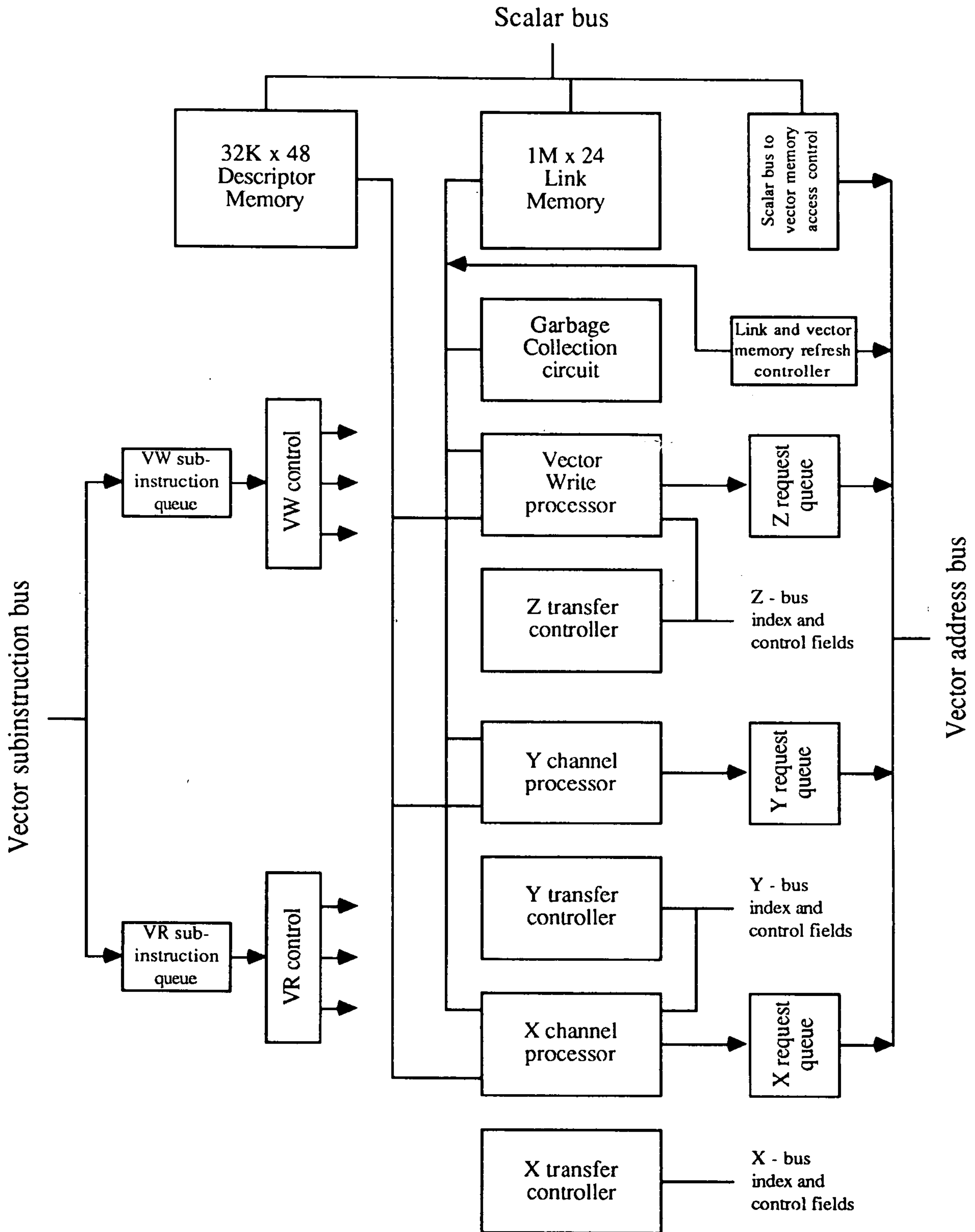

Figure 6–5: The memory controller

R1X Read a single memory location into the memory board output queue for the X-bus.

W4Z Write four consecutive memory locations starting at the given address, with data from the memory board Z-bus input queue.

W1Z Write one location with a word of data from the memory board Z-bus input queue.

RS Read a memory location into the memory board scalar register.

WS Write a memory location with data from the memory board scalar register.

Refresh Perform a refresh cycle.

6.8.1 The Vector Read circuitry

The Vector Read circuitry forms the first section of the vector processor pipeline. It receives instructions from the vector processor Arithmetic Section board via the vector instruction bus – a new instruction is sent by the vector processor as soon as there is space in the Vector Read instruction queue, and there is a vector instruction available. The instruction received by the Vector Read circuit comprises a sequence of one or more 32-bit words – the first word is a copy of the vector instruction control word. This may be followed by the vector descriptor address for one or two vector input operands.

The Vector Read circuitry is implemented as two microprogrammed 32-bit integer processors, each constructed from an **EP448** programmable microcode sequencer/store, and a pair of **IDT49C402** 16-bit integer processors. One of these microprogrammed processors, the *X-channel processor*, is responsible for issuing memory requests for the X-bus data stream, the other, the *Y-channel processor*, for the Y-bus data stream.

The X-channel processor has three modes of operation. One corresponds to the Vector Read modes designated **AA2**, **AL2** and **A1** (see Appendix section A.1) –

here the X-channel is reading an array vector. First the start address is fetched from the descriptor memory, and then a sequence of **R4X** requests is issued into the X-request queue, until the entire vector length (given in the non-zero.count field of the descriptor, but incremented to allow for the end-of-vector element, and then rounded up to a multiple of four) has been requested. The necessary address increment and length checking is performed in the 32-bit integer processor. As the issuing of the memory requests will normally proceed faster than they can be processed, the X-request queue will fill, and an interlock holds up the X-channel processor while the queue is full.

The second X-channel mode corresponds to VR modes **LL2** and **L1** - in this case, a list vector is being read on the X-bus. Again, the start address is fetched from the vector memory, and **R4X** requests are issued, but the link memory must be examined to generate the correct vector addresses. The operation terminates when the end of the vector is reached – this is determined from the non-zero.count in the descriptor, as before.

The third X-channel mode supports indexed access into an array vector (VR mode **AiL2**). This is a little more complex. The vector start address is fetched from descriptor memory as before, but, instead of repeatedly accessing blocks of four consecutive vector elements, only those elements whose indices are present in the list vector stream being read on the Y-bus should be fetched. In this mode, therefore, the X-channel processor watches the Y-bus, and copies the index part of each word as it is transferred from the vector memory output queue to the vector arithmetic section input queue. This index is added to the X-channel vector start address, and an **R1Y** request issued to the X-request queue. This continues until the end-of-vector marker index is detected on the Y-bus, indicating the end of the list vector. A final R1X request is then issued, to address zero, which always contains an end-of-vector marker, thus ensuring that there is an end-of-vector marker on the end of the X stream. Note that the transfer of the X data words will be delayed with respect to the transfer of the Y data words, but the words will be matched up again when they are read from the Arithmetic Section input queue by the Index Match circuit.

The Y-channel processor supports the first two modes described above – for fetching an array vector to the Y-bus (VR mode **AA2**) and for fetching a list vector (modes **LL2** and **AL2**).

6.8.2 The Vector Write circuitry

The Vector Write circuitry (together with the Garbage Collector and Sideways List Unit) forms the final section of the vector pipeline. It receives its instructions from the vector processor Arithmetic Section board, over the vector instruction bus – the next instruction, if there is one, is sent as soon as the Vector Write section instruction queue has space for it. The instruction comprises a copy of the vector instruction control word, followed by the descriptor address of the output vector (if there is one), and the SLU **row.number** (see section 5.3.1), if there is one.

The Vector Write circuit itself is implemented in a similar way to the each of the Vector Read channels, with a 32-bit integer processor and a microprogrammed controller. It has three modes: for writing an array vector result (VW mode **A**), writing a list vector result (VW mode **L**), and for writing elements into an array vector using indexed addressing (VW mode **Ai**).

In mode **A**, instruction execution starts with the fetching of the vector start address from descriptor memory into one of the VW processor registers. **W4Z** requests are then issued into the Z-request queue, but, unlike vector read requests, these are not issued as fast as the Vector Write circuit can generate them, but are only issued as required. This is achieved by the vector write circuit monitoring the Z-bus, and issuing a new **W4Z** request every time four elements have been passed from the arithmetic section to the memory board Z input queue. The final **W4Z** request is signalled by the presence of the end-of-vector marker in any of the most recent block of four elements (the vector output stream coming from the Arithmetic Section will always be padded out to a multiple of four elements, and the end-of-vector marker will always be in the element immediately after the last real vector element, so may be in any of the four last elements).

Mode **L** is similar, except that the addresses in the **W4Z** requests are generated not by counting, but from the free list, by examining the link memory starting at the address given by the free list pointer register. In this case, at the end of the operation the start address field in the vector descriptor, and the free list register, are updated accordingly. In addition, the vector write circuit keeps a count of the number of true vector elements which have passed on the Z-bus (*ie* not counting the end-of-vector marker or any pad elements), and this is written to the non-zero count field in the vector descriptor. At the end of the operation, the old vector start address is passed to the Garbage Collection circuit, so that the space previously occupied by the vector can be reclaimed.

In mode **Ai** output elements are written into an array vector using indexed addressing. At the start of the vector write operation, the vector start address is fetched from descriptor memory. The index field of each element passing on the Z bus is added to the vector start address, and a **W1Z** request issued to the Z request queue. This continues until the end-of-vector marker is detected on the Z-bus. A final **W1Z** request is then issued for vector address zero, causing the end-of-vector marker to be written to this address (which is used solely for disposing of these unwanted values).

6.8.3 The Garbage Collection circuitry

The Garbage Collection circuit operates under the control of the Vector Write circuit. If the Vector Write instruction specifies mode **G**, and also, at the end of each Vector Write instruction using mode **L**, the Garbage Collection circuit is passed the (original) start address of the specified output vector, so the space previously used by the vector can be merged with the free list. This circuit is implemented in a similar way to the Vector Read and Write circuits – with a 32-bit integer processor and a microprogrammed control section. It implements the garbage collection algorithm described in section 4.2.4.

6.8.4 Vector memory request arbitration

At any time, there may be outstanding vector memory read requests in the X and Y request queues, and outstanding write requests in the Z request queue. In addition, the memory controller must ensure that the dynamic vector memory is refreshed as often as required, and the scalar processor may also attempt a read or write to or from vector memory at any time. As vector memory is 88 bits wide, while the scalar bus is only 32 bits wide, a special mechanism is used to access vector elements via the scalar bus. (The same mechanism is used for vector memory access via the scalar bus, by the *control* processor.)

To write an element to vector address n, the scalar processor first writes the more significant half of the vector value field to address **msvalue**, a fixed address in the scalar processor address space, which identifies a 32-bit register on the vector memory boards. It then writes the less significant half of the value to **lsvalue**, a second 32-bit memory board register. Finally, the scalar processor writes the vector index field (zero-extended from 24 to 32 bits) to address (**vectorbase** + n). The index data is loaded directly into a register on the memory boards, but the requested address is captured by the memory controller. The memory controller then issues a **WS** (write scalar) request to the memory boards, one of which performs the update of vector memory location n from the three data registers.

To read a vector element from vector address n, the scalar processor reads from (**vectorbase** + n). The memory controller recognises the vector memory read request on the scalar bus, and issues an **RS** (read scalar) request to the vector memory. One of the memory boards performs the reading of the 88-bit vector element into three 32-bit data registers on the memory board. The register containing the index (zero-extended from 24 to 32 bits) is enabled onto the scalar bus in the following clock cycle, and the scalar bus read request is acknowledged, completing the first read cycle. The scalar processor then reads the two halves of the value field from locations **msvalue** and **lsvalue** – these read requests are handled by the memory boards; the memory controller is not involved.

In order to avoid holding up the scalar processor, and because scalar accesses

to vector memory are expected to be relatively infrequent, scalar bus access to vector memory is given priority over X, Y and Z requests. Outstanding X, Y and Z requests are passed to the vector memory in a round-robin fashion (except when the X queue full or Y queue full signal from the memory is asserted – see below). A memory refresh timer and row counter request refresh cycles as required; these have the highest priority. The memory arbitration controller is implemented using EPLDs.

6.8.5 The data transfer controllers

The memory controller board carries three circuits which control data transfer between vector memory and the arithmetic section of the vector pipeline, across the X, Y and Z buses. Words are transferred from the memory boards to the Index Match circuit (IM) X and Y input queues via the X and Y buses, and are written from the Arithmetic Section Vector Output circuit to the memory boards (and to the Sideways List Unit) via the Z bus.

All transfers are under the control of the memory controller board data transfer controllers. Using a pair of X bus transfer request signals to the memory board, and an X load signal to the IM input queue, the X data transfer controller may request five types of transfer on the X bus:

X0, X1 These are used when **R4X** memory requests are being used. One word is enabled onto the X bus from the X queue on memory board 0 or 1 respectively, and is clocked into the IM X input queue.

X0K, X1K As above, but the word is killed, *ie* it is enabled onto the X bus from the X data queue, but is *not* clocked into the IM input queue. This is for removing elements after the end-of-vector marker, and relieves the Index Match circuit of the this task. The X transfer controller monitors the index field of the X bus to identify the end-of-vector marker, and knows how many kill requests to issue from the value in a two-bit transfer counter.

XB This is used when **R1X** memory requests are in use. One memory board's X queue will have an actual memory word at the front, the other board's X queue will have a dummy value at the front (see section 6.9). The board without the dummy value places the front value from the X queue onto the X bus; the other board does not enable its X bus drivers, but discards the dummy value. The bus value is loaded into the IM X input queue.

Apart from up to three 'kill' transfers when the read mode is **R4X**, transfers cease after the transfer controller recognises the end-of-vector marker in the word being transferred.

The Y transfer controller supports the first four types of transfer only, as the only Y read mode is **R4Y**.

To achieve proper synchronisation, each of the X and Y transfer controllers monitors a bus signal which is generated by the second memory board (*ie* board 1), which indicates that the X (or Y) data queue on the memory board is empty. A similar signal from the each of IM X and Y input queues indicates that the queue is full.

The Z bus works in a similar way. A signal from the Vector Output circuit on the Arithmetic Section board indicates that a data item has been enabled onto the Z bus, for transfer to the Z data queue for vector memory. The Z transfer controller issues one of three possible transfer requests, using two signal lines to the memory boards, and an 'next item' signal to the Vector Output circuit. **Z0** and **Z1** requests cause the data on the Z bus to be copied into the Z queue on memory board 0 or 1 respectively, and signal the Vector Output circuit to proceed with the next data item (these requests are for use with **W4Z** mode writes). **Z0D** and **Z1D** ("D" for dummy) requests are similar, but do not signal next item to the Vector Output circuit. These are used to pad out the final block of four data words with dummy values, after the end-of-vector word. The Z transfer controller recognises the end-of-vector index pattern on the Z bus, and then issues up to three of these dummy requests, according to value of a two-bit count of words transferred so far. The actual data loaded into the Z queue is immaterial - it will

be whatever happens to be on the Z bus in those cycles – either the bus float value or, possibly, the first data output item of the next vector instruction. Finally, **ZB** requests cause the word on the Z-bus to be copied into *both* Z queues, and the next item signal to the Vector Output circuit to be asserted – this is for use with **W1Z** mode writes.

Z0, **Z1** and **ZB** requests also cause the index field on the Z-bus to be loaded into the Sideways List Unit input queue, if an additional control signal on the Z-bus, driven by the Vector Output circuit, indicates that the index of this vector element is to be appended to an SLU list (*ie* it is a new non-zero in the matrix). A wired-or status signal, driven by memory board 1 and by the SLU board, indicates to the Z transfer controller that the memory board Z data queues or the SLU input queue are full; this holds up the issue of transfer requests.

6.9 The vector memory

The two vector memory boards are virtually identical. Each board carries two banks of vector memory, and each bank comprises 1M words of 88-bit wide memory (24 bits to store the index, 64 bits for the value). A block diagram of the vector memory hardware on one board is given in figure 6–6. Memory banks 0 and 2 are on board 0; banks 1 and 3 are on board 1.

At the start of each clock cycle, the memory boards may latch a memory request from the memory controller into the memory request register. The two memory boards always receive exactly the same request in each cycle. The action of the memory board in the clock cycle depends in the kind of memory request (refer to section 6.8 above), as follows:

R4X, R4Y Each memory board reads a word from each of its two memory banks, into holding registers. In the following clock cycle, each board loads the two words read, in order, into the X queue or Y queue.

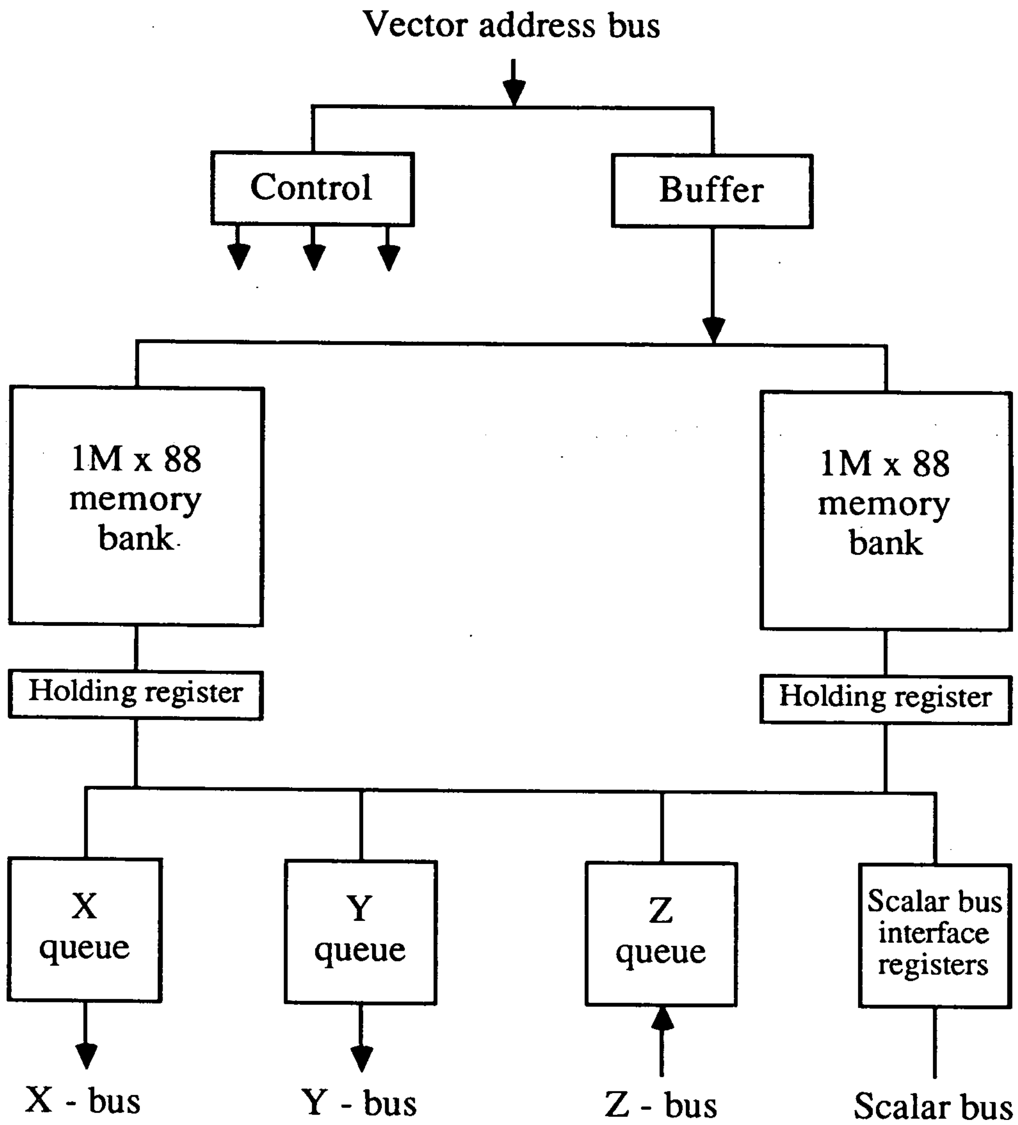

Figure 6–6: The vector memory board

R1X The memory board with the selected bank (as indicated by the bottom two bits of the memory address) reads a word from that bank into a holding register. The unselected memory board places a special dummy word into the holding register. In the following cycle, both boards load the single word from the holding register into the X queue.

W4Z Each memory board writes the front two words in the Z queue into its two memory banks.

W1Z The memory board with the selected bank writes one word from the Z queue into that bank. The other board deletes one item from the Z queue.

RS The memory board with the selected bank reads one word into the scalar output registers, and sets the scalar access flag flip-flop. The unselected board unsets its scalar access flag flip-flop. In the following cycle, the selected board's index field scalar register is enabled onto the scalar bus, and a scalar bus acknowledge signal generated.

WS The memory board with the selected bank writes the scalar input registers into the memory.

Refresh Both memory boards perform a single refresh cycle. No data transfers take place.

X-queue-full and Y-queue-full status signals are provided by memory board 1, to the memory controller (note that it is impossible for a queue to become full on memory board 0 without the corresponding queue on memory board 1 also being full, and note also that "full" here means zero or one places free, as two free places are needed in each queue for a memory request which reads four words to be safely issued). When asserted, these signals prevent the issue of X memory requests and Y memory requests, respectively. No status signal is needed to signal that the Z queue is empty, because the memory controller only issues Z memory requests when it has monitored the necessary number of data items being transferred across the Z bus into the Z queues.

The memory boards are also able to respond directly to scalar bus accesses to the two addresses **lsvalue** and **msvalue**, which correspond to the less significant and more significant half of the value field of the scalar input and output registers. A scalar bus read access from either of these addresses will cause the memory board with the scalar access flip-flop set to enable the relevant register onto the scalar bus, and generate a bus acknowledge signal. A scalar bus write access to either address will cause both boards to load the relevant register from the scalar bus.

Each memory bank is implemented with eleven 1M by 8-bit SIMM modules, while the queues are built from **Texas Instruments SN74ALS2232** 64-word deep, 8-bit wide FIFO devices [46].

6.10 The vector Arithmetic Section

The Arithmetic Section board carries the Arithmetic Section of the vector pipeline (*ie* the Index Match circuit, the Arithmetic Unit, and the Vector Output circuit), plus the interfaces between the scalar and vector processors (the Vector Instruction Queue, the Instruction Identifier Register, the Scalar Result Queue and the vector exception and trap enable registers). It also carries the Overall Pipeline Controller, which splits each vector instruction taken from the Vector Instruction Queue into the sub-instructions required for each of the three pipeline sections, and controls the overall timing of vector instruction execution. The Overall Pipeline Controller passes the Vector Read section subinstruction to the memory controller board, and the Vector Write section subinstruction to both the memory controller and Sideways List Unit boards, and it receives completion and exception signals back. The Arithmetic Section subinstructions are queued on the Arithmetic Section board itself, and, as the Arithmetic Section becomes free, a new subinstruction is decoded by the Arithmetic Section control circuit, to generate the required control signal sequences for the Index Match circuit, the Arithmetic Unit, and the Vector Output circuit.

A block diagram of the Arithmetic Section board is given in figure 6–7.

6.10.1 The Index Match circuit

Index Match input data is loaded into the X and Y input queues under the control of the Vector Read circuitry. For a vector instruction which reads two vectors from memory, the X and Y input queues will each contain a number (possibly zero) of index/value words from the vector, followed by an end-of-vector word. For a vector instruction which reads one vector from memory only, the X input queue will contain a number of data words followed by an end-of-vector word; the Y input queue will contain no words relating to the instruction. For a vector instruction with no operands read from memory, neither queue will contain any words relating to that instruction.

The output of the index match circuit feeds the Arithmetic Unit (AU) input queue, which is a 152-bit wide queue, capable of holding one 24-bit index and two 64-bit values per word. The Index Match circuitry performs the operations set out in section A.2, which may involve comparison of the index fields of the words at the front of the X and Y input queues, deletion of one of those words, copying of one or both of those words to the AU input queue, lookup in the vector register, and insertion of explicit zero values and/or missing indices.

Vector elements are processed asynchronously with their consumption by the Arithmetic Unit – the Index Match circuit continues to process input elements and generate output elements as fast as possible, until the index match operation is complete (see section A.2.3), delaying only if one of its input queues becomes empty or if the AU input queue becomes full. After all data words have been written into the AU input queue, an end-of-vector word is written – this may come from the input queue, or if there is no input from memory (IM modes **R1** and **I0**), it will be generated by the Index Match circuit itself.

The queues are implemented using the **SN74ALS2232** FIFOs, while the Index Match circuit itself is based around two **IDT49C402** data path devices, and an EPLD-based controller.

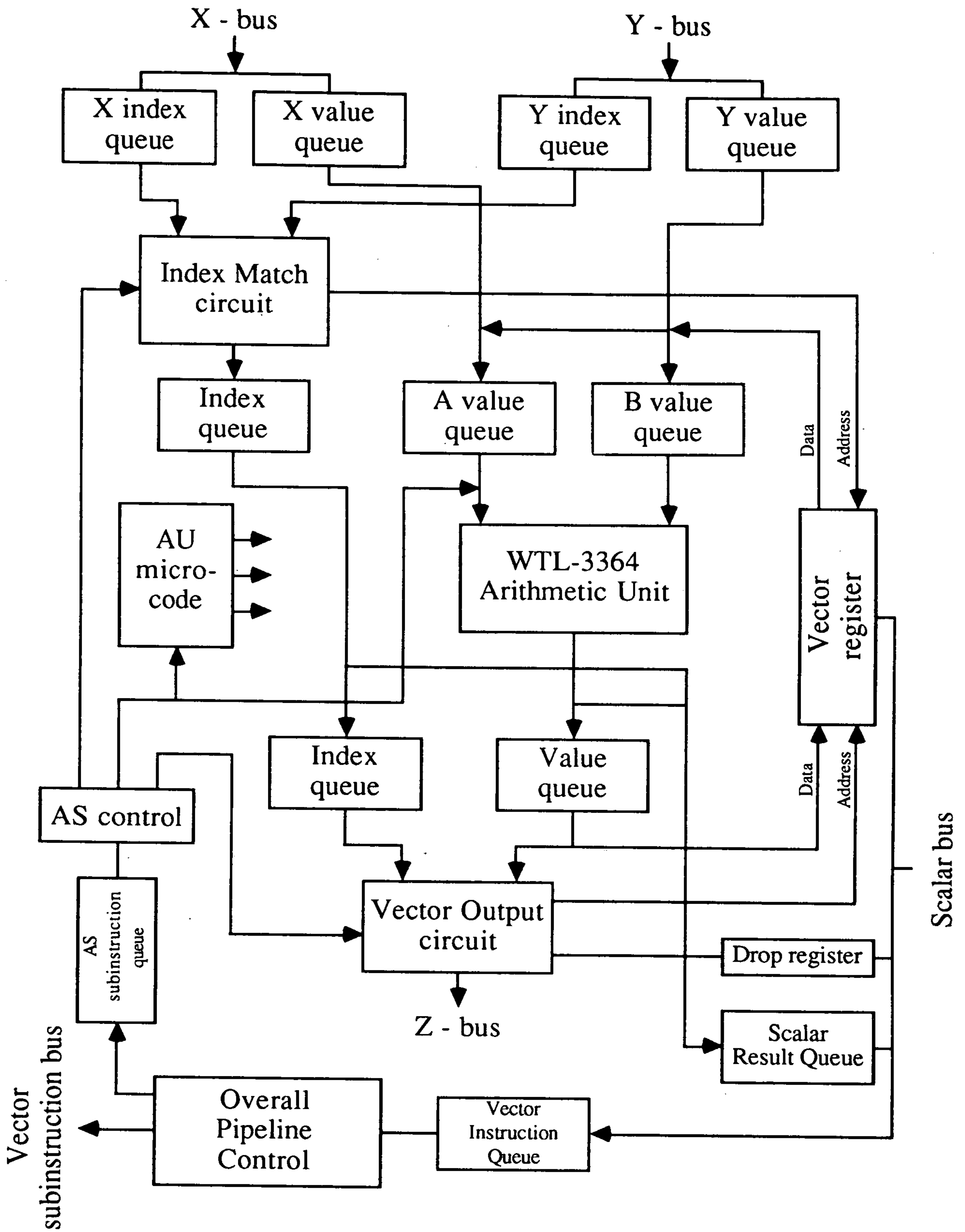

Figure 6–7: The Arithmetic Section board

6.10.2 The Arithmetic Unit

The design of the Arithmetic Unit centres around a single-chip arithmetic processor, which implements both integer and *IEEE* standard floating-point arithmetic. At the time of design, three manufacturers were supplying VLSI components with this capability: **Analog Devices**, **AMD** and **Weitek**. The AMD device (**Am29325**) has the lowest latency of all the available components; it is implemented internally in emitter coupled logic, and can perform a 32-bit add or multiply in a single 100ns cycle, without any pipelining. However, it does not support 64-bit arithmetic, a requirement for ESP. The Analogue Devices and Weitek parts are CMOS chips, and use pipelining to increase the throughput of the multiplier and adder. The Analogue Devices components are a chip pair comprising an adder (**ADSP-3220**) with a 3 clock cycle latency for 64-bit additions, and a multiplier (**ADSP-3210**) with a 7 cycle latency for 64-bit multiplications. Both can operate with a clock speed of up to 10MHz, and the adder can start a 64-bit operation every clock cycle. However, the multiplier throughput is limited by the 32-bit width of the internal multiplier array, and the relatively narrow I/O ports; as a result, it can only start one 64-bit multiply every 4 clock cycles. The Weitek part, the **WTL-3364**, is more highly integrated, containing separate add and multiply pipelines, plus a register file of 32 64-bit general purpose registers, on a single chip. It is capable of simultaneous pipelined 64-bit multiply and add operations, at a clock speed of 10MHz, with latencies of three clock cycles only. The I/O port bandwidth is sufficient to start a new three-operand (two input, one output) 64-bit operation every clock cycle. The chip can therefore support 64-bit vector addition at 10 MFlop/s and scalar product at 20 MFlop/s. The **Weitek** part was chosen to form the basis of the Arithmetic Unit.

Data transfers within the Arithmetic Unit, and the operation of the **WTL-3364**, are controlled by microcode generated by the Arithmetic Section control circuit, to implement the operations described in section A.3. The two input buses of the **WTL-3364** are usually fed directly from the two value fields of the Arithmetic Unit input queue, although scalar values may be input to the **WTL-3364** from the AS instruction queue. The output bus of the **WTL-3364**

is connected to the value field of the Vector Output circuit input queue, and also to the input of the Scalar Result Queue. Indices bypass the arithmetic chip, being copied directly from the index field of the AU input queue to the index field of the VO input queue.

The AU pipeline is halted if the AU input queue becomes empty, or the VO input queue or Scalar Result Queue become full.

6.10.3 The Vector Output circuit

Input data for the Vector Output circuit comes from the VO input queue, an 88-bit wide queue containing index/value pairs. If mode **D** is specified (see section A.4), the exponent of each input value is compared against the drop value register contents. If the input item exponent is less than the drop value exponent, the input item is discarded. Otherwise, if the **R** mode is specified, the input item is written into the vector register at an address given by adding the **register.offset** value from the vector instruction and the input item index field, and the item is removed from the input queue. If the **M** mode is specified, the input data item is driven onto the Z bus, and the Z bus ready signal is asserted to the Z transfer controller on the memory controller board. When the 'next item' signal is received from the memory controller, the item is removed from the both the Z bus and the VO input queue. This continues until the end-of-vector marker is found in the input queue. In mode **M** that marker is driven onto the Z bus as usual, while in mode **R** it is discarded.

6.11 The Sideways List Unit

The Sideways List Unit is illustrated in figure 6–8. The memory comprises 4M words of 48-bit wide memory (since the total memory size is only 4M words, the link field need only be 24 bits wide, rather than the full 32), implemented using 24 off 1M by 8-bit SIMM dynamic memory modules. The count and address

register banks are each implemented as a 32K bank of 24-bit registers, each using three 32K by 8-bit static RAM devices. A simple processor is provided to support incrementing and clearing of the count registers. Input data from the Vector Output circuit arrives via the Z bus into the SLU input queue. The Z bus hold signal is asserted by the SLU if the input queue becomes full.

The SLU receives a copy of the Vector Write pipeline section subinstruction over the vector instruction bus, and executes the SLU subinstruction specified (see sections 5.3.9 and A.5.1), using the **row.number** field from the subinstruction when relevant. The control circuitry handles refresh of the dynamic memory.

6.12 Testing ESP

At the time of writing, the host interface board, the control board, and the scalar processor are constructed and under test. Initial testing uses C programs compiled for the **68010** and loaded onto the host interface and control processor boards via the RS232 ports. This allows testing of the host interface access to the host memory, the host interface to control processor connection, and the ESP scalar bus. An assembler has been written for the ESP scalar instruction set, and the scalar processor is tested by loading programs prepared in assembler, into the program and data memories, and operating the scalar processor under the control of the control processor.

Initial tests on the vector memory and memory controller will be carried out by simulating control and data inputs, and testing access to vector memory from the scalar bus, using the control processor. Once construction of the vector Arithmetic Section is complete, the complete vector processor will be tested by feeding it single vector instructions direct from the control processor, over the scalar bus. Finally, the Sideways List Unit will be constructed and tested on single vector instructions.

Complete scalar/vector programs, prepared in assembler, will then be tested, including subsections of complete applications, such as sparse Gaussian elimination. Mounting of Linear Programming code on ESP will proceed by altering an

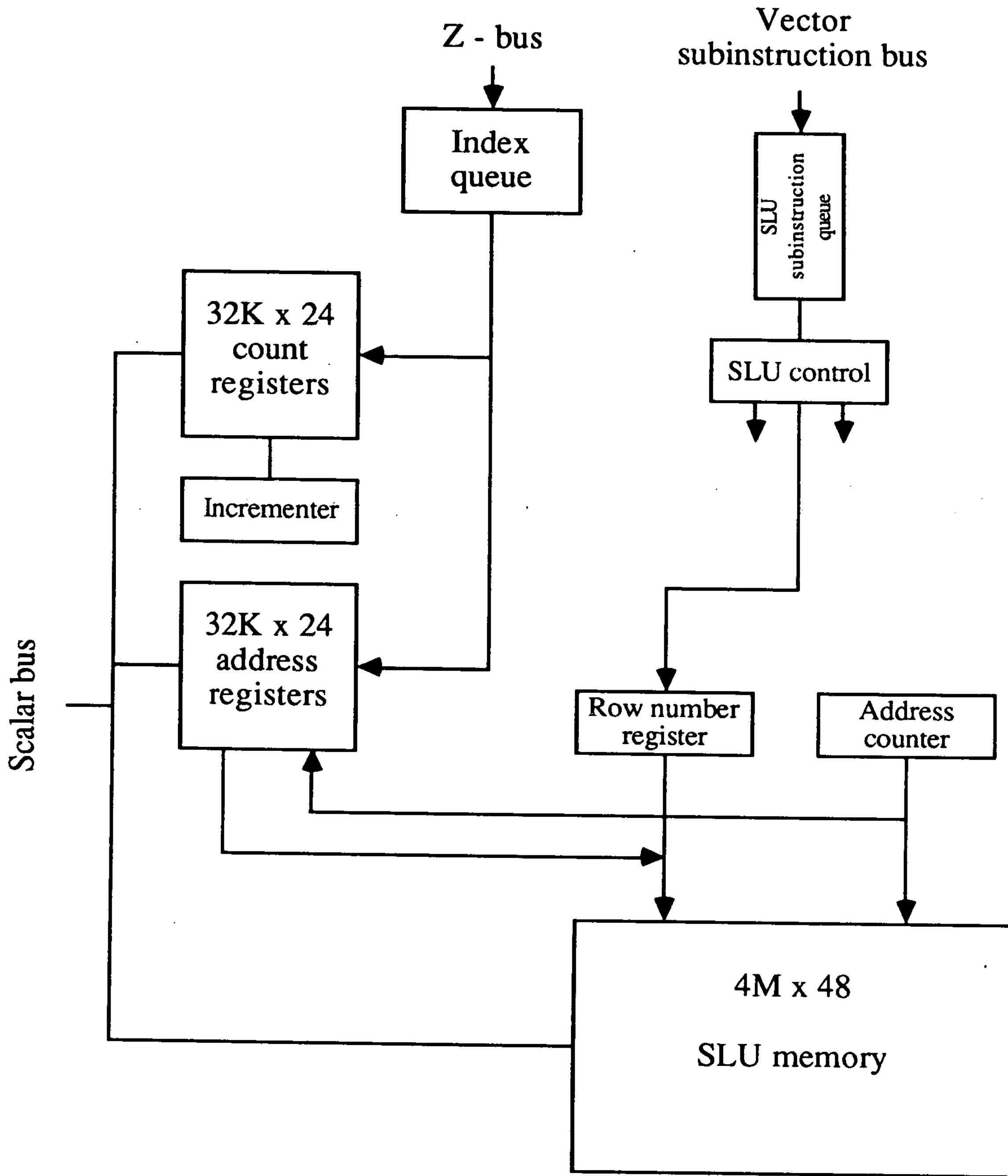

Figure 6–8: The Sideways List Unit

existing LP package running on the host workstation, so that time critical parts of the algorithm are transferred to ESP for execution. Initial work on a FORTRAN compiler for ESP is also underway.

Chapter 7

The Performance of ESP

7.1 Introduction

The mechanisms used by ESP to store and process sparse vectors were chosen after a simple pencil-and-paper analysis of the performance of the various alternatives (described in chapter 3). Similar analysis also guided the development of the architecture of the machine, suggesting important architectural features, such as the three section vector pipeline, and the decoupling of scalar and vector instruction execution. The guiding principle behind the architectural design was to obtain the maximum possible utilisation of the vector arithmetic unit. It was also important that this performance be met by hardware built using similar technology to that of the arithmetic unit, with the same clock rate. In this way, the new design should be capable of scaling to faster technology – increases in the speed of the arithmetic circuits should be matched by corresponding increases in the speed of the new memory control hardware.

Once the architecture was defined to the level described in chapter 5, it was clear that, if it could be implemented as described, the performance of individual sparse vector arithmetic operations would be several times better than conventional implementations on scalar or standard vector architectures. Vector/vector operations, such as add or scaled subtract, generate one output non-zero per ESP clock cycle (ignoring cancellation of result elements to zero). Because a sparse vector operation on a conventional machine requires either a scalar loop of several instructions, or a short sequence of standard vector instructions including scatter and gather, plus the overhead of memory management checks, the new architecture

should have a clear performance advantage for vector operations on sparse vectors with many non-zeroes. If the vector operands have few non-zeroes, the start-up time of the vector instruction becomes important. It was expected that each of the three pipeline sections could be designed with a start-up time of around 8 clock cycles, and so a performance degradation of a little over 50% might be expected for sequences of vector operations on sparse vectors with 6 or so non-zeroes, such as are commonly found in large Linear Programming problems.

What was not clear at early stages of the design process was whether the vector speed advantage of the new architecture could be successfully exploited on the target applications, or whether overall execution speed would be limited by the scalar processor, or by other unexpected bottlenecks in the design. Also unclear was whether the defined architecture could be implemented successfully, with the estimated start-up times and processing bandwidths.

To investigate the first of these questions, simulation studies on the Linear Programming target application were carried out by K.I.M. McKinnon, using estimated execution timings for scalar and vector instructions, provided by the author. The simulation method used was to augment the time-consuming parts of an existing Linear Programming implementation with code which kept track of the number of ESP clock cycles elapsed. The simulation accounted separately for scalar and vector processor clock cycles, on an instruction by instruction basis, and was therefore able to estimate the delays caused by synchronisation between the two processors. The results are described below, and were generally encouraging – no bottlenecks were found, and the loads of the two processors seemed to be well-balanced, with over 70% utilisation of both scalar and vector processors. The vector arithmetic unit was used to 30 to 50% of capacity on many problems (an excellent figure, even for a vector architecture). After the subsequent detailed design work on the prototype, it has become clear that the scalar processor timing estimates used in this simulation were over-optimistic by a factor of about two, and thus the average vector processor utilisation and arithmetic rates of the real prototype, on LP problems, will probably be 25% to 50% less than these results

suggest. However, the originally estimated scalar performance could probably be achieved by improvements to the current prototype scalar processor design.

These results gave some confidence that the architecture was reasonable. As more detailed design was progressing, two further simulation studies were carried out, both by students in the Department of Computer Science, under the supervision of the author. Both of these involved writing code to simulate the operation of ESP, using the simulation system SIM++ [30], a package for distributed discrete event simulation, which is based on the language C++. The first study, by A.G. Manning [35], simulated parts of the vector pipeline of ESP at the level of architectural detail described in chapter 5. The tests used simple sequences of various vector instructions, on array and list vectors of various sparsities. The second study, by Goh Boon Seng [20], simulated the scalar and vector processors at a rather more detailed level, incorporating some of the implementation detail described in chapter 6. Simulation experiments included execution of simple vector instruction sequences, and central parts of the Gaussian elimination algorithm, using matrices of various sizes and sparsities. The results of both these studies are described below.

7.2 The LP simulation study

The purpose of this study, by K.I.M. McKinnon, was to estimate the performance of the proposed ESP architecture on typical Linear Programming problems. The study was performed by adding code to an existing Linear Programming solver written in the language IMP [42]. The three intensive parts of the LP calculation (described in section 2.4), *BTRAN*, pricing and *FTRAN*, were examined in detail, and five LP problems, covering a range of typical problem structures, were used as test data.

7.2.1 The model

ESP was modelled by estimating the number of scalar processor clock cycles required for each scalar operation, and the number of vector processor cycles required for each phase of a vector instruction (*ie* start-up time and execution time, in each section of the pipeline). The modified IMP program kept track of the elapsing clock cycles as the computation progressed. The scalar processor/vector processor interface was also modelled – the Vector Instruction Queue was assumed to be able to hold three vector instructions in addition to any instructions actually executing, and the operation of the Instruction Identifier Register was modelled accurately. Most scalar operations were assumed to take one clock cycle if acting on operands likely to be in registers, while 2.5 clock cycles was allowed for the scalar processor to access operands in scalar or vector memory. Scalar floating-point add and multiply were assumed to take 2 and 4 cycles respectively. The start-up times for the three vector pipeline sections were estimated at 6, 7 and 6 cycles (VR, AS and VW respectively), and it was assumed that the vector Arithmetic Unit could start a new operation per clock cycle. Finally, it was estimated that the scalar processor would require 6 clock cycles to prepare a vector instruction and add it to the Vector Instruction Queue. If that queue was full when the scalar processor attempted to insert a new vector instruction, the scalar processor would enter a wait loop until space became available in the queue.

During each phase of the computation, the program kept track of the amount of time for which the scalar processor was executing useful instructions, and the amount of time it was blocked by a full Vector Instruction Queue. The amount of time the vector processor was executing instructions was also accumulated, as two figures – the time the Arithmetic Unit was performing useful work, and the time spent in vector instruction start-up. For comparison purposes, a second version of the program was prepared, which calculated the elapsed time for the same algorithm executing entirely on the scalar processor.

7.2.2 The results

For the *BTRAN* and pricing phases, behaviour was relatively straightforward. In each case, a lengthy sequence of vector instructions is issued – for *BTRAN*, a sequence of the special *BTRAN* instructions, one for each η-vector, and for pricing, a sequence of instructions performing scalar products between the price vector in the vector register and each column of the matrix $\mathbf{A}_I$. In both cases, the scalar processor simply loops, preparing the vector instructions. For the pricing step, the matrix columns are very sparse (six or so non-zeroes), while for *BTRAN*, the η-vectors are similarly sparse immediately after a re-invert operation, but the density of new η-vectors produced as iterations proceed rises to 10 or 20%. The study found that the vector processor utilisation was high (the vector processor was executing instructions 70% or more of the time), because the preparation of vector instructions by the scalar processor was overlapped with vector instruction execution. However, between one quarter and two thirds of the vector processor execution time (the proportion varied from problem to problem) was spent on instruction start-up, reflecting the very small number of non-zeroes in many of the vectors processed. For both *BTRAN* and pricing, the speed-up, compared with the simulated scalar-only implementation, was between six and twelve times for most test problems, but only three times on one test problem of unusual structure.

The *FTRAN* phase was more interesting. *FTRAN* consists of a sequence of scaled additions of the η-vectors into the vector register. The scaling factor for each

vector addition is itself an element taken from the vector register. At the start of the *FTRAN* phase, the vector in the register is very sparse, and it usually remains fairly sparse throughout. Because of this, many of the η-vector addition operations are unnecessary – the scaling factor is zero. The *FTRAN* implementation used in this study used a mixture of scalar and vector operations to identify and skip, as far as possible, the unnecessary η-vector operations. The was done by preprocessing (using a mixture of scalar and vector operations) the list of η-vectors to determine data dependencies between them. The scalar processor could then determine, as it prepared each η-vector scaled add operation, whether the relevant vector register element was zero, and whether any vector operations issued but not yet complete could possibly insert a non-zero into that register element. If the element was already non-zero, or could be made non-zero by a vector operation issued but not complete, the η-vector operation under consideration had to be issued into the Vector Instruction Queue, otherwise it could safely be skipped. The scalar processor was able to identify which of the vector instructions issued had yet to complete, by examining the Instruction Identifier Register.

Despite the complexity of the *FTRAN* algorithm, and the fact that the scalar processor is used for considerably more than simply loop iteration, the results again showed good utilisation of the vector processor. The η-vector data dependency check took between 10 and 50% of the total *FTRAN* execution time, depending on the problem, and over the whole *FTRAN* operation, the vector processor was executing 75 to 90% of the time. As with *BTRAN* and pricing, between one quarter and two thirds of that time, depending on the problem, was vector instruction start-up time. The scalar processor was doing useful work for between 50 and 80% of the time. For the problems involving denser η-vectors, the scalar processor spent up to 40% of the total *FTRAN* time waiting because the Vector Instruction Queue was full. During the data dependency check phase, the scalar processor was held up about half of the time, waiting for vector instructions to complete. The overall speed-up, compared to a scalar *FTRAN* implementation, was in line for that for *BTRAN* and pricing, namely six to twelve on most problems, but as little as three on peculiar problems. Within each problem, the relative speed-up of the

three phases, *BTRAN*, pricing and *FTRAN*, was about the same, so that although the proportion of time spent on each phase varies considerably between problems, the proportions remained about the same for the vector implementations of each problem as for the scalar implementations.

Further experiments were performed on the *FTRAN* phase to determine the effect of varying some of the estimated instruction execution times. An increase in all three pipeline section start-up times by two cycles (to 8, 9 and 8 for VR, AS and VW) reduced performance by 3 to 6%, while an increase in the time required for the scalar processor to issue a vector instruction, from 6 to 8 cycles, also reduced performance by 3 to 6%. Similar sized performance *improvements* were observed for 2 cycle *reductions* in each of these times. Over this range of variation, therefore, these times are not critical to the ESP performance, for this particular phase of the LP algorithm, at least.

Overall, the results gave confidence that the architecture could provide substantial performance improvement over a scalar architecture of the same technology. Despite the fact that the test problems were very sparse, the measured speed-ups of six to twelve times on most of them are similar to speed-ups achieved on conventional vector processors for dense matrix computation with long vectors. The proportion of the vector processor's time spent in vector instruction start-up confirmed the importance, for the LP application, of keeping the the pipeline sections short – start-up times much above 10 cycles would begin to have a considerable impact on performance. The advantages of the loose vector processor/scalar processor coupling were also confirmed, with the two processors working simultaneously at least 50% of the time. An interesting unexpected result was that, for the *FTRAN* operation, an increase in the length of the Vector Instruction Queue above three instructions caused a slight drop in performance. This was because the greater the number of issued but uncompleted vector instructions, the more often an unnecessary η-vector operation had to be issued just in case one of the uncompleted vector instructions changed its scaling factor (an element of the vector register) from zero to non-zero. Further experiments to find the optimal Vector Instruction Queue length could be made on the prototype.

7.3 The first SIM++ simulation study

This study [35] was carried out in late 1991 and early 1992 by A.G. Manning, under the supervision of the author. At the time, the detailed implementation of ESP had not been developed, and so the simulation model was based on the architecture level description given in chapter 5. The aims were to examine the performance of the architecture both on simple vector instruction sequences and on complete sparse Gaussian elimination problems, and to identify the performance limiting parts of the design. The discrete event simulation system SIM++ was used, and C++ code was developed to simulate the behaviour of each part of the simplified ESP architecture shown in figure 7–1.

7.3.1 The model

The simulated behaviour of each vector pipeline section was straightforward. The Vector Read section was modelled as having a constant start-up time, after issue of its vector subinstruction, before it began producing output. It was then assumed to transfer, from vector memory into its output queues, one vector element for each input vector of the instruction, per clock cycle. The start-up time models the subinstruction decode time plus the delay in fetching the first vector element from memory. Similarly, the Vector Write circuit was assumed to transfer one element from the Arithmetic Section output queue, to memory, per clock cycle, starting a constant start-up time after the issue of the Vector Write subinstruction. The Arithmetic Section was also modelled as having a constant start-up time after subinstruction issue. After that time, it was assumed to remove one element from one or both of the Vector Read output queues per clock cycle. The operation of the Index Match circuit was modelled fully, and determined whether one or two elements were to be fetched from the Vector Read queues in each clock cycle. The timing of the rest of the Arithmetic Section was modelled simply as an instruction-dependent pipeline delay: the delay between the fetching of each element or pair of elements from the Vector Read output queues, and the load-

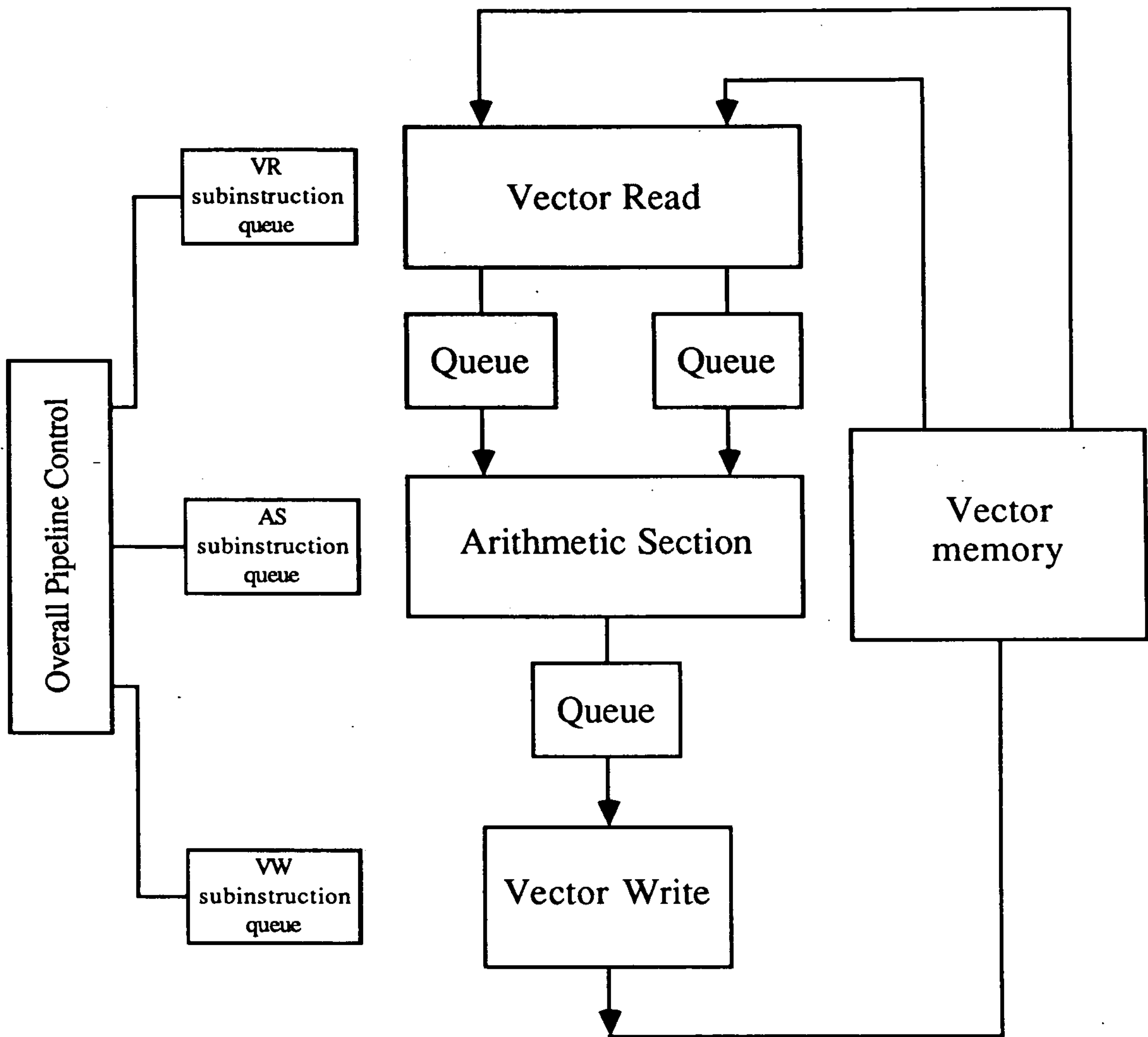

Figure 7–1: ESP as modelled by first SIM++ simulation study

ing of the corresponding result (if any) into the Arithmetic Section output queue. The various subinstruction start-up times were estimated, taking into account the complexity of the respective pipeline section initialisation operations, as follows: Vector Read - 8 cycles; Arithmetic Section - 4 cycles; Vector Write - 8 cycles. The Arithmetic Section pipeline delay was estimated from the known characteristics of the **Weitek** floating point unit, plus an allowance for the delay in the Index Match and Vector Output circuits, and was set at 6 cycles for an add or multiply operation, and at 9 cycles for a scaled subtraction operation.

Vector memory was modelled as a simple system capable of satisfying simultaneous access for two vector read elements and one vector write element, per clock cycle. Written vectors were not actually stored, and the indices and values of vectors read from memory were generated randomly as they were read, with simulation parameters controlling their size and sparsity. The simulation was driven by a model of the vector processor's Overall Pipeline Control circuit, which simply issued a preset sequence of vector instructions into the subinstruction queues for the three pipeline sections.

This simple model differs from the real implementation in a number of important respects. The Arithmetic Section model is a simplification – the queues between the Index Match circuit, the Arithmetic Unit, and the Vector Output circuit are not modelled, and thus the time behaviour of the complete Arithmetic Section is oversimplified. The timing of the vector memory was not explicitly modelled at all – it was simply assumed that two vector elements could be read, and one written, per clock cycle, with no competing accesses by the scalar processor. Thus no account was taken of contention for the vector memory, or of the organisation of the memory into banks. The scalar processor and its interface to the vector processor were not included in the model.

7.3.2 The results

The experimental tests carried out with this model concentrated on checking the behaviour of the data queues between the vector pipeline sections. The simplest

test involved logging the number of items in the Vector Read output queues and in the Arithmetic Section output queue, during execution of sequences of addition operations on pairs of *array* vectors. The behaviour of the queues in these circumstances is determined only by the relative start-up times of the three pipeline sections. As was expected, the Arithmetic Section output queue never contained more than one item, because the modelled Vector Write start-up time (8 cycles) was less than the total start-up time for the add operation in the Arithmetic Section (4+6 cycles). On the other hand, because the Vector Read start-up time in this model was 8 cycles, the Vector Read section was able to proceed through the list of instructions slightly faster than the Arithmetic Section, leading to a gradual build-up of vector elements in the Vector Read output queues. Overall performance, as expected, was limited by the longest of the three section start-up times, in this case the 10 cycles for the Arithmetic Section.

The behaviour of the queues is rather more interesting when the operands are *list* vectors. Tests with vectors of densities ranging from 1% to 40%, with randomly distributed non-zeroes, showed again that the output queue of the Arithmetic Section was not used, while data built up in the Vector Read output queues more rapidly than was the case for array vectors, because the Index Match circuit was often reading an element from only one of the two queues.

None of these results was surprising – they could have been predicted by a simple analysis of the model, and indeed with such a simple model of the vector pipeline, it is unlikely that any unexpected behaviour would be uncovered in simulating straightforward sequences of vector instructions. It had originally been planned to extend this model to simulate execution of a complete Gaussian elimination program, by adding a model of the scalar processor. This would have highlighted potential problems due to the scalar/vector processing rate ratio or the scalar processor/vector processor synchronisation mechanisms. However, the necessary extensions to the model were not completed.

7.4 The second SIM++ simulation study

This study [20] was carried out over the period May to September 1992, by Goh Boon Seng, under the supervision of the author. By that time, a considerable amount of detailed work on the implementation of ESP had been completed. The aims of the study were similar to those of the earlier SIM++ study: to test the performance of the machine on a variety of vector operations, and on the Gaussian elimination algorithm, but using a more accurate model of ESP than the earlier study.

7.4.1 The model

The vector processor was modelled to the level of detail illustrated in figure 5–3, except that the Trap registers were not modelled, nor were the vector register, the Garbage Collection circuit and the Sideways List Unit. The Overall Pipeline Control circuit, and the three control circuits for the pipeline sections, were modelled as single cycle delays to allow for instruction decoding. All of the instruction and data queues were modelled accurately, from specifications of the FIFO devices used to implement the queues in the prototype. The Vector Read and Vector Write circuits were modelled closely on the expected behaviour of the prototype implementations, described in sections 6.8.1 and 6.8.2. The link memory and descriptor memory were not included in the model, but the time for descriptor access was built into the start-up time for both VR and VW. VR was estimated to require 4 cycles start-up time for reading list vectors, and 3 cycles for array vectors, while VW timing allowed 3 cycles start-up time for writing an array vector, and for a list vector, 2 cycles start-up time, plus an additional 4 cycles to update the descriptor at the end of the instruction (these times are all in addition to the one cycle subinstruction decode time). It was assumed that, once started, VR and VW could both generate one four-element memory access request every clock cycle. The model is slightly optimistic in that it does not model the detailed timing of arbitration of the link and descriptor memory between the Vector Read

and Vector Write circuits. Descriptor memory clashes would lead to extra delays if both VR and VW were to start an instruction simultaneously, and link memory clashes will slow down slightly the rate of generation of memory requests when list vectors are being accessed. The vector memory was modelled on the implementation described in section 6.9. The memory was assumed to be capable of one read or write operation across all four banks, per cycle, and arbitration of memory requests was modelled using two different arbitration protocols, for comparative purposes. Memory refresh cycles were ignored – these have only a very small effect on the overall memory bandwidth.

Within the vector processor Arithmetic Section, the Index Match circuit was assumed to process one input element (or pair of elements) per clock cycle, while the Arithmetic Unit was modelled as an instruction-dependent pipeline delay (ranging from one cycle for a no-op, through three for an add or multiply, to five cycles for a scaled subtract instruction – these times were calculated from the detailed data for the **Weitek** floating point device). The Vector Output circuit was modelled as a single-cycle pipeline delay.

The scalar processor was not modelled in detail, but was used to drive the rest of the simulation, by issuing vector instructions to the Vector Instruction Queue. The timing of vector instruction issues was determined by careful consideration of the timing of the scalar code which would be used to implement the modelled applications on the actual prototype scalar processor, the **Weitek XL-8364**. For example, the scalar instructions to iterate a simple loop were estimated to take 4 cycles, while the actual issue of a vector instruction into the VIQ was modelled as taking 10 to 14 cycles depending on the instruction.

7.4.2 Results for vector instruction loops

Two sets of experiments were performed. In the first, sequences of repeating vector instructions were fed to the vector processor. The timing of vector instruction issue took into account the scalar processor operations involved in constructing the vector instruction and in loop iteration. Three different vector instructions

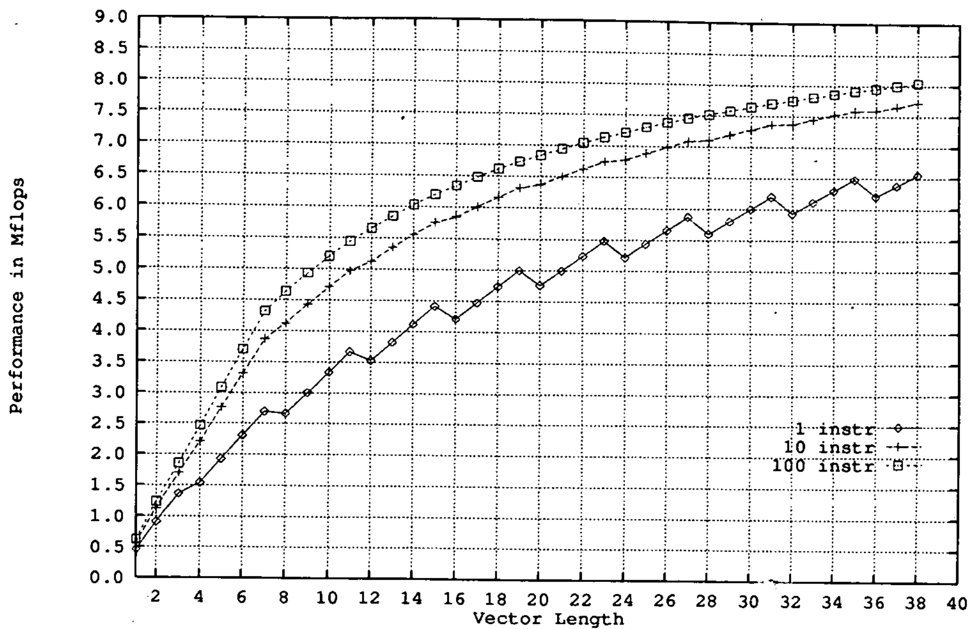

Figure 7–2: Performance vs. vector length for scalar add [20]

were simulated: the addition of a scalar to each of the elements of a vector, the element-wise multiplication of two vectors, and the scaled subtraction of one vector from another. The first two instructions were simulated on array vectors; the third on list vectors, but there it was assumed that the non-zeroes in the two vector input operands coincided exactly. Tests were carried out in each case with a single vector instruction, a loop of ten instructions, and a loop of 100 instructions, and were repeated for vector lengths (number of non-zeroes, in the list case) of one to 38. The resulting overall arithmetic rates, assuming a clock cycle of 100ns, are plotted in figures 7–2, 7–3 and 7–4.

For vectors of length 38, the performance achieved is almost 80% of the theoretical maximum performance of 10 MFlop/s for the first two operations and 20 MFlop/s for the scaled subtract. Operations on longer vectors would reach even higher performance, asymptotically approaching the theoretical maximum. The vector half-performance length (*ie* the vector length for which half the asymptotic performance is achieved) can be read from the graphs, and is 20 to 24 for a single vector instruction, reducing to 8 to 11 for a loop of 100 instructions. This reduction is due to the overlapping of vector instructions in the pipeline sections, and

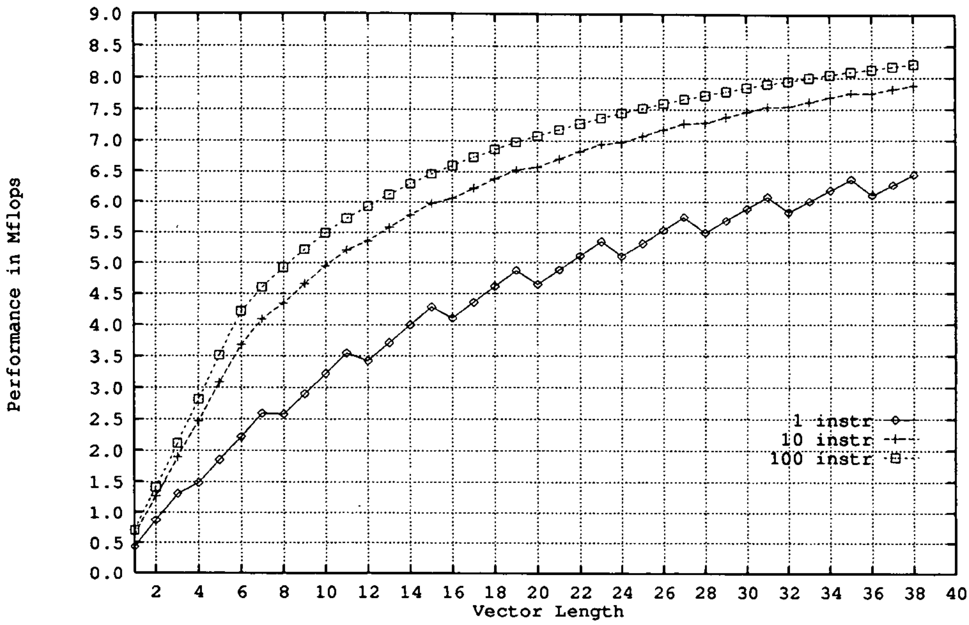

Figure 7–3: Performance vs. vector length for multiplication [20]

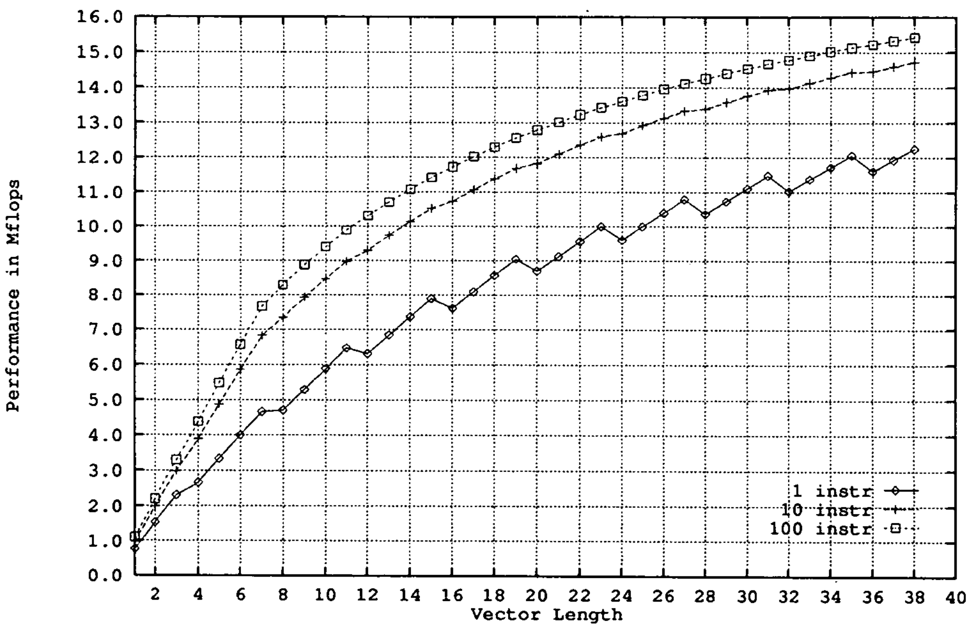

Figure 7–4: Performance vs. vector length for scaled subtraction [20]

to the relative reduction in effect of the initial fixed scalar overhead for issue of the first vector instruction. Additional interesting features of the graphs are the jagged nature of the single instruction graph, due to the four-way memory bank interleaving, and the linear section of the graphs for the 10 and 100 instruction loops, for vector lengths up to six or seven. This last feature occurs because the loop iteration time is constant for vector lengths up to six or seven, limited by the minimum 14- to 18-cycle scalar execution time per iteration. For longer vectors, the vector instruction execution time limits the loop iteration rate.

Other results from these experiments were that, as expected, the Vector Read section of the pipeline ran ahead of the Arithmetic Section, so that the VR output queues filled up, as did the AS and VW section subinstruction queues. The other queues did not fill at all. The Vector Read section speed exceeded that of the Arithmetic Section by a considerable margin, and it seems certain that it would continue do so with the small extra delays caused by vector memory refresh cycles, and descriptor and link memory arbitration, taken into account.

7.4.3 Results for Gaussian elimination

The second set of experiments involved simulating the central part of the Gaussian elimination algorithm. Because of the limited time available for program preparation, it was decided not to attempt to simulate threshold pivoting (section 2.2.5), but rather to simulate the elimination part of the algorithm under the assumption that the matrix is already permuted to arrange a good pivot sequence down the diagonal. This corresponds to actual practice for matrices (such as symmetric and positive definite ones) where numerical stability is not an issue, and the pivot sequence can be chosen so as to minimise fill-in, before factorisation starts. It was assumed that the positions of the non-zeroes in each pivot column had been precalculated during the selection of the pivot sequence, and were available in scalar memory, and it was further assumed that the rows and columns of the matrix had been physically permuted, so that the pivot column element was always the

first non-zero of the rows involved in subtraction operations. The lower triangular factor of the matrix was to be discarded.

The time for the scalar processor to set up and issue each scaled subtract operation was calculated from the detailed data for the prototype **Weitek** scalar processor. In order to simulate accurately the timing of each vector subtraction operation, the number of non-zeroes in each row at each stage of the elimination was estimated from a simple model of the progress of fill-in. The simulation was performed with three different matrix models, based on real sparse matrix problems discussed in [15]. The small matrix model was of order 130, with 713 non-zeroes, the medium model was of order 147, with 1298 non-zeroes, while the large model was of order 1176, with 9864 non-zeroes. In all three cases, real data was available about the amount of fill-in to be expected at each stage of the elimination, and the fill-in model for each matrix was based on that data.

In order to calculate the scaling factor for each vector subtract operation, the scalar processor must access the non-zeroes in the pivot column, by reading them from the row vectors in vector memory. Thus the mechanism for scalar processor access to vector memory must be modelled, and this model was based on the implementation described in section 6.8.4. It was assumed that the scalar processor would require three cycles to access the vector descriptor, followed by three cycles to generate a vector memory access request, and one cycle to read each 32-bit half of the vector element once the memory access request was satisfied by the vector memory arbiter. Two alternative vector memory arbitration mechanisms were modelled – *round robin*, in which pending memory requests for the two Vector Read data streams, the Vector Write stream, and the scalar processor, are serviced in round robin fashion, and *scalar priority*, in which any pending request from the scalar processor is always serviced first, with Vector Read and Write requests then serviced in round robin order.

The results indicated an average performance during factorisation of the small matrix of 5 MFlop/s; for the medium matrix, 10 MFlop/s, and for the large matrix 15 MFlop/s (the theoretical maximum arithmetic rate for scaled subtraction operations is 20 MFlop/s). It should be noted that a "Flop" here denotes any

floating point operation passing through the Arithmetic Unit, *including* the addition of a zero to a non-zero, when the non-zero indices in the two input vectors do not match. Given that the number of non-zero output elements produced by each vector subtract operation ranges from 5 to 16 for the small problem, from 9 to 31 for the medium problem, and from 8 to 80 for the large problem, these arithmetic rates are rather lower for the small and medium problems than might be expected from the graphs in figure 7–4. The explanation is that for the small and medium problems, the *scalar* processor limits the overall performance. For these smaller problems, the scalar processor takes longer to read the pivot column element, calculate the scaling factor and issue the vector instruction, than each section of the vector pipeline takes to complete the vector instruction, so that the vector processor is not kept supplied with vector instructions fast enough to take full advantage of the overlapping of instructions in the pipeline sections. However, the use of scalar priority arbitration for the vector memory, rather than round robin arbitration, increased the performance by only 1/2%, indicating that contention for the vector memory was not significantly delaying the scalar processor.

7.5 Summary

The first simulations to be carried out, those based on Linear Programming code, modelled ESP in a crude way, using early estimates of instruction timings. Nevertheless, the measured performance gain over a scalar version of the same algorithm – six to twelve times on many problems – and the high degree of overlap observed between execution in the scalar and vector processors, indicated that the sectioned vector pipeline and the decoupling of the two processors would be successful architectural features of the machine. Although tests showed that an increase in the longest pipeline section start-up time from 7 to 9 cycles decreased *FTRAN* performance by no more than 6%, it was clear from the fraction of vector processor execution time spent in instruction start-up that the three-fold increase in start-up time which would be incurred if the pipeline were not sectioned would reduce performance greatly. Similarly, although an increase from 6 to 8 cycles in the time

required for the scalar processor to issue a vector instruction reduced *FTRAN* performance by no more than 6%, the fact that many vectors processed have only six or so non-zeroes indicates that an increase in the scalar processor overhead for each vector instruction much above 14 cycles would reduce performance substantially, as the vector processor would not be kept busy. This effect would probably be worse for the LP pricing step, as the average number of non-zeroes per vector is smaller there than for *FTRAN* and *BTRAN*.

After more detailed design of the ESP prototype had been completed, it was possible to construct a much more accurate model of the machine. By the time of the second SIM++ study, it was clear that the original estimates of the pipeline section timings had been reasonably accurate, but that, as designed, the prototype scalar processor would be rather slower than estimated originally. In particular, scalar processor accesses to vector memory are expected in the prototype to take up to 10 cycles, including vector descriptor access, and the construction and issue of a vector instruction may take 10 to 14 cycles. The SIM++ simulations indicated that for simple vector instruction loops the scalar processor was the limiting factor in performance when the vectors had fewer than 7 non-zeroes, while for the more realistic code of the Gaussian elimination model, the scalar processor limited the overall performance until the average number of non-zeroes per vector reached 30 or more. Nevertheless, good overall performance was measured on both simple vector loops and the Gaussian elimination model. Vector half-performance lengths were measured at only 8 to 11 for the simple vector instruction loops. These figures are much lower than corresponding figures for commercial vector processors, and although the difference is partly due to the slower technology of ESP, which reduces the number of clock cycles required for memory access and data communication, it is also a reflection of the sectioned pipeline architecture. The overall performance of 5, 10 and 15 MFlop/s respectively on the three Gaussian elimination tests was also impressive, representing 25, 50 and 75% utilisation of the arithmetic unit; many times higher than could be expected with scalar or conventional vector processor implementations of similar sparse matrix code. However, it should be borne in mind that threshold pivoting operations, and storage of the

lower triangular factor, were omitted in these models, and that those operations will add to the amount of scalar processing to be carried out during elimination, increasing the effect of slow scalar processing. The clear lesson is that attempts should be made to improve on the scalar performance of the prototype, which at present is limited by implementation choices, rather than real technological constraints.

Chapter 8

Summary and Conclusion

The vector processor architecture which has been developed for ESP contains the following innovative features:

- the Index Match and Vector Output hardware of the Arithmetic Section, which support direct execution of vector arithmetic on sparse vectors stored in compressed index/value form, using the in-phase scan algorithms described in chapter 3;

- the Vector Read, Vector Write, and Garbage Collection circuitry, which support the reading and writing of sparse vectors as lists of linked blocks, and the management of memory space as vectors fill-in;

- the Sideways List Unit, which supports the maintenance of information concerning the two-dimensional structure of sparse matrices;

- a three-section vector pipeline, with asynchronous instruction queues for each section, and data queues between sections, to reduce effective vector instruction start-up times to a minimum;

- scalar and vector processors which are loosely coupled via the Vector Instruction Queue and the Instruction Identifier Register, which identifies the last completed vector instruction.

The development of each of these features has been required to satisfy the principal aim of the project: the design of a machine able to support efficient sparse vector arithmetic in the context of the target applications of Gaussian elimination and Linear Programming.

A prototype implementation of the new architecture has been designed, using off-the-shelf components, including wherever possible the use of erasable programmable logic devices, to simplify modification of the design. The prototype, which will occupy approximately 5000cm^2 of circuit board space, is currently under construction. When complete, it will be hosted by a standard workstation, to which it will be connected by a dedicated high-bandwidth link: programs and data prepared on the workstation will be transferred to the ESP for execution, and results transferred back.

The architecture has been partially evaluated by a number of simulation experiments, the most recent of which have been able to take into account performance predictions for each part of the machine arising from the detailed design of the prototype implementation. The SIM++ simulation model currently models most parts of the vector pipeline, but does not yet include models of the Garbage Collection circuit and Sideways List Unit.

8.1 Evaluation of the ESP architecture

At the time of writing, only a part of the prototype ESP has been built and tested. Undoubtedly the detailed design will be refined as testing continues, but work has progressed far enough to give some confidence that implementation of the proposed architecture is technically feasible. In terms of circuit board area, the special features of the machine occupy the majority of three 30cm square boards, while the vector memory, scalar processor, and interface to the host are on four more. In terms of silicon area, were the machine to be implemented in a set of VLSI devices, the proportion of the complete scalar/vector processor silicon area (without memory) taken by the new features would be much less.

8.1.1 The sparse vector handling mechanisms

The sparse vector handling mechanisms are incorporated in the Vector Read, Index Match, Vector Output, Vector Write, and Garbage Collection circuits. These facilities are required if the advantages of vector processing (the concurrent use of addressing and memory access hardware to keep the arithmetic unit fully supplied with data during a vector instruction) is to be realised for sparse vectors. Consideration of the alternatives (chapter 3) indicated that the chosen mechanisms would be the most effective, overall, on any typical sparse vector computation. The detailed design of the vector memory, and Vector Read and Write circuits is complete, while the behaviour of the Index Match and Vector Output circuits is straightforward and predictable. Simulations of the complete pipeline plus memory have been performed (section 7.4), and these indicate that high arithmetic unit utilisation will be achieved from the design.

The only part of this section of the design which is not yet fully determined is the Garbage Collection algorithm. As discussed in section 4.2.4, the proposed simple algorithm may take a long time to execute in some circumstances. Whether these circumstances will occur in practice will depend on the distribution of the blocks of list vectors through memory, which itself depends on the order in which the application algorithm carries out vector computation. The Garbage Collection circuitry is being designed using programmable logic, and with sufficient functionality to implement an alternative de-allocation strategy with a much lower worst-case execution time, if this turns out to be necessary. The effectiveness of the simpler algorithm can only be tested by extending the simulation model to model the entire machine on real applications (so that the data-dependent pattern of memory use is modelled), or by testing the applications on the completed prototype.

8.1.2 The Sideways List Unit

The Sideways List Unit (SLU) is required because information about the non-zero structure of a sparse matrix by both column and row is needed by many

sparse Gaussian elimination algorithms (section 3.8). Without the SLU, *scalar* code would be needed to maintain the column non-zero structure of a matrix stored as row vectors, and the execution time of that scalar code would obliterate the speed advantages of the vectorised matrix row arithmetic. The detailed design of the SLU is not yet complete, and its operation has not been modelled in the simulations. However, the latency of the SLU operation is not a critical factor in the speed of the machine, while the bandwidth required for accesses to the dedicated SLU memory during a vector operation is never more than the Vector Write bandwidth for the same operation, *ie* a fraction of the total bandwidth required from the vector memory itself. Thus it is not anticipated that major problems will arise in this part of the design, or that the incorporation of the SLU will impact on the performance of the rest of the machine.

The Sideways List Unit is perhaps the least satisfactory part of the whole design; it presents the appearance of an afterthought bolted on the side. This appearance, however, belies the design history of the machine. Considerable thought went into the search for a storage mechanism for sparse matrices which would support rapid and efficient row-wise *and* column-wise access, despite the constantly changing non-zero structure. A suitable mechanism could not be found, and the Sideways List Unit was the best alternative.

For simplicity of design and construction, the prototype SLU has its own memory for storing the non-zero position lists. The number of words in this memory needs to equal the number of vector memory words, to allow full use of the vector memory on a Gaussian elimination problem, but in many algorithms the SLU is not used, and its memory is wasted. It was originally planned to have the SLU share the main vector memory; a more flexible approach. Since the memory bandwidth required by the SLU is never more than a quarter of the vector memory bandwidth provided in the current implementation, it would not be particularly difficult to provide sufficient extra vector memory bandwidth. However, the memory arbitration and data bus routing become more complex. In any future redesign, this complexity should be weighed against the flexibility of having one large memory rather than two.

8.1.3 The vector pipeline control strategy

The partitioning of the vector pipeline into three sections was introduced when it became clear that the total pipeline length (and hence minimum start-up time) would be at least 20 cycles. Many of the vectors occurring in large Linear Programming problems contain around six non-zeros, and with a start-up time of more than 20 cycles, operations on such vectors could achieve no more than 25% of the maximum vector arithmetic performance, at best. The three stage pipeline partition should reduce the effective vector instruction start-up time (during sequences of rapidly issued vector instructions) to the length of the longest of the three pipeline sections, and so should double the speed of processing such short vectors.

The vector pipeline control mechanisms, and the data queues between the sections, were modelled in the SIM++ simulation (section 7.4). The results indicated that the effective vector instruction start-up time was reduced from between 20 and 24 for a single vector instruction, to between 8 and 11 for a sequence of instructions, provided that the scalar processor was able to issue those instructions fast enough to ensure that the Vector Instruction Queue was never empty.

8.1.4 The scalar processor/vector processor interface

In order to take advantage of the low start-up time supported by the sectioned vector pipeline, the next vector instruction must be ready before the first pipeline section finishes executing the current instruction. If the scalar processor is always able to prepare a vector instruction in time, full use will be made of the vector pipeline. However, in some parts of the target applications, in particular the *FTRAN* phase of the LP algorithm, the scalar operations required between each vector instruction are complex and data dependent. A queue for vector instructions was therefore incorporated, to help keep the vector processor busy, despite variations in the amount of scalar code to be executed between vector instruction issues.

Even without the Vector Instruction Queue, to take full advantage of the three pipeline sections, up to three vector instructions must be able to execute at once. With the incorporation of the queue, an even larger number of vector instructions may have been issued but not yet completed. The scalar processor in ESP must be able, wherever possible, to continue execution, despite the fact that several vector instructions issued have not finished (or even started). For the reasons described in section 5.4.2, it was decided to check for data dependencies between instructions in software, and this required the Instruction Identifier Register mechanism.

The effectiveness of the scalar processor/vector processor interface design will not be determined until accurate simulations are carried out on complete applications, or until the prototype becomes available for testing. However, a simple model of the scalar/vector interface was incorporated into the Linear Programming simulation experiments (section 7.2), and although the model and its parameters were approximate, the results indicated that neither the scalar nor the vector processor suffered serious delays during any phase of execution.

8.1.5 The implementation

The decision to base the design of much of the prototype around general purpose data path components controlled by programmable logic devices, built using wire-wrap technology, was intended to make easy the inevitable changes which will be required to the design (especially since simulation tests of the complete design have not been carried out). As prototype testing is at an early stage, and the most innovative parts of the design have not yet been tested, it is not yet known to what extent such changes will be required. Since the prototype technology and speed (8MHz) have been kept deliberately conservative, it is hoped that there will be no major unanticipated technological problems. What is already clear, however, is that the specific design of the scalar processor in the prototype has resulted in a lower scalar performance than originally anticipated, and that this could well prove to be the limiting factor in the performance of the machine on many sparse problems (section 7.4). Work is underway to re-examine the scalar

processor design, and its interface to the vector memory and Vector Instruction Queue, to increase the scalar performance while retaining the existing clock speed.

8.1.6 Scalability of the design

The implementation details of ESP are such that the speed of each part of the processor should scale linearly if re-implemented in faster gate technology. As with a standard vector processor though, as the processor speed is increased, more memory banks are required to supply the necessary memory bandwidth. The performance of the machine should not be greatly affected by a change to eight memory banks. However, because the design requires that list vector lengths be a multiple of the number of memory banks, any further increase in the number of banks, for example to 16, will increase both the amount of wasted memory per vector, and, more importantly, the effective vector start-up time. This could cause a considerable performance drop on problems involving very short list vectors, such as occur in large Linear Programming applications.

Limiting the number of memory banks to eight would mean a maximum clock speed of around 20MHz at current dynamic memory speeds. To increase speed much beyond this, greater use of vector registers, or some kind of cacheing system would be required. The effectiveness of a cache would depend on the pattern of vector access by the application, however, and experiments on the ESP prototype will be needed to examine the feasibility of a cache, and to estimate the size required.

The memory bandwidth required in a vector machine also presents a further difficulties for implementation of the architecture in VLSI. VLSI integration levels have almost reached the point where it is feasible to integrate an entire vector processor on a single chip, and a clock speed of 100MHz or higher might be possible on-chip. However, the external interfaces of the chip are unlikely to be able to support data transfers at this speed, and the pin-out of the chip would be limited by the I/O pad dimensions. Thus the full performance could only be achieved,

again, if much of the operand data could be held on chip, in vector registers or cache.

8.2 Evaluation of the design methodology

The choice of the basic sparse vector mechanism was very much dictated by the chosen aims of the project, and by technological constraints. The validity of other design choices, however, like the number of pipeline sections and division of functions between sections, and the details of the vector/scalar processor interface, are less clear. These choices have been partially supported by the simulations which have been carried out, but the Linear Programming simulation was based on a rather crude model of the processor, while the SIM++ simulation only simulated a simpified version of one of the target applications. Thus, the detailed implementation work commenced without fully validating the architecture, by simulation. This has necessitated a flexible design, with much use of programmable logic, but there is a limit to the extent to which it will be possible to modify the architecture once the prototype is constructed. Some parts of the implementation, in particular, the Garbage Collection algorithm, have had to be left undecided until tests indicate which algorithm is suitable in practice.

Similarly, construction of the prototype has been proceeding with little advance verification of the details of the design. The SIM++ simulation of the vector pipeline was carried out at the same time as the detailed design of parts of the pipeline, and although some results of the simulation experiments were fed back into the design process, the simulation is currently at a more abstract level than the implementation, and is unable to check many of the details. The potential problems, for example design changes required in the middle of construction, are clear, although they have not arisen yet. These potential problems could have been minimised by more extensive use of simulation earlier in the project. Ideally, a simulation of the architecture, modelling instruction execution and expected timing, would have been developed, and used to model execution of the target

applications to determine the effect on performance of each of the architectural features. That model could then have been refined later in the light of information about timings, and other details indicated by the prototype design process. Gate-level simulations of some parts of the prototype might also have assisted development.

Instead, for a variety of reasons, design and construction of the machine has outstripped testing through simulation. One major reason for this was that it was originally planned to emulate the ESP architecture using an **ORION** minicomputer [22], which incorporates several writeable microcode stores, between which the processor can switch on a context switch, allowing different tasks to be executing different instruction sets simultaneously. However, it became apparent, as attempts to microcode the ESP instruction set were made, that the architectural gap between the **ORION** micro-architecture and the ESP architecture was too large to be bridged in microcode, and the emulation had to be abandoned.

Of the simulation modelling which was carried out, the Linear Programming based simulator developed by K.I.M. McKinnon, although a crude model, was relatively easy to put together, and provided an early indication of the feasibility of the architecture. For more accurate simulation, the SIM++ [30] system has proved suitable, and the model written by Goh Boon Seng (section 7.4) is well-structured and can form the basis of a more complete simulation, by adding models of the scalar processor and Sideways List Unit. Although it is intended to extend the SIM++ model in this way, many of the tests which should ideally have been made using a simulated or emulated model will probably be made on the prototype hardware. However, even with sophisticated logic analysers, it is much more difficult to measure the behaviour of hardware than a model, and there is limited scope for experiments which involve alteration of the architecture.

8.3 Further work on ESP

The detailed design of the Arithmetic Section of the vector pipeline, and of the Sideways List Unit, remain to be done. The latter is relatively straightforward, but the former requires considerable amounts of microcode, to implement the Arithmetic Unit control functions for the large number of different arithmetic operations provided. Construction will continue steadily – the prototype is expected to be complete by the autumn of 1993.

Testing of the complete machine will involve mounting the time critical parts of the Gaussian elimination and Linear Programming applications on ESP (the input/output, high-level control, and result presentation parts of the applications will be executed on the host workstation). ESP programming will initially be carried out in assembler, but investigation has started into the feasibility of developing a FORTRAN compiler for the machine.

As well as experimentation with the prototype, further experiments on architectural or implementation alternatives will be carried out by extending the SIM++ model of ESP developed by Goh Boon Seng. The aims of tests using the prototype and simulated model will include:

- to determine the efficiency of the sparse vector arithmetic operations in the context of the target applications, using a variety of real data of differing sparsity and structure, and to identify the performance limiting factors in the design;
- to experiment with architectural and implementation alternatives, and in particular to finalise the choice of Garbage Collection algorithm;
- to estimate the scalability of the architecture to faster technology, including the feasibility of use with a vector data cache.

If these tests indicate that the architecture performs well, further experiments on the machine, and investigation of other potential implementations would be

worthwhile. In particular, the performance on other sparse matrix applications should be tested – this would certainly require the development of a compiler – and preliminary work could be done on a custom silicon implementation of the architecture, to determine the cost of the special architectural features, the problems introduced by pin-out limitations, and the likely performance.

8.4 Conclusion

An architecture has been developed which appears to have the potential to provide, for very sparse matrix problems, the kind of speed-up associated with the standard vector processing technique for dense problems (a factor of ten over scalar implementations of the same problems). Whether the architecture could be as effective at clock speeds above 20MHz remains to be investigated. Development of a special-purpose computer architecture is risky – if the special architecture supports an order of magnitude performance increase, but the implementation takes several years to complete, then technological advances will have improved the speed of conventional architectures by the same order of magnitude, by the time the special-purpose machine is commissioned. (This effect was partly responsible for the commercial failure of the **STAR-100**, for example.) In addition, a special-purpose architecture can only succeed commercially if there is a sufficiently large market for the applications which it accelerates (examples are vector processors, digital signal processors, and graphics processors), or if the special features do not add too greatly to the cost of the complete machine, and so can be incorporated into machines destined for a wider market (examples include the provision of on-chip floating-point hardware in processors in computers which may never be used for intensive numerical applications, and scatter/gather hardware in vector processors).

Whether architectural features like those developed for ESP could ever be commercially viable remains to be seen. Notwithstanding this, it is hoped that the results of experimentation with ESP will also prove useful in guiding potential

implementations of sparse matrix applications on other newly emerging hardware platforms, such as superscalar and decoupled processor architectures.

Bibliography

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[2] Altera Corp., San Jose, California. *Altera Data Book*, 1991.

[3] H. Amano et al. $(SM)^2$: Sparse matrix solving machine. In *ACM Proc. 10th Symposium on Computer Architecture*, pages 213–220. ACM, 1983.

[4] H. Amano et al. $(SM)^2$-II : A new version of the sparse matrix solving machine. In *ACM Proc. 12th Symposium on Computer Architecture*, pages 100–107. ACM, 1985.

[5] H. Amano, B. Taisuke, and T. Kudoh. $(SM)^2$-II: A large-scale multiprocessor for sparse matrix calculations. *IEEE Transactions on Computers*, 39(7):889–905, July 1990.

[6] G. Amdahl. The validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Proc. Spring Joint Comput. Conf.*, 1967.

[7] T.W. Archibald, K.I.M. McKinnon, and L.C. Thomas. Parallel iterative algorithms and their application to Markov decision processes. In *Proceedings of the 1991 Edinburgh Workshop on Parallel Numerical Analysis*, 1991.

[8] J.H. Austin. The Burroughs Scientific Processor. In C.R. Jesshope and R.W. Hockney, editors, *Infotech State of the Art Report: Supercomputers, vol. 2*, pages 1–31. Infotech Int. Ltd., 1979.

[9] O. Axelsson. Conjugate gradient type methods for unsymmetric and inconsistent systems of linear equations. *Linear Algebra and Applications*, 29:1–66, 1980.

[10] W. Buchholz. The IBM System/370 vector architecture. *IBM Systems Journal*, 25:51–62, 1986.

[11] T.F. Coleman. *Large Sparse Numerical Optimization.* Lecture Notes in Computer Science. Springer Verlag, 1984.

[12] Control Data Corporation, St. Paul, Minnesota. *Control Data CYBER 200 Model 205 Computer System – Hardware Reference Manual*, 1981.

[13] Cray Research Inc., Mendota Heights, Minnesota. *CRAY X-MP Computer Systems: Mainframe Reference Manual*, 1982.

[14] T.A. Davis and E.S. Davidson. Pairwise reduction for the direct parallel solution of sparse unsymmetric sets of linear equations. *IEEE Transactions on Computers*, 37:1648–1654, 1988.

[15] I.S. Duff. Full matrix techniques in sparse Gaussian elimination. In *Proc. Dundee Biennial Conference on Numerical Analysis*, June 1981.

[16] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices.* Oxford Science Publications, 1986.

[17] I.S. Duff et al. Sparse matrix test problems. *ACM SIGNUM Newsletter*, 17(2):22, 1982.

[18] G. Feierbach and D. Stevenson. The ILLIAC IV. In C.R. Jesshope and R.W. Hockney, editors, *Infotech State of the Art Report: Supercomputers, vol. 2*, pages 77–92. Infotech Int. Ltd., 1979.

[19] J.J.H. Forrest and J.A. Tomlin. Vector processing in simplex and interior methods for Linear Programming. *Annals of Operations Research*, 22:71–100, 1990.

[20] Goh Boon Seng. Simulation of the Edinburgh Sparse Vector Processor. M.Sc. Project Report, Department of Computer Science, University of Edinburgh, September 1992.

[21] P.M.J. Harris. Pivot selection methods of the Devex LP code. *Math. Progr.*, 5:1–28, 1973.

[22] High Level Hardware Ltd., Headington, Oxford, UK. *ORION Time Sharing Manual*, 1986.

[23] W.D. Hillis. *The Connection Machine.* MIT Press, Cambridge, Mass., 1985.

[24] R.G. Hintz and Tate D.P. Control Data STAR-100 processor design. In *Proceedings of the IEEE COMPCON*, pages 1–4, New York, Sept 1972. IEEE.

[25] R.W. Hockney and C.R. Jesshope. *Parallel Computers 2: Architecture, Programming and Algorithms.* Adam Hilger, Bristol, England, second edition, 1988.

[26] R.N. Ibbett, P.C. Capon, and N.P. Topham. MU6V: A parallel vector processing system. In *Proceedings of the 12th ACM/IEEE International Symposium on Computer Architecture*, 1985.

[27] R.N. Ibbett and N.P. Topham. *The Architecture of High Performance Computers*, volume 2. Macmillan Press, second edition, 1989.

[28] R.N. Ibbett and N.P. Topham. *The Architecture of High Performance Computers*, volume 1. Macmillan Press, second edition, 1989.

[29] INMOS Ltd., Bristol, England. *IMST414 Transputer Reference Manual*, 1985.

[30] Jade Simulations International Corporation. *Sim++ Release 2.2 Programmer Reference Manual*, 1982.

[31] D.E. Knuth. *The Art of Computer Programming*, volume 1 - Fundamental Algorithms. Addison-Wesley, second edition, 1973.

[32] J.G. Lewis and H.D. Simon. The impact of hardware gather/scatter on sparse Gaussian elimination. Mathematics and Modelling Technical Report ETA-TR-33, Boeing Computer Services, Seattle, Washington, 1986.

[33] Q.I. Lin. *Design of a Vector Processor.* PhD thesis, Department of Computer Science, University of Manchester, 1983.

[34] J. Luo, F. Bruggeman, and G.L. Reijns. Revised simplex method on a network of T800 transputers. In A.S. Wagner, editor, *Transputer Research and Applications: Proceedings of the Third North American Transputer Users Conference*, pages 39–50, Sunnyvale, California, April 1990. IOS Press.

[35] A.G. Manning. Simulation of hardware mechanisms for sparse vector processing. 4th Year Project Report, Department of Computer Science, University of Edinburgh, June 1992.

[36] H.M. Markowitz. The elimination form of the inverse and its application to Linear Programming. *Management Science*, 3:255–269, 1957.

[37] K.I.M. McKinnon and H.P. Williams. Linear Programming and its suitability for processing by the DAP. Technical report, International Computers Ltd., 1980.

[38] J.T. O'Donnell. An efficient architecture for implementing sparse array variables. In *Proceedings of the 23rd Annual Allerton Conference on Communications, Control and Computing*, Urbana, Illinois, Oct 1985. Coordinated Science Laboratory, University of Illinois.

[39] C.E. Pfefferkorn and J.A. Tomlin. Design of a Linear Programming system for the ILLIAC IV. Technical Report SOL 76-8, Dept. of Operations Research, Stanford University, 1976.

[40] S.F. Reddaway. The DAP approach. In C.R. Jesshope and R.W. Hockney, editors, *Infotech State of the Art Report: Supercomputers, vol. 2*, pages 311–329. Infotech Int. Ltd., 1979.

[41] R. Rettberg and R. Thomas. Contention is no obstacle to shared-memory multiprocessing. *CACM*, 29(12):1202–1212, Dec 1986.

[42] P.S. Robertson. The IMP-77 language. Technical Report CSR-19-77, Department of Computer Science, University of Edinburgh, 1983.

[43] R.M. Russell. The CRAY-1 computer system. *CACM*, 21:63–72, 1978.

[44] J.E. Smith. Decoupled access/execute computer architectures. *ACM Transactions on Computer Systems*, 2(4):289–308, Nov 1984.

[45] C.B. Stunkel. Linear optimization via message-based parallel processing. *ICPP*, 3:264–271, 1988.

[46] Texas Instruments, Houston, Texas. *MOS Memory Data Book*, 1989.

[47] N.P. Topham. *A Parallel Vector Processing System.* PhD thesis, Department of Computer Science, University of Manchester, 1985.

[48] Weitek Corporation, Sunnyvale, California. *WTL 3164/XL-3164 WTL 3364/XL-3364 64-bit Floating-Point Data Path Units - Advance Data*, April 1988.

[49] Weitek Corporation, Sunnyvale, California. *XL-8136 32-bit Program Sequencing Unit Data Sheet*, April 1988.

[50] Weitek Corporation, Sunnyvale, California. *XL-8137 32-bit Integer Processing Unit Data Sheet*, April 1988.

[51] Z. Zlatev. *Computational Methods for General Sparse Matrices.* Kluwer Academic Press, 1991.

Appendix A

The Vector Processor Instruction Set

Vector instructions consist of a list of 32-bit words, illustrated in figure A–1. The control word, the **index.count** word, and the **instruction.identifier** word are always present; the remaining words are optional - their presence is determined by the relevant field in the control word, described below.

The control word comprises separate fields to control each of the following vector pipeline parts: the *Vector Read (VR)* circuit, the three parts of the Arithmetic Section of the pipeline, namely the *Index Match (IM)* circuit, the *Arithmetic Unit (AU)*, the *Vector Output (VO)* circuit, and the *Vector Write (VW)* circuit. The Vector Write instruction field also controls the operation of the Garbage Collection circuit and the Sideways List Unit. Each field is specified to the assembler by a mnemonic, with the fields separated by commas. If any field is left null or blank, a *no-operation* is assumed for the associated pipeline part (*ie* input is discarded, and null output is produced). In building up a complete control word, it is necessary to ensure that the operand stream passed on by each pipeline part is compatible with the instruction selected for the following pipeline part; however, there remains an enormous number of allowable combinations of instructions for the individual pipeline parts, not all of which are useful. To aid programming, the assembler allows predefinition of more meaningful mnemonics for the complete control word of commonly used vector instructions.

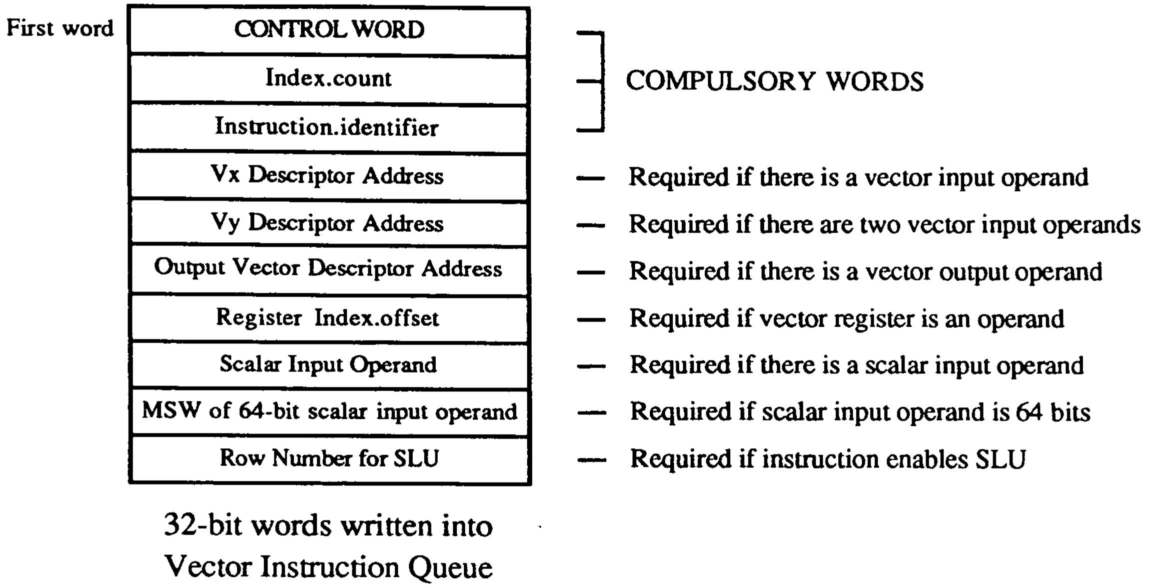

Figure A–1: The format of ESP vector instructions

A.1 Vector Read circuit instructions

A.1.1 Two operand output modes

Two descriptor addresses must be specified in the instruction. The two specified vectors are known as Vx and Vy. Two streams of I/V pairs are sent to AS, on the X and Y data paths. The last element of any stream sent is always an end-of-vector marker (but when no stream is generated, *ie* on the Y path for single operand output modes, and on both the X and Y paths for mode N0, no end-of-vector marker is sent either). The following is a list of the mnemonics for the different VR modes, and the corresponding VR function:

LL2 Vx and Vy are both list vectors. For each, VR reads in index/value pairs – pairs from Vx go to the X stream, pairs from Vy to Y.

AA2 Vx and Vy are both array vectors. VR reads index/value pairs from each. Vx feeds X and Vy feeds Y.

AL2 Vx is of type array, Vy is of type list. Vx is accessed as in AA2 above, Vy as in LL2. Vx feeds X and Vy feeds Y.

AiL2 Vx is of type array, Vy is of type list. Vy is accessed as in mode LL2. The index of each index/value pair in Vy is used to index into Vx and the corresponding index/value element of Vx is fetched. The Vx index/value pairs are fed to AS on X, and the Vy index/value pairs on Y.

A.1.2 Single operand output modes

In these modes, only a single index/value stream is passed to AS, on data path X. Only the Vx descriptor address need be specified.

L1 Vx is a list vector, accessed as mode LL2 above. A stream of index/value pairs is passed to AS on X.

A1 Vx is a full vector, accessed as mode AA2 above. A stream of index/value pairs is passed to AS on X.

A.1.3 Null mode

Neither Vx nor Vy is specified. VR passes nothing to AS.

N0 VR does nothing.

A.1.4 Instruction termination

Termination of the VR instruction occurs when vector input is exhausted (*ie* the end-of-vector elements have been passed to AS).

A.2 Index Match circuit instructions

The Index Match circuit, which is the first part of the Arithmetic Section, receives from VR two streams of index/value pairs (one on the X data path, one on the Y data path), or one stream of index value pairs (on the X data path), or nothing at all. It passes into the Arithmetic Unit input queue a stream of index/value/value triples, or of index/value pairs with the values on the $V\alpha$ data path, or a stream of indices only. The last data item passed is always an end-of-vector marker.

A.2.1 I/V/V triple output modes

The IM instruction mnemonics and corresponding functions are:

***2** IM expects two I/V streams from VR. It passes as I/V/V triples to AU only elements which appear in both I/V streams. I/V pairs whose indices appear in one input stream only are deleted. X values go to $V\alpha$, Y to $V\beta$.

+2 IM expects two I/V streams from VR. It passes an I/V/V triple to AU for every distinct index appearing in either I/V stream. If the index appears in one stream only, IM generates a zero value to complete the triple. X values go to $V\alpha$, Y to $V\beta$.

++2 IM expects two I/V streams from VR. It passes an I/V/V triple to AU for every index from 1 to index.count, even if the index appears in neither I/V stream. To do this it generates one or two zeroes to insert as appropriate in I/V/V triples. X values go to $V\alpha$, Y to $V\beta$.

Ri2 IM expects one I/V stream from VR, on X. It uses the indices in that stream to look up in the vector register (first adding the Register Index.offset specified in the instruction) and passes I/V/V triples to AU consisting of the index and value from the X stream (the value on the $V\alpha$ data path), and the corresponding register value (on the $V\beta$ data path).

++Ri2 As Ri2 above, except that before the lookup in the register is done, IM fills out the incoming I/V stream so all indices from 1 to index.count are explicitly present.

Ri2K As Ri2 above, except that two I/V streams are expected from VR – the Y stream is discarded (for use with VR mode AiL2).

++Ri2K As ++Ri2 above, except that two I/V streams are expected from VR – the Y stream is discarded (for use with VR mode AiL2).

A.2.2 I/V pair output modes

IM passes a single I/V stream to AU, with the values on the $V\alpha$ data path. The mnemonics and functions are:

1 A single I/V stream is expected from VR, on X. IM simply passes this stream through to AU.

1K As 1 above, but two I/V streams are expected from VR – the Y stream is discarded (for use with VR mode AiL2).

++1, ++1K As 1 and 1K above, but IM fills the stream out by inserting explicit zeroes, so all indices from 1 to index.count are explicitly present.

Ri1 A single I/V stream is expected from VR, on X. IM uses the indices from the X stream, and discards the values. It uses the indices to look up into the vector register (first adding the Register Index.offset specified in the instruction), and passes the generated index/value pairs to AU.

R1 IM expects nothing from VR. It generates indices by counting from 1 to index.bound, adds the Register Index.offset specified in the instruction to look up in the vector register to get the corresponding values, and passes the resulting index/value pairs to AU (the indices passed are the ones generated *before* addition of the Register index.offset).

Index only output mode

In this mode, IM passes indices only to AU.

IO IM expects nothing from VR. It generates a stream of indices by counting from 1 to index.count, and passes these to AU, followed by an end-of-vector marker.

A.2.3 Instruction termination

The IM is part of the Arithmetic Section (AS), and the instruction terminates when execution is complete in all parts of the AS (IM, AU and VO). IM is finished when the end-of-vector marker has been written to the AU input queue.

A.3 Arithmetic Unit instructions

The Arithmetic Unit receives from IM, via its input queue, a stream of I/V/V triples, a stream of I/V pairs (with the values on the $V\alpha$ data path), or a stream of indices only. Depending on the instruction, it may pass to VO a stream of I/V pairs, into the VO input queue, and may also pass a scalar output value to the Scalar Result Queue. If an I/V stream is output, it will end with an end-of-vector marker.

Scalar input operands, which may be arithmetic or logical data values, are specified in the vector instruction in one or two 32-bit words. Scalar output operands are single arithmetic or logical data values, and are passed to the Scalar Result Queue. Some instructions (searching operations) produce as output only a *single* I/V pair – this is passed to the Scalar Result Queue, not VO.

The Arithmetic Unit operation is specified by a function field, a subfunction field (only relevant for some values of the function field) and a type field, all in the vector instruction control word. The type field may take the values 32-bit IEEE floating point, 64-bit IEEE floating point, 32-bit integer, or 32-bit logical.

The instructions are listed below in four sets, *data movement* operations, *arithmetic* operations, *logical* operations, and *special* operations.

A.3.1 Data movement operations

MOVE The AU expects an I/V stream from IM, and passes this through to VO, unchanged. This mode is used, for example, in instructions which change the format of a vector from list to array. The type and subfunction fields are irrelevant here.

FILL The AU expects a stream of indices from IM, and uses the supplied scalar input operand as the value in I/V pairs passed to VO. The type field may be any of the four types, and the scalar input operand will be of the same type; the subfunction field is irrelevant.

INCFILL As FILL, except that the scalar value is incremented every time it is used. The type must be 32-bit integer; the subfunction field is irrelevant.

SELECT The AU expects an I/V stream from IM. It tests each value against the scalar input operand S, deleting I/V pairs which fail the test, and forwarding those which pass to VO. The subfunction field specifies the test. The type field may be any of the four types, but there are restrictions on the type/subfunction combination. The valid subfunctions are:

000 Pass pairs where $V = S$

001 Pass pairs where $V \neq S$

010 Pass pairs where $V > S$ (not logical type)

011 Pass pairs where $V \geq S$ (not logical type)

100 Pass pairs where $V < S$ (not logical type)

101 Pass pairs where $V \leq S$ (not logical type)

FIRSTSELECT As SELECT, but the instruction terminates after generating just one I/V pair which passes the test, which is passed to the Scalar Result Queue.

SEARCH The AU expects an I/V stream from IM. It generates a single I/V pair as output - the pair with the most extreme value, as defined by the subfunction. This pair is passed to the Scalar Result Queue. The type field may be any arithmetic type. The valid subfunctions are:

000 Find the most positive value

001 Find the largest absolute value

010 Find the smallest absolute value

011 Find the most negative value

Note that in the event of two or more I/V pairs containing the extreme value, the first occurrence of the extreme value in the input stream is the one returned.

RELATIVESELECT As SELECT, but the AU expects I/V/V triples as input, and compares the values pairwise in the way specified by the subfunction field, rather than comparing against a single scalar. Indices and $V\alpha$ values from triples passing the test are forwarded to VO. Again, the type may be any of the four, but there are restrictions on the type/subfunction combination. The subfunctions are as SELECT, except that the value field $V\beta$ replaces S.

A.3.2 Arithmetic operations

ADD The AU expects I/V pairs or I/V/V triples from IM, and adds the values in the way specified by the subfunction, to produce I/V pairs. The type may be any arithmetic type. The valid subfunctions are:

000 The AU expects I/V pairs from IM, and adds the scalar input operand to each value.

001 The AU expects I/V/V triples from IM, and adds the two V fields.

010 The AU expects I/V/V triples from IM, and produces output values equal to $V\alpha + (V\beta * S)$ where S is the scalar input operand.

SUBTRACT The AU expects I/V pairs or I/V/V triples from IM, and performs the operation specified in the subfunction, to produce a stream of I/V pairs. The type may be any arithmetic type. The valid subfunctions are:

000 The AU expects I/V pairs from IM, and subtracts the scalar input operand from each value.

001 The AU expects I/V/V triples from IM, and outputs values equal to $V\alpha - V\beta$.

010 The AU expects I/V/V triples from IM, and outputs values equal to $V\alpha - (V\beta * S)$, where S is the scalar input operand.

011 The AU expects I/V pairs from IM and subtracts the values from the scalar input operand (*ie* the output values are the negative of those from subfunction **000**).

100 As subfunction **001**, except that the values are forced to positive sign before output.

REVERSESUBTRACT As SUBTRACT, but the $V\alpha$ and $V\beta$ inputs are exchanged in the operation.

MULTIPLY The AU expects to receive I/V pairs or I/V/V triples from IM, and outputs I/V pairs to VO. Any arithmetic type is allowed. The valid subfunctions are:

000 The AU expects I/V pairs from IM, and multiplies each value by the scalar input operand before outputting it.

001 The AU expects I/V/V triples from IM, and multiples the $V\alpha$ and $V\beta$ values together to form the output values.

010 The AU expects I/V/V triples from IM, and forms output values from $V\alpha * (V\beta + S)$, where S is the scalar input operand.

DIVIDE The AU expects to receive I/V pairs or I/V/V triples from IM, and outputs I/V pairs to VO. Any arithmetic type is allowed. The valid subfunctions are:

000 The AU expects I/V pairs from IM, and divides each value by the scalar input operand before outputting it to VO.

001 The AU expects I/V/V triples from IM, and divides each $V\alpha$ value by the corresponding $V\beta$ value before outputting it.

010 The AU expects I/V/V triples from IM, and outputs values equal to $V\alpha/(V\beta + S)$ where S is the scalar input operand.

011 The AU expects I/V/V triples from IM, and outputs values equal to $V\alpha/V\beta$ if $V\beta > 0$, or equal to the scalar input operand S if $V\beta \leq 0$. (This is useful in some Linear Programming algorithms).

REVERSEDIVIDE As DIVIDE, except that the $V\alpha$ and $V\beta$ values are exchanged in the definition of the operation.

ABSOLUTE The AU expects I/V pairs from IM, and converts each value to its absolute value before outputting it.

ADDUP The AU expects I/V pairs or I/V/V triples from IM. No I/V output stream is produced, but a single scalar output operand is passed to the Scalar Result Queue. The type can be any arithmetic type. The valid subfunctions are:

000 The AU expects I/V pairs from IM, and adds up the values.

001 The AU expects I/V pairs from IM, and adds the squares of the values.

010 the AU expects I/V/V triples from IM, and adds the products $V\alpha * V\beta$ (scalar product).

MULTIPLYUP The AU expects I/V pairs from IM. No I/V output stream is produced, but a single scalar output value is passed to the Scalar Result Queue. That value is the product of all the values in the input stream. The type may be any arithmetic type; the subfunction field is ignored.

A.3.3 Logical operations

LOGICAL The AU expects I/V pairs or I/V/V triples from IM, according to the subfunction. The type must be logical. The allowed subfunctions are:

- **000** The AU expects I/V pairs from IM, and **AND**s each value with the scalar input operand.
- **001** The AU expects I/V/V triples from IM, and **AND**s the $V\alpha$ and $V\beta$ values to produce output values.
- **010** As subfunction **000**, but **OR**.
- **011** As subfunction **001**, but **OR**.
- **100** As subfunction **000**, but **EXOR**.
- **101** As subfunction **001**, but **EXOR**.
- **110** The AU expects I/V pairs from IM, and logically inverts each value before output.

SHIFT The AU expects I/V pairs from IM, and outputs I/V pairs according to the subfunction:

- **000** Values are shifted one place left, with 0 shifted into LSB. (Logical type only).
- **001** Values are shifted one place right, with 0 shifted into MSB. (Logical type only).
- **010** Values are shifted one place left, except that the MSB is unaltered ('arithemtic shift left'). Zero is shifted into the LSB. (32-bit integer type only).
- **011** Values are shifted one place right, and the MSB is unaltered ('arithmetic shift right'). (32-bit integer type only).

LOGICALUP The AU expects I/V pairs from IM. No I/V output stream is produced, but a single scalar output value is passed to the Scalar Result Queue. The type must be logical, and the subfunction field determines the operation, as follows:

000 The output value is formed by **AND**ing together all the input values.

001 The output value is formed by **OR**ing together all the input values.

010 The output value is formed by **EXOR**ing together all the input values.

A.3.4 Special operations

BTRAN The AU expects I/V/V triples from IM. It produces a single I/V pair, which is passed to VO, followed by an end-of-vector element. It adds up the products $V\alpha * V\beta$ to form the value field of the result pair, while the index field of the result is given by the least significant 16 bits of the scalar input operand S.

FTRAN The AU expects I/V/V triples from IM, and outputs I/V pairs to VO. It reads the value R_S at the vector register location whose index is given by the least significant 16 bits of the scalar input operand S plus the Register index.offset, and then outputs the values $V\alpha * R_S + V\beta$

A.3.5 Instruction termination

Arithmetic exceptions occurring during instruction execution will not cause the instruction to terminate; however, they will set a bit in the Vector Flag Exception Register.

The AU is part of the Arithmetic Section – the AS instruction terminates when all of IM, AU and VO are finished. AU is finished when all the input values have been read (including the end-of-vector marker) and processed, and all output values (including the end-of-vector marker where appropriate) have been written into the VO input queue or the Scalar Request Queue.

A.4 Vector Output circuit instructions

The Vector Output circuit receives from AU a stream of I/V pairs, which may be empty, followed by an end-of-vector marker. It may forward these to VW, by passing them to the Z data queue followed by an end-of-vector marker, or it may write values to the vector register.

The VO instruction mnemonic consists of any combination of of the symbols **D**, **R**, and **M**, which enable, respectively, dropping of elements whose exponent is less than that in the drop register (floating-point types only), writing to the vector register, and forwarding to VW. If neither **R** nor **M** is specified, no output vector is produced.

When writing to the vector register is enabled, the Register Index.offset specified in the instruction is added to the incoming indices, before they are used to address the vector register.

A.4.1 Instruction termination

The VO is part of the Arithmetic Section, and the AS instruction terminates when all of the IM, AU and VO are finished. VO is finished when the end-of-vector marker has been read from its input queue, and the end-of-vector marker has been written to the Z data queue (if mode **M** is specified), and the last element has been written to the register (if mode **R** is specified).

A.5 Vector Write circuit instructions

The Vector Write instruction consists of two parts, one to control the operation of the Vector Write circuit and the Garbage Collector, the other to control the operation of the Sideways List Unit.

The Vector Write circuit receives a stream of I/V pairs from VO, into the Z data queue, writes them to vector memory, and optionally updates the Sideways List Unit memory. If the VW mode is L, A, Ai or G, a vector output operand must be specified in the instruction. VW is responsible for accessing descriptor memory to read the contents of the output operand descriptor for an array vector, at the start of the instruction. It is also responsible for updating the start address and non-zero count fields of the descriptor at the end of the instruction, if the output operand is a list vector.

The instruction mnemonics and functions are as follows:

L The I/V pairs are written to memory as a list vector (into space taken from the front of the free list). At the end, the vector descriptor is updated to contain the correct start address, and number of non-zeroes is also updated. At the end of the operation, the old start address of the output vector is passed to the Garbage Collector, so that the space may be reclaimed.

A The I/V pairs are written to an array vector in memory. This VW mode should only be used with VR and IS modes which ensure all indices are explicitly present.

Ai The index field of each incoming I/V pair is added to the output vector start.address, and the index/value pair is written to the resulting location.

G Do not write data to memory (VO instruction must not specify mode M). Do pass the start address specified in the descriptor for the vector output operand to the Garbage Collector, so the space may be reclaimed. Used to get rid of unwanted list vectors.

N No-Op.

A.5.1 Sideways List Unit instructions

The Sideways List Unit receives a stream of indices from VO (the same stream that is passed, with associated values, to VW). The Sideways List Unit has four mutually exclusive operating modes, with the following mnemonics:

F Save the position of flagged new non-zeroes (using the row.number specified in the instruction) in the relevant lists, specified by the incoming indices.

A Save the position of all non-zeroes (the row.number) in the relevant lists, specified by the incoming indices.

Z Zero the count and address registers, and set the memory address register to 1 (used for initialisation).

N No-Op.

If modes **F** or **A** are specified, the instruction must contain a **row.number** word.

A.5.2 Instruction termination

The Vector Write section operation terminates when the end-of-vector marker, plus any padding elements required, have been written to memory, the Sideways List Unit updates are complete, the output vector descriptor has been updated if required, and, if relevant, the garbage address has been passed to the Garbage Collector. Note that the Garbage Collector may not merge the garbage with the free list until later. When the VW instruction terminates, the VW section control logic writes the instruction.identifier word of the vector instruction into the Instruction Identifier Register.

Appendix B

Reprint of published paper

The following paper was presented by the author at the ACM/IEEE International Symposium on Computer Architecture, Jerusalem, May/June 1989.

(Copyright on this paper has been transferred to the Association for Computing Machinery, which has granted to the authors the right to republish without specific permission.)

Architectural Mechanisms To Support Sparse Vector Processing

R.N. Ibbett, T.M. Hopkins and K.I.M. McKinnon

Departments of Computer Science and Mathematics
University of Edinburgh
James Clerk Maxwell Building, King's Buildings
Mayfield Road, Edinburgh, EH9 3JZ

Abstract

We discuss the algorithmic steps involved in common sparse matrix problems, with particular emphasis on linear programming by the revised simplex method. We then propose new architectural mechanisms which are being built into an experimental machine, the Edinburgh Sparse Processor, and which enable vector instructions to operate efficiently on sparse vectors stored in compressed form. Finally, we review the use of these new mechanisms on the linear programming problem.

1 Introduction

Sparse vectors are an important feature of a number of computer applications. Their distinguishing characteristic is the occurrence of large numbers of zero elements in vectors and arrays, and in mapping sparse applications on to existing computers a variety of software techniques have been employed to reduce the storage and processing required for the zero elements. Most sparse codes run on *scalar* machines, or the scalar processors of vector computers, and make no use of standard vector processing facilities. A few computers, notably the CDC CYBER 205, have provided architectural support for sparse vectors in the form of addressing modes and special orders [7], but these orders have proved difficult to use in practice, mainly because of the *fill-in* problem. Fill-in occurs when, for example, two sparse vectors are added and the positions of the non-zero elements in the two vectors do not match, so that the result vector contains more elements than either of the source vectors. Since the extent of fill-in cannot be predicted at compile time, the compiler cannot know how much space to allocate to sparse vectors which are created during the running of a program.

2 Sparse Matrix Computation

Computation with sparse matrices whose pattern of sparsity is regular (*eg* matrices arising from partial differential equation solution by finite difference or finite element methods) can often be carried out efficiently on standard vector processor machines. Computation on matrices with irregular sparsity pattern is not amenable to these techniques, and so it is problems of this nature in particular that the proposed new architectural mechanisms address.

Irregular sparsity patterns arise from irregularity in the real-world problem being modelled, typical examples being found in engineering design simulations of physical structures or electrical circuits, and in Linear Programming (LP) problems. As examples of the type of vector and matrix calculation steps a sparse vector processor must support, we examine the steps of the *Product Form of Inverse (PFI) simplex* algorithm for Linear Programming, which include (in the re-invert step), the more general problem of the direct solution of a system of sparse linear equations.

2.1 Linear Programming

LP problems are typically very sparse. A large LP problem might involve a matrix of several thousand rows and 3 times as many columns, but with only 6 or so non-zeroes in each column. The four most computationally intensive steps of the PFI simplex algorithm are the so-called **BTRAN**, **pricing**, **FTRAN**, and **re-invert** steps, which typically take 30%, 35%, 20%, and 7% respectively of the total solution time. In the PFI method, the current inverse of the basis matrix is held as a series of *PFI matrices*, of special form, the product of which is the basis inverse. Each pivot step in the solution adds another PFI matrix to the series, so the series will usually contain many hundreds of matrices. Each of these has the same number of rows as the original problem matrix, and has the form of the unit matrix plus a single non-zero column, which may typically have a density between 1% and 20%. For each PFI matrix, only the non-zero column, plus a record of its position in the matrix, need be stored; these column vectors are generally known as the η (eta-) vectors.

2.1.1 BTRAN

This involves the post-multiplication of a vector by each of the PFI matrices in turn. Because of the special form of the PFI matrices, each multiplication step reduces to the replacement of one element of the vector being updated with the scalar product of that vector and the η vector of the relevant PFI matrix:

$$v_k \leftarrow \mathbf{v}.\eta$$

where k is the column position of the η vector in its PFI matrix.

2.1.2 Pricing

The result of the BTRAN operations is a *price vector*, and the pricing step involves the formation of the scalar product of this vector with each of the columns of the original problem matrix. The price vector will typically be 40% dense, while the original columns are very sparse - less than 0.1%.

2.1.3 FTRAN

This step involves the pre-multiplication of a vector (a column of the original problem matrix) by each of the PFI matrices in turn. Each multiplication step reduces to the summation of the vector with the relevant η vector, first scaled by a single element of the vector being updated:

$$\mathbf{v} \leftarrow \mathbf{v} + v_k * \eta$$

As above, k is the column position of η in the PFI matrix.

The vector being updated starts very sparse ($< 0.1\%$) and typically reaches 20% density at the end.

2.1.4 Re-invert

For reasons of numerical stability, it is necessary from time to time during the solution to replace the accumulated series of PFI matrices with an equivalent series of matrices of the same form derived by direct Gaussian Elimination on the current basis matrix. That matrix is a portion of the original problem matrix, and is therefore less than 0.1% dense. Each pivot step in the elimination will consist of the subtraction of a multiple of the pivot row from rows of the (updated) matrix with a non-zero in the pivot column:

$$B_{n*} \leftarrow B_{n*} - S * B_{p*}$$

where B_{n*} is a row of the matrix B with a non-zero in the pivot column, B_{p*} is the pivot row, and S is a scalar equal to the element of B_{n*} in the pivot column, divided by the pivot element.

Here, both vectors are of roughly equal sparsity (although if the Markowitz criterion *(see below)* is used to choose the pivot, the pivot row will tend to be somewhat sparser than the other). When the matrix is very sparse, the subtraction is fast, and it is crucial that one can rapidly determine which rows have a non-zero in the pivot column, and access the value of that non-zero, or this operation will dominate.

As the elimination proceeds, the part of the matrix remaining to be eliminated becomes more dense. This can lead to a drastic increase in the number of floating point operations required, both to complete the elimination, and in the remainder of the LP solution. It is therefore vital to minimize fill-in by careful choice of pivot at each step. However, except in the special case of a matrix which is symmetric positive definite, it is also necessary to consider numerical stability in choosing the pivots. The usual compromise is known as *threshold pivoting* [2]. A potential pivot is chosen by selecting an element whose row and column are both very sparse (the *Markowitz* criterion [9]), but the potential pivot is rejected if it is too much smaller than the largest element in its column. Operation of this method requires knowledge at each pivot step of the number of non-zeroes in each row and column of the updated matrix. It also requires that the each non-zero in the proposed pivot column be accessible in reasonable time.

3 ESP Sparse Vector Mechanisms

ESP is a vector processing system consisting of a scalar control processor and a separate vector processing pipeline, in a manner similar to the CYBER 205. However, ESP is not designed as a stand-alone computer, but rather as a high-performance back-end co-processor which executes routines to complete critical core sections of numerical programs, under control of a main program running on the host computer. These main programs may be compiled on the host, and a range of optimised ESP routines will be available as a library. The prototype is being designed as a co-processor for an ORION [6] minicomputer, but it is intended that the design be easily adaptable to operate with commonly available workstations as host machines.

The routines executed by the ESP co-processor consist of a mix of vector instructions, and more conventional scalar and control instructions. A typical vector instruction involves two source vectors and one destination vector. As in the CYBER 205 [7], each vector is accessed via a descriptor, but in ESP the descriptors contain more information about the vector, and are more reminiscent of those used in MU5 [10]. In order to support the kinds of operation required for efficient execution of LP and other sparse problems, two different storage mechanisms for vectors are provided in ESP and many instructions do not specify the storage mechanism used for their operands (this information is in the descriptor) but will work on vectors stored in either form. The descriptor also contains the necessary information for the storage controller to find the vector elements, and information on the type of the vector elements (single or double precision etc.).

The first storage mechanism is the full vector. Here a vector is stored in standard form as a sequence of values in store, so that an n-element vector occupies n storage locations, and the element of a vector with a specified index is found by offsetting from the vector base address specified in the descriptor. While efficient for access to any element from its index, this mechanism does not support fast access to the non-zero elements only, and of course wastes enormous amounts of store if vectors are very sparse. For sparse vectors, the list vector storage mechanism is used.

3.1 The List Vector Mechanism

3.1.1 Linked List Methods - General Strategy

Storing a sparse vector as a list of non-zero values, plus a corresponding list of the indices of the non-zeroes within the vector is attractive, as it incurs *no* storage or access overhead for the zeroes, and thus remains efficient even for very sparse vectors. Standard vector/vector operations, such as add, subtract and scalar product, can be performed directly on vectors stored in this form if *both* the index list and the value list of each operand are simultaneously accessible by the processor, with the lists maintained in order of ascending index.

This ordering of the lists is necessary for efficient implementation of vector/vector operations such as $C \leftarrow A + B$. These are implemented by streaming the lists A and B into the vector processor's arithmetic unit (AU), using additional hardware in the AU input stage to align A and B elements with equal indices. Thus in the example of the add instruction, illustrated in figure 1, if the first index in the A list is n, and that in the B list is m, then if $n < m$ the first element output is the first index/value pair from A. This element is discarded from the list A, while the list B is unchanged. Similarly for $m < n$, while if $n = m$, the output element is the sum of the elements at the front of the A and B queues, both of which are then discarded. Input vector index matching is also required for vector/vector multiplication

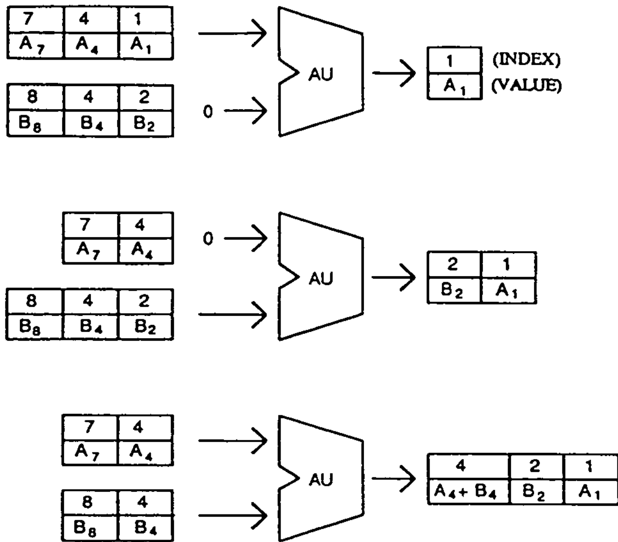

Figure 1: An ADD operation on list vectors

operations, but in this case a multiply step is only required if non-zeroes appear at the same index position in *both* input streams. If either or both of the vector lists were not in ascending index order, vector/vector arithmetic would involve searching for matching indices, and would be very inefficient.

As has already been noted, the amount of space needed to store a sparse vector in compressed form is not usually known at compile time. Nor is it in general known at run-time, even at the start of the operation producing the vector. In the add example above, the output list C may be as short as the longer input list or as long as the sum of the input list lengths, depending on the extent to which the postions of non-zeroes in the input vectors coincide. In some operations (*eg* the run-time compression of a vector from full storage format to list format) the resulting list length may be anywhere within much wider bounds. For the list storage format to be useful therefore, the amount of memory space allocated to a vector must be dynamically and automatically variable. The hardware must maintain a pool of free memory space, allocating extra space from the pool to vectors as required. Because it is impossible tell at the start of a vector operation how long the result list will be, it is not feasible to allocate at the start a single free block of store guaranteed to be large enough to hold the result. The solution adopted is to allow the index/value list comprising a single vector to reside in one or more *linked blocks* of memory locations. If the block allocated to hold the result at the start of an operation turns out to be too small, another block can be linked onto it. The unused portion of the last block allocated may be returned to the free pool.

Space may be freed by explicit de-allocation (under program control) of the memory used by temporary vectors which are no longer required. However, more often, it will become free automatically. To see why this is so, consider an operation of the type $A \leftarrow A + B$, and suppose vector A has non-zeroes at index positions 100 - 199, while B has non-zeroes at index positions 0 - 99. If the result vector is written into the memory blocks already occupied by A, then the first 100 output elements will overwrite the 100 non-zeroes of the original A. The first few of these will be in the arithmetic unit input pipe, but most will not yet have been read from memory, as the AU must deal with all 100 non-zeroes of B before using up any from A. As a result, most of the required input elements will have been corrupted before they are read. To avoid this, when a vector appears as both an input and result of an operation, the result vector must be allocated new space, and the original space used by that vector automatically returned to the free pool at the end of the operation.

3.1.2 Implementing Linked Lists in a Single-level Memory Environment

These list structures can be implemented in a simple way by treating memory as a pool of fixed size (small) blocks, each with a single link field. The free space is a linked list of unused blocks. As a vector operation produces its result, that result is written into the first locations on the free list, following links as required. The result vector's descriptor is updated to hold a pointer to the start of the vector list. The unused part of the final block in the result vector is wasted; this is the reason for small blocks. Reclaimed space is linked onto the start or end of the free list. The hardware required to support these operations is simple, and the operations of allocating new space and reclaiming old space are both very fast, each requiring only the updating of processor pointer registers, and the alteration of two links in memory.

Matrix codes tend to use many operations of the type $A \leftarrow B$ op A. For example, every elimination step in Gaussian Elimination causes a vector to be re-written, usually with a small increase in density, and for reasons in explained section 3.1.1 above, these re-writing steps involve the allocation of new space for the updated vector, and the reclaiming of space previously used. As space from vectors is reclaimed and later used again by vectors of different length, the blocks on the free list will become thoroughly mixed. As a result, the blocks used to store any vector will be randomly distributed throughout the whole memory space. In a single-level memory environment this may not matter, but in a hierarchical memory environment it is very likely to lead to thrashing of the paging/cacheing system. We believe that to restrict ESP to problems which will fit into a restricted single-level memory space (or which can be explicitly partitioned into smaller problems which fit) would be a mistake. We have therefore rejected this simple implementation in favour of mechanisms which retain more locality of reference.

3.1.3 Implementing Linked Lists in a Hierarchical Memory Environment

In this implementation, the free list remains a linked list of blocks of free memory, and allocation of new space for a result vector proceeds as above, except that blocks may now be of any length. Any space in the last block allocated (to a result vector) which remains unused at the end of the operation is left, as a smaller block, on the front of the free list. The key to maximising locality of reference lies in ensuring that the blocks on the free list remain as large as possible, so that the space allocated to a new vector consists of a small number of large blocks; this requires a more complex de-allocation algorithm. In general, a vector to be de-allocated itself consists of a list of blocks, and the de-allocation algorithm must check, for each of these blocks, whether is is *adjacent in memory* to a block (or blocks) already on the free list, and if it is, must merge the blocks. A simple way of achieving this merging de-allocation is to maintain the blocks in the vector lists and in the free list in order of ascending memory address (*ie* links from block to block are always *forward* through the memory address space). The de-allocation algorithm may then merge the two sorted lists of blocks in a straightforward way.

In ESP, this mechanism is supported by hardware interposed

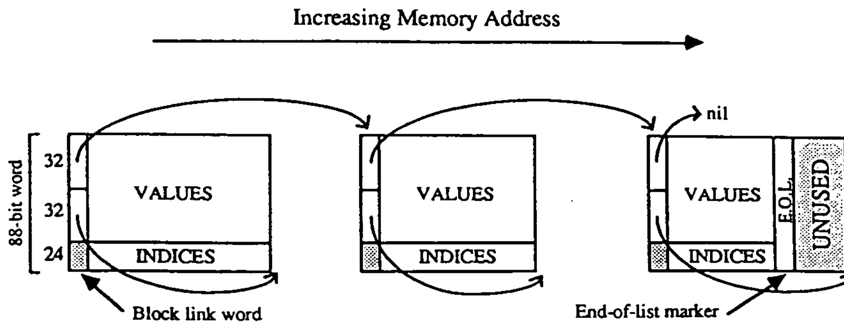

Figure 2: Structure of linked list vector

between the arithmetic unit and the memory. Vector elements are held in memory as an index/value pair, in a single memory word of 88 bits (64 bits for the value, and at 24 for the index). In addition to straight index/value pairs, memory words may hold an *end-of-list marker*, or they may hold *block link* words. A list vector is held in a series of blocks of consecutive memory words, the final word in the list being an end-of-list marker (see Fig. 2). The first word of each block is a block link word, and these words contain *two* pointers, each of which is a 32 bit (or larger) virtual memory address, known as the *external* pointer and the *internal* pointer. The external pointer holds the address of the first word (*ie* the word containing the block link) of the next block in the list, and is there to maintain the list linkage. The external pointer in the last block of a list holds the special value nil. The internal pointer holds the address of the first word *after* the end of the current block, and is there because the de-allocation algorithm needs to know the size of blocks to perform concatenation of adjacent blocks. Because the blocks are in order of increasing memory address, all pointers point forwards. Vector descriptors contain, in addition to other information about vector type and size, a pointer to the current position of the first word of the first block in the vector list. The free list is of identical structure, and a pointer to the start of it is maintained in a register.

As the AU produces index/value pairs as the results of a vector operation, these are written into the free list. The hardware for performing this contains a small number of pointer registers to keep track of the position in the free list currently being written to, and the block linkage. Eventually, the AU will signal the end of the result vector, by producing an element with the form of the special end-of-list marker. This is written out in the normal way, and a check is performed to determine how much free space remains in the block currently being written to. If this space is less than the minimum block size, it is left on the end of the result vector, otherwise it is reclaimed (by writing new block link words), and left at the front of the free list.

The de-allocation hardware, which reclaims vector space for the free list, requires two pointer registers (A and B), plus a small number of working registers. The algorithm merges two ordered lists of blocks - the free list, and the vector list being de-allocated. At the start, the free list pointer register is updated to point to the lower of the two list starting addresses. A also points to this address; B points to the other list. During the algorithm, the pointers A and B proceed along the two lists. The external pointers in the block link words are adjusted to merge the two lists, while the internal pointers are examined to check for contiguous blocks, and updated to merge such blocks. The algorithm terminates when one list is exhausted, and at this point, all blocks are linked, in order of increasing memory address, onto the list pointed to by the free list pointer. The maximum number of algorithm iterations will equal the total number of blocks on both lists which occupy memory locations between the lower starting address and the lower end address of the two lists.

3.1.4 Efficiency Considerations

The transfer of operand elements between memory and the arithmetic unit can be made as efficient for list structured vectors as it is for vectors stored in the usual full form. Because the link word is at the *start* of the block, if blocks are above a certain minimum size, there is time to emit the start address of the next block to the memory sufficiently far in advance to avoid a gap in the address generator→memory→AU pipeline. The writing of result elements back to memory may also be effectively pipelined.

Many vector processing systems use interleaved banks of memory to achieve the memory bandwidth required to run the arithmetic unit at full speed and in ESP, within a block of a list vector, interleaving will work effectively. However, even though the link address is known well in advance, if the first word of the next block falls into the wrong bank, there will be a hiatus in the interleaving. To avoid this, it is sensible to restrict all blocks to starting in a particular bank, and all pointers are thus multiples of the number of banks. For example, in an eight-way interleaved memory, to allow full use of interleaving and pipelining, pointers should be restricted to be a multiple of 16.

A potential performance limitation in the system so far described is the de-allocation operation (a de-allocate must be performed after most vector operations). How long this takes depends on the number of blocks on the free list and on the vector list being disposed, and on the start and end positions of both lists; in the worst case, the algorithm must examine every block on both lists to complete the de-allocation. However, there are several ways of mitigating the delays caused by this operation.

Firstly, note that the list restructuring remaining to be performed at any time during de-allocation takes place *beyond* the locations pointed to by A and B. Since the new free list pointer is set up at the *start* of the de-allocate, writing the result of the next vector operation into the free list can commence almost immediately after any pending de-allocate has started, and can continue concurrently with the de-allocate, subject to the condition that each free block used by the write must start at a memory address *less than* the value of pointer A (which in the particular algorithm used, is itself always less than B). If this condition fails, the write must be delayed until it is again satisfied. In this way, so long as

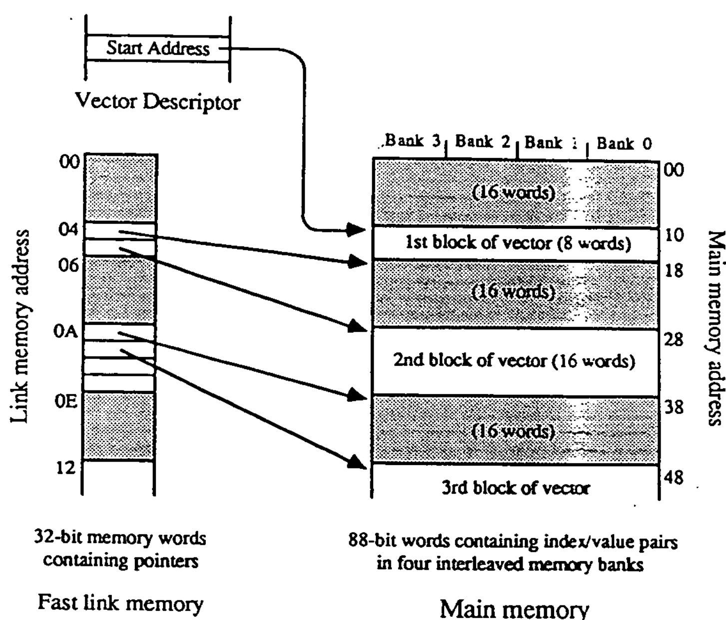

Figure 3: List vector storage with separate link memory

the de-allocation of vectors normally takes less time than writing them, de-allocation need not necessarily delay the processor.

For de-allocation to work concurrently with writing (which is also often concurrent with the reading of one or two streams of input operands *into* the arithmetic unit), there must be plenty of memory bandwidth, and since all pointers are restricted to one bank of memory (all blocks start in the same bank), the bandwidth requirement is considerably greater for that bank than for the other banks. One way of providing this extra bandwidth is to provide a completely separate memory to hold the block link words, as illustrated in figure 3. Here, the words in the link memory are 32 bits wide, which is large enough to hold one pointer only. There is one word of link memory per four words of main memory, and the internal and external pointers of a list vector block which starts at main memory address a are at link memory addresses $a/4$ and $a/4+1$ respectively. Since every block uses two link memory words, the minimum block size is 8 words. Virtual to real mappings must be maintained in parallel on both memories, but the link memory need not be accessible by the vector or scalar processors, only by the memory controller. Of course, if blocks are large, large amounts of the link memory will be unused, and so the dual memory system introduces an overhead of wasted memory area. However, this overhead is more than compensated for by the main memory bus bandwidth gained, and by the simpler bus arrangements which result from the separation of the two memory types.

Finally, alternative deallocation strategies have been considered. By linking lists in both directions, and tagging the block link words in free blocks, it is possible to deallocate a list of blocks in a time proportional to the number of blocks in the list being deallocated, independent of the number of free list blocks (see for example, memory management algorithms in [8]). Because, as described above, deallocation can be overlapped with vector writing, we do not think that the extra complexity of deallocation hardware would be justified. However, simulations of ESP's mechanisms, and we hope, the prototype hardware itself, will be flexible enough to experiment with alternatives in this respect.

3.2 The LP problem on ESP

List vectors do not waste store, and they provide immediate and implicit identification of the non-zero positions of the vector. An operation like the Gaussian Elimination step $B_{n*} \leftarrow B_{n*} - S * B_{p*}$, where both vectors are of roughly equal sparsity, will execute efficiently using list vector storage and a single ESP vector instruction, which operates by streaming operand elements into the arithmetic unit in the manner described in section 3.1.1 above. This is also true of other vector/vector operations on vectors in list form, where both vectors are of roughly equal sparsity. In the *pricing* step of LP (section 2.1 above), however, the price vector is much denser than the column vectors it is multiplied into. To execute the multiplication by streaming in two list vectors would be inefficient, as most of the elements of the price vector would be discarded because there is a zero in the corresponding position of the column vector.

If one vector is several times denser than the other, the scalar product operation is more efficient with the denser vector stored in *full* form. The sparser vector (stored in list form) moves into the vector unit element by element, and the index fields of the elements are used to offset into the denser vector, using normal indexed addressing (this is similar to the *gather* operation supported in hardware in several vector supercomputers [7]). Obviously, access to the elements of the denser vector is slower than streaming a vector out of memory, because the elements accessed are in non-consecutive locations, and memory bank interleaving will be interrupted. and so this method of access is only preferable where the sparsity of the two vectors differs by a factor of four or more.

Performance on list vector/full vector operations can be further increased, if the same full vector is to be operated on many times over, by providing a fast access vector register near the arithmetic unit, to hold the full vector operand. This reduces memory bandwidth use, and circumvents the problem of failure of interleaving.

The usefulness of a vector register is even clearer in the case

of the *BTRAN* and *FTRAN* steps. BTRAN also requires a scalar product of a sparse η vector with a vector which is (for most of the BTRAN steps) less sparse, and then requires that one element of that less sparse vector be replaced with the result of the product. This final step requires access to an element of the vector with specified index, and is clearly *very* inefficient on a list vector. However, it can be carried out with ease if the vector is stored in full form in a register. The result of the complete series of BTRAN steps is the price vector, which is thus conveniently in the register ready for the pricing step.

FTRAN requires a summation of a sparse η vector (scaled) with a vector which for most of the FTRAN steps is denser than the η vector. The scaling factor to be applied to the η vector is an element of the denser vector, specified by its index value. It is therefore useful to store the vector being updated in full form in the register, to allow indexed access to these scaling elements.

Finally, the *re-invert* step provides special problems. These cannot be overlooked, as it is intended that ESP should be generally useful on a wide range of sparse matrix problems, including the solution of large sparse linear systems of equations by Gaussian Elimination, which corresponds to the re-invert step of LP.

3.3 Sparse Gaussian Elimination on ESP

In some cases, in particular when the matrix to be factorised is symmetric positive definite, it is possible to decide which elements to use as pivots on sparsity grounds only, before the elimination starts [2]. The matrix may be permuted so that pivoting proceeds down the diagonal, and it is possible to work out in advance (ignoring cancellation during the subtraction steps) the positions of non-zeroes in each pivot column. Elimination will then be very efficient on ESP with the rows of the matrix stored as list vectors. Many Gaussian Elimination implementations on standard computers need to switch over from 'sparse code' to 'dense code' when the density of the filling matrix reaches a critical value. This is not necessary on ESP - operations on list vectors, such as that illustrated in figure 1, remain more efficient than equivalent operations on full vectors, however much the vectors fill in.

As described in section 2.1.4 above, Gaussian Elimination on matrices which are not symmetric positive definite requires knowledge of the number of non-zeroes in each row and column of the partially eliminated matrix, and also requires rapid access to the non-zeroes in each column. In this case, the number and position of non-zeroes in each row and column of the matrix as elimination proceeds cannot be determined in advance. If the matrix is stored within ESP as row vectors in list form, then the elimination steps themselves are efficient, but choosing the pivot is very inefficient. This is because the number and positions of the non-zeroes in each *row* are available (vector descriptors include a field specifying the number of non-zeroes in the vector), but not the corresponding information for each *column*. It is also not possible to access directly the non-zeroes in a specified column - one must search down the row vectors to find them. To support the pivot choosing algorithm, codes for Gaussian Elimination of sparse indefinite matrices which run on *scalar* machines normally store the matrix as a linked structure linked in two directions, along both rows and columns. However, to provide links to matrix elements by column is directly at variance with the dynamic nature of list vector storage in ESP - if a row of the matrix is operated on by a vector instruction, it will move in memory, invalidating any pointers to its elements. An extra facility has therefore been added to the vector processor in ESP, known as the *sideways list unit (SLU)*, which supports maintenance of lists of non-zero *indices* (but not values) by column as the elimination proceeds (ignoring cancellation during subtractions).

3.4 The Sideways List Unit

The SLU keeps updated counts of the *number* of non-zeroes per column of a matrix, and their positions, throughout Gaussian Elimination by rows. The list of non-zero positions in each column is kept in main memory as a linked list of single memory locations each holding a non-zero position (24 bits) and a link to the next word on the list (32 bits), as illustrated in figure 4. These 56-bit pairs are held in the 64-bit value field of locations of a vector which is itself stored in the ESP list format described in section 3.1.3 above. Many such lists of non-zero positions can be stored inside a single ESP list vector and, since the non-zero position lists do *not* have to be linked forwards in memory, a single non-zero position list can extend through several ESP list vectors. The reason for storing the non-zero position lists *inside* ESP list vectors is that memory space for extending the non-zero position vectors can then be allocated using the standard list vector allocation mechanism, and when elimination is complete, all the space can be de-allocated by de-allocating all the list vectors used for this purpose.

After the first pivot is chosen, but before the elimination steps using that pivot row are executed, a list vector is produced (using a vector instruction which generates a vector of specified length) with number of non-zeroes equal to the maximum number of new non-zeroes that the elimination with that pivot row can possibly produce. (That number is the multiple of the number of non-zeroes in the pivot row and the number of non-zeroes in the pivot column, and has already been calculated during pivot choice using the Markowitz criterion.) The descriptor of this list vector is passed to the SLU, so that the vector can be used as space into which to expand the lists of non-zeroes in the matrix columns, during the elimination steps with the first pivot row. The vector is known as the *SLU space vector.*

A single elimination step consists of the operation $B_{n*} \leftarrow B_{n*} - S * B_{p*}$. Whenever the arithmetic unit produces a non-zero in an index position in the result vector B_{n*} which contained a zero in the left-hand input operand, that element is a new non-zero, and its index, i, (its column position in the matrix) is passed to the SLU. The SLU contains a register holding the row number n, and three vector registers, one (the *count register*) holding a count of the number of non-zeroes in each column of the matrix, the second (the *base register*) holding the address of the start of the list of non-zero positions for each column, and the third (the *address register*) holding, for each column, a pointer to the address of the next free location in the list of non-zeroes in that column. On receiving an index i from the AU, the SLU increments the non-zero count for column i, and adds the row number n of the new non-zero onto the non-zero list for column i, by writing it, togther with a link to the next free location in the SLU space vector, to the address pointed to by the ith entry in the address register. It then updates that entry in the address register, loading into it the address of the next free location in the space vector. Since the space vector is an ESP list vector, determining the next free location may involve following a link to the next block of the space vector.

When all the elimination steps with the first pivot row are complete, the SLU space vector may still contain some unused locations, as the real amount of fill-in may have been less than the possible maximum calculated at the start. The SLU maintains a count of the number of locations remaining in the space vector. After the second pivot is chosen, the maximum possible fill-in during elimination with *that* pivot may be calculated, and a new list vector generated and queued for use by the SLU, to replace the current SLU space vector when it is full. This is repeated for

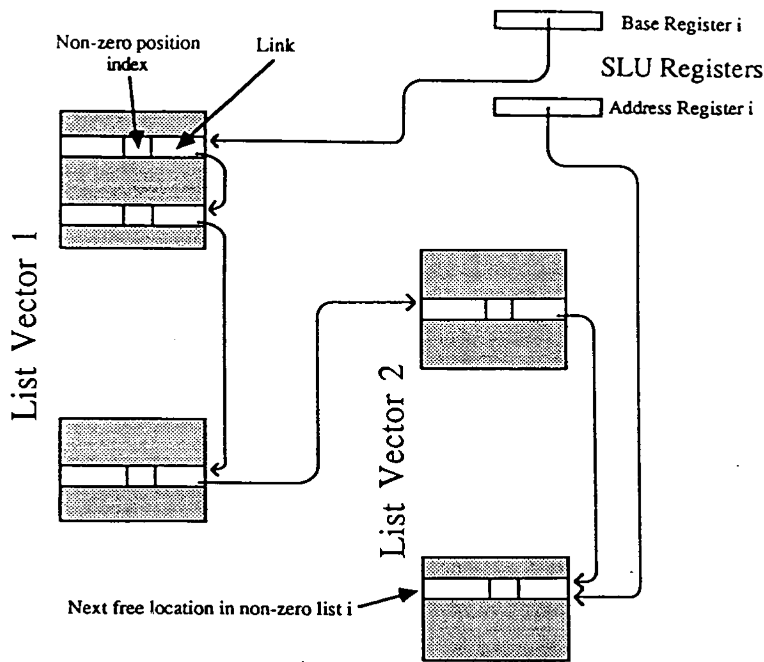

Figure 4: The structure of non-zero position lists

each new pivot, and ensures that the SLU will never run out of space.

The information maintained by the SLU is accessed by ESP's scalar/control processor during pivot choice. The new non-zero counts for all the columns in which there were non-zeroes in the previous pivot row (these are the only columns which can have had extra non-zeroes added during the elimination steps with that pivot row) are read from the SLU to enable pivot choice by the Markowitz criterion. The positions of the non-zeroes in the chosen pivot column are then read from the SLU, which itself reads them direct from main memory, following the links. The *values* of the non-zeroes must be found by using the vector pipeline to search down the relevant rows until the correct column index is reached (using a vector instruction which extracts from a vector the element with a specified index). Depending on sparsity, these operations are likely to involve an overhead of perhaps 100% on top of the time for the subtraction steps of the elimination, and this compares very favourably with the total time required for pivot choice (compared with elimination time) in scalar Gaussian Elimination codes for sparse indefinite matrices [2].

Before the elimination steps start, the SLU must be fed the positions of the non-zeroes in the original matrix, and this is achieved by first allocating a space vector large enough to contain all the non-zeroes in that matrix. The SLU registers are then initialized by loading zeroes into the count register, and loading the base and address registers with locations taken from the space vector. The operation $B_{n*} \leftarrow 0 + B_{n*}$ is then performed on each row of the matrix. Here, the left-hand operand is always zero, so every non-zero position in B_{n*} is passed to the SLU to be added to the relevant column non-zero position list.

4 Performance

Mechanisms similar to those described above for operating directly on compressed sparse vectors have been implemented in the *software* of sparse matrix programs for many years [1, 3, 11, 2]. Such programs run on scalar processors and do not use vector instructions. By providing hardware to implement the vector loops in these programs as single vector instructions, with separate hardware units operating in parallel on the subfunctions which scalar implementations control with separate instructions, we expect to achieve an order of magnitude improvement in the arithmetic rate of these codes.

More recently, there has been interest in the use of the hardware vector indirect addressing facilities (*scatter/gather*) provided in some vector machines (eg CYBER 205, IBM 3090VF), to support sparse vector operations. Because of the bank conflict problem (see section 3.2 above), such operations will always be several times slower than ESP's sparse vector instructions, if the vectors concerned are of roughly equal sparsity. Where the sparsity of the two operand vectors differs markedly, indirect addressing is relatively more efficient, and ESP's provision of a large vector register allows such operations to proceed at the full rate of the arithmetic pipeline.

A further problem (on existing machines) with vector instructions involving indirection into one of the operand vectors is the long instruction startup time, due to the long effective pipeline length. Because LP problems involve sparse vectors with a very small number of non-zeroes, we have been concerned to keep vector startup times to a minimum, and have therefore decided to implement the vector pipeline as several short, independently controlled, sections. A queue is provided for vector instructions which have been issued by the control processor, but not yet executed, and there is a mechanism for the control processor to determine whether a particular vector instruction has completed.

Together, these provisions allow us to have several vector instructions flowing through the pipeline at once, substantially reducing effective start-up times. Simulations are underway to quantify the resulting speed-up on representative problems, and to determine the optimum number of pipeline sections.

The detailed design of the prototype ESP is now underway. This will be built in slow technology (clock speed 100-200ns), using standard VLSI arithmetic components, and will achieve peak speeds of up to 20 MFLOPs. The prototype will allow us to make a thorough investigation of the new mechanisms on real, large, sparse matrix problems.

5 Conclusions

Pipelined vector processors achieve their high performance in part by taking advantage of the storage of vector elements in sequential memory locations. The mechanisms developed for ESP allow this advantage to be maintained in the case of sparse vectors, by providing a new form of vector storage, the *list* vector. The list form wastes no space for the zeroes in a vector, and unlike compressed sparse vector storage mechanisms in other machines, solves the problem of fill-in.

ESP supports all the normally found vector/vector operations, including two operator functions such as scalar product and the Gaussian Elimination step $B_{n*} \leftarrow B_{n*} - S * B_{p*}$, as single vector instructions which will operate directly and efficiently on full form vectors, list form vectors, or a combination of the two. The operations work efficiently over the whole range of non-zero densities. This allows the advantages of vector processing to be extended to the case of sparse vectors.

However, some particular computations commonly performed on sparse matrices, such as the Gaussian Elimination of an indefinite matrix, require that information be maintained about the matrix as a two-dimensional object, rather than simply as a set of one-dimensional vectors, throughout the program's execution. Although it is possible to write algorithms which treat the matrix symmetrically, allowing it to be viewed both by row and by column, fully symmetrical treatment is not possible without sacrificing the fundamental advantage of sequential vector element storage. A compromise solution to this particular problem has therefore been developed, which retains the full advantages of the *one-dimensional* list vector system described above, but in addition allows information about the non-zero distribution in the *two-dimensional* matrix to be maintained in easily accessible form. The mechanism which supports this, the *sideways list unit*, has been developed with the specific requirements of Gaussian Elimination codes in mind, but it is expected to prove useful in other sparse matrix computations.

There has recently been interest in putting sparse problems on to *parallel* processors [4, 5]. We believe that, whilst coarse grain parallelism will clearly enable the solution of very much larger problems, the increasing scale of silicon circuit integration will mean that individual processors can cost-effectively incorporate extra hardware to exploit parallelism at the level of single vector or matrix operations. This is the level of parallelism at which ESP derives its advantages. Such individual processors may then be connected to work in parallel on larger problems.

Acknowledgements

The ESP project is a collaborative project between the Departments of Computer Science and Mathematics at the University of Edinburgh, and High Level Hardware Ltd., of Oxford. The authors would like to acknowledge the support of High Level Hardware, and of the UK Science and Engineering Research Council.

References

[1] James R. Bunch and Donald J. Rose, editors. *Sparse Matrix Computations.* Academic Press, 1976.

[2] Thomas F. Coleman. *Large Sparse Numerical Optimization.* Lecture Notes in Computer Science. Springer Verlag, 1984.

[3] Iain S. Duff and G.W. Stewart, editors. *Sparse Matrix Proceedings 1978.* SIAM, Philadelphia, 1979.

[4] H. Amano et al. $(SM)^2$: Sparse matrix solving machine. In *ACM Proc. 10th Symposium on Computer Architecture,* pages 213–220. ACM, 1983.

[5] H. Amano et al. $(SM)^2$-II : A new version of the sparse matrix solving machine. In *ACM Proc. 12th Symposium on Computer Architecture,* pages 100–107. ACM, 1985.

[6] High Level Hardware Ltd., Headington, Oxford, UK. *ORION Time Sharing Manual,* 1986.

[7] R.N. Ibbett and N.P. Topham. *Architecture of High Performance Computers,* volume 1. Macmillan, Basingstoke, Hampshire, UK, 1989.

[8] D.E. Knuth. *The Art of Computer Programming,* volume 1 - Fundamental Algorithms. Addison-Wesley, 2 edition, 1973.

[9] H.M. Markowitz. The elimination form of the inverse and its application to linear programming. *Management Science,* 3:255–269, 1957.

[10] D. Morris and R.N. Ibbett. *The MU5 Computer System.* Macmillan, London, 1979.

[11] Ole Østerby and Zahari Zlatev. *Direct Methods for Sparse Matrices.* Lecture Notes in Computer Science. Springer-Verlag, 1983.