MU6V: A PARALLEL VECTOR PROCESSING SYSTEM

R. N. Ibbett, P. C. Capon & N. P. Topham

Department of Computer Science
University of Manchester

ABSTRACT

MU6V is a parallel vector processing system in which a linear set of processors, each capable of both vector and scalar operations, is interconnected by a common communication highway. Memory is fully distributed among the processors and interprocessor communication occurs by means of global broadcasts controlled in hardware by a data driven synchronization mechanism. A prototype model based on 68000 microprocessors, in which vector orders are emulated as "extracodes", has been constructed. A host machine provides a software support environment and a run-time system in the prototype provides diagnostic information. Programming language features have been designed to exploit this configuration and an example algorithm is presented.

INTRODUCTION

The MU6V computer described here is a prototype of a potentially very high performance system. It was designed for use in the solution of compute-bound problems amenable to both vectorization and to decomposition into concurrently executable sub-processes. Typical applications could involve, for example, direct or indirect solutions of equations, solutions of Laplace equations or matrix multiplication. An important characteristic of the algorithms used is that the number of data items transferred from one sub-process to other sub-processes is at least an order of magnitude less than the number of arithmetic operations within that sub-process.

The design of MU6V can be traced back to MU5 [1], a system in which instructions contained information about the type of operand they required (scalar, vector element etc.). This information was used, for example, to control special hardware provided to assist with accesses to vector or array elements, although operations on whole vectors were organised by program loops rather than by specific vector orders. An MU5 with vector orders would have resembled the CDC Cyber 205 computer [2] with the MU5 Primary Operand Unit corresponding with the Cyber 205 Scalar Unit and the MU5 Secondary Operand Unit with the Cyber 205 Vector Unit. In principle the number of Vector Units could be expanded in either MU5 or the Cyber 205 to give a system (fig.1) similar to the TI-ASC [3].

The number of Vector Units which can be operated in parallel in the system shown in fig.1 is limited by two principal factors, the first being the performance of the Scalar Unit. If all the Vector Units are to be kept busy the Scalar Unit must be able to initiate operations in all of them within the time taken by any one of them to execute a complete vector operation. The Scalar Unit is also required to carry out some scalar arithmetic on results produced by the Vector Units, and this essentially sequential activity can itself limit performance. In MU6V each Vector Unit is endowed with scalar facilities and made independently programmable. The result is a set of parallel processors in which each processor happens to have a vector instruction capability. The prototype is based on 68000 microprocessors and the instruction set of each MU6V processor comprises 68000 scalar instructions augmented by a set of emulated vector instructions. These instructions are planted by the compiler as ordinary in-line code but the op-codes used for the vector instructions cause the processor to trap to supervisor-level instruction emulator "extracodes".

The second performance limitation of the system shown in fig.1 is access to common memory and a variety of mechanisms have been used in other systems to allow such access from a number of processors (cross-bar switches, multi-port memories, etc.). All involve additional hardware and/or time penalties, and alternative system design techniques which reduce the number of accesses to common memory have therefore been sought. Systolic architectures [4] have significant attractions from this point of view, but suffer from the drawback that the preferred topology of interconnection is algorithm dependent.

An alternative solution to the common memory problem is to distribute some or all of the memory among the processing units. Now most memory requests are satisfied by local memory, and the common memory bottleneck is largely removed. Accesses to common memory (or to the memories of other units in a fully distributed system) are still necessary, however, since in most applications some results must pass between units, and some global data must normally be available to all units. MU6V uses a fully distributed memory and the mechanisms used for the communication and synchronization of data passing between units are essential features of its design.

COMMUNICATION AND SYNCHRONIZATION

One of the main objectives of the MU6V design was that it should consist of a set of parallel vector processors which could be used effectively on a wide variety of algorithms. The system was not to be constrained by an interconnection structure which would restrict the movement of data between processors. Thus a topology was sought which would allow communication between any combination of processors to occur with equal ease and without reference to their physical position. The structure chosen was that of a linear array of processors, each with its own local memory and each connected to the others by a common communication highway (fig.2).

Communication Protocol

Inter-processor communication occurs when a process running on one of the processors reaches a point where it must update the value of a variable held elsewhere in the system. The processor effects this communication by placing on the highway a packet containing the new value for the variable being updated plus an identifier which allows other processors to recognise it. This value is now the most up-to-date version of the corresponding variable, and consequently all vector units holding a copy of the variable must update the value they hold for it. In the prototype system the highway is implemented as an LSTTL tri-state bus, and an arbitration unit is used to ensure an orderly flow of information on to the bus. Within each vector unit an autonomous logic unit, acting in parallel with the instruction processor, recognises and either accepts or rejects each result placed on the bus.

This communication protocol implies that global variables are not read from one processor by another, but are sent from one processor to all others at the moment of production. This 'pushing' of global information has a number of advantages over the more conventional 'pulling' of information by the issuing of read requests. Firstly, a conventional read-request scheme requires two bus cycles, one for the outgoing memory address and the other for the returning information; a write-scheme only occupies a single bus cycle per global result and furthermore, reduces significantly the time lost through latency delays. Secondly, if several vector units require the same result, then in a read-request scheme they would each access it separately and generate a large number of read requests to the same value.

This system is to some extent analogous with the Common Data Bus used in the floating-point unit of the IBM System/360 Model 91 [5]. Here each result produced by an arithmetic sub-unit is placed on the Common Data Bus, together with an identifier, and is taken off by whichever device attached to the bus holds a matching identifier. The MU6V system is different, however, in that not every result produced within a unit is broadcast to other units. Thus mechanisms are required in both hardware and software to determine which results should be transmitted to the bus, and which results should be taken from it. Identification of results placed on the highway requires the use of a global addressing mechanism, and a synchronization mechanism is required to ensure that the corrrect copy of a given result is used in each unit.

Global Addressing

The global addresses used in MU6V do not identify specifically the unit(s) in which the corresponding data items are held. They are therefore virtual addresses, and within each unit there must exist a separate virtual-to-real address translation mechanism. In a conventional virtually addressed system the basic units of store area handled by the store management system are normally pages or segments. This allows arbitrary data structures to be dealt with on an equal basis. In a machine specifically geared to vector processing, the vector is clearly the most important data structure, and the addressing mechanism ought therefore to be tailored to the efficient manipulation of vectors. In MU6V a vector identifier (or name) is used in a manner analogous to a segment identifier in conventional systems and a virtual address references a vector element by a combination of a vector name and a vector subscript:

```
| NAME | SUBSCRIPT |
```

When performing the virtual-to-real address translation one of the most important items of information required about a particular vector is its real origin. The simplest and most straighforward way of translating from a vector name to the corresponding vector origin is by a direct table look-up. The vector name field width defines the length of the look-up table while the size of the vector origin defines the width of each entry in the table. If this logic is implemented using 256K RAM devices, for example, a table length of 256K may be assumed, and this corresponds to a field width, for the vector name, of 18 bits. In the prototype MU6V, however, 64K RAM devices are used and the vector name is therefore restricted to 16 bits.

Synchronization Protocol

A synchronization protocol is required in MU6V to ensure an orderly flow of data between producers and consumers. There are two instances in which the synchronization constraint must be applied:

(a) Processor B consumes its old value of variable X and some time later attempts to access X, expecting to find a new value; however, no new value has arrived and processor B must not reference its copy of variable X until a new value does arrive.

(b) Processor A produces a result for variable X and attempts to send this result to processor B, but processor B has not yet finished processing the previous instance of X.

The protocol required to deal with case (a) is straightforward; the processor reading the out-of-date variable must be held up until the new value arrives. Case (b) is more difficult, however. Other processors may be able to accept the new value for X and may even be waiting for it. Processor A therefore places its output packet on the highway, but unless all the processors containing a copy of variable X indicate that they are able to accept this new result, processor A must abandon its attempt to transmit the result and wait for the highway arbitration unit to allocate it another cycle. Processors which do not contain X indicate immediate acceptance so far as the highway is concerned.

The mechanism used to ensure proper synchronization is relatively simple. A single bit is associated with every word of vector memory to indicate the validity of that vector element. The act of writing to an element of a vector memory automatically sets the synchronization bit, while the act of reading the element may (according to a parameter of the instruction) reset it. While the bit is set the local vector unit may read from or write to that element, but the element may not be overwritten by data coming from other vector units. If the bit is not set then the local vector unit may write to the element but may not read it, and data may be written into the element from any other vector unit. Thus after receiving new data an element may be read, and after being "consumed" an element may receive a new value. If synchronization is not required for a specific vector, synchronization may be inhibited for that vector. Mechanisms similar to these are used in the Denelcor HEP [6]. Like MU6V, HEP contains multiple processors, but within any one HEP processor there is also apparent parallelism created by the interleaving of multiple processes at the instruction level within the pipeline. This avoids instruction dependency delays and hence improves overall performance. MU6V uses vector orders for this purpose.

THE VECTOR INSTRUCTION SET

The design of the vector instruction set was influenced by a variety of considerations. Important among these were the naming concept and the use of descriptors derived from MU5, and the need to handle sparse vectors. A vector is identified in MU6V by its name, which acts as a pointer to a descriptor containing not only an origin address (as in the Cyber 205, for example) but also type and size information. As in MU5, elements of various sizes may be close packed in store, and implicit in the design of the instruction set is the assumption that there will exist hardware capable of automatically packing and unpacking elements within store words. The type information defines the nature of the vector eg. directly or indirectly accessed, dense or sparse, and by encoding this information as part of the vector description, instructions can be independent of the vector data type.

Operation Codes

A typical vector instruction operates on two source vectors (V1 and V2) to produce a destination vector (V0) and may involve the use of a single scalar operand (S) in instructions of the form:

$$V0 \leftarrow V1 + (V2 * S)$$

In the 68000-based prototype vector instruction are implemented by emulation as "extracodes". Each vector instruction consists of a 16-bit op-code and a 16-bit operand specification word followed (optionally) by up to three 32-bit literal address fields (fig.3a). The four most significant bits of the op-code define the instruction as a vector operation and cause the processor to switch to the emulation code. The vector operations themselves are grouped into four types (defined by bits 10 and 11 of the op-code): Data Movement, Data Identification, Arithmetic and Accumulative Arithmetic.

Within each group eight major functions may be defined and for each major function eight sub-functions may be defined, although none of the sets is currently full. The Data Movement group of major functions includes MOVE, FILL and COMPRESS, where the criterion used by the COMPRESS function, for example, is defined by the sub-function (copy elements of V1 = S to V0, elements of V1 > S to V0, etc.). The Data Identification group includes SEARCH instructions which return, for example, the value or index of the maximum or minimum element within a vector, COMPARISON instructions which return the index of the first corresponding pair of elements in the two source vectors which satisfy a condition specified by the sub-function (=, >, etc.) and COUNT instructions which return the number of elements in source vector V1 satisfying a condition relative to the scalar operand S. The Arithmetic group includes the major functions ADD, SUBTRACT, MULTIPLY, etc. with the sub-function defining the choice of operand in instructions such as:

```
V0 <- V1 + V2
V0 <- V1 + S
V0 <- V1 + (V2 * S)
```

or the type of SHIFT or LOGICAL operation, while the Accumulative Arithmetic group includes functions which allow the logical AND, OR, etc. or the arithmetic SUM of all elements in a vector to be formed, and in particular the sum of the products of two vectors, ie. the vector inner product.

The scalar operand is implicitly specified by the function to be taken either from the (single) accumulator or from one of the four B (index) registers which the instruction set assumes to exist in the processor. These are mapped by the extracodes on to the data and address registers within the 68000 processor. If the accumulator is specified the operand size may also be selected to be 8, 16, 32 or 64 bits. An additional register used by the instruction set is the N register, the content of which defines the number of elemental operations to be performed by the vector instruction.

Operand Types

The operand specification word contains the information required to access the (maximum of) three vectors associated with each vector instruction. Associated with each vector are two access mode control bits, the Z-bit and the W-bit, which operate independently of both the function and the vector type, and must therefore be contained within the operand specification word. This leaves three bits per vector for specification of the name. These bits are therefore interpreted as shown in fig.3a; the name may either be a 32-bit literal following the instruction or may be the contents of one of the four B-registers. In either case the 32-bit word is interpreted as a 16-bit vector name and a 16-bit starting index (ie. an offset to be added to the origin address at the start of the operation).

The Z-bit controls the use of the communication and synchronization mechanisms. When the Z-bit is set for a source operand (V1 or V2) the act of reading an element of that vector causes the synchronization bit for that element to be reset. When the Z-bit is set for a destination operand (V0) the act of writing to an element of that vector causes the data to be distributed globally rather than being written into the local vector memory.

The W-bit enables or disables the wrap-around of indices within the bounds of the specified vector. If the W-bit is set and the processor attempts to access the next element beyond the upper bound of the vector, the element at the lower bound (origin + offset) is taken instead and further indexing proceeds normally from that point. If an attempt is made to access an element beyond the upper bound without the W-bit set, a bound fail exception is generated. For certain types of algorithm wrap-around allows better utilisation, and hence greater overall performance, to be obtained from the set of parallel processors.

The vector name is used to address a descriptor (fig.3b) containing the origin address (16 bits long in the prototype; 32 bits would be more realistic in a 'production machine'), the length of the vector (used as an upper bound in some circumstances), two operand size bits (vector elements may be of size 8, 16, 32 or 64 bits), two type bits and a defined bit. When a packet is received from the communication bus the first task which each unit performs (in parallel with all others) is a table look-up on the name field in dedicated memory to find the corresponding descriptor. Not all vectors are required by any given unit, however, and so the defined bit is used to determine whether or not that vector exists within the unit. The defined bits are initialised by special instructions within each processor before any input data values are broadcast at the start of a process and before any computed values can be produced.

SOFTWARE

The software system on MU6V is designed to exploit both the vector processing capability of the architecture and the parallel processing and synchronization mechanisms. A host machine (MU6-G [7]) is used to provide support for software development in the form of a cross-compilation system for an extended form of Pascal and for MUSL (a high-level system programming language). Code generated by these compilers is down loaded into the prototype via a serial link. The host also provides run-time support for application programs in the form of a remote filestore facility.

Pascal is augmented with a small set of implicitly defined vector handling procedures, through which the application code accesses vector memory and communicates with other processes. In the prototype these vector functions are emulated at a low level, and this provides a flexible environment in which to develop both algorithms for vector addressing mechanisms and protocols for inter-processor communication. Application programs are written in this augmented Pascal, cross-compiled on the host machine and down loaded into one of the MU6V processors (designated the "master")and tranferrred, where appropriate, into other processors via the common communication mechanism.

Within MU6V itself there exists a runtime system which provides a mechanism for interrogating the vector memory. The current prototype system also allows each vector unit to direct Pascal output to an appropriate window at the user terminal. These facilities, together with a diagnostic window which shows information relating to interprocessor communication, have been found to be extremely useful during the development of several algorithms, including the Gauss-Seidel algorithm described below.

A software simulator, running entirely on the host machine, has been developed. This uses the multi-tasking facilities of the MUSS operating system running on MU6-G [8] to simulate any number of vector units. This simulator runs MU6V programs and has facilities for producing a detailed trace of all communication and process synchronization.

Parallelism

Since the processors in MU6V are autonomous and independent it is necessary to control their operation at least to the extent of initiating them to perform a particular task and determining when the task is complete. This controlling process could be external to MU6V, but in fact resides in one of the MU6V processors which could be regarded as a master processor. This means that it can use the same facilities as all the other processors and if necessary take an active role in the computation. Many programs decompose naturally into such a master or "harness" process and subprocesses. The master performs initialisation before initiating parallel subprocesses. While the subprocesses continue the master process might test for termination or convergence of subprocesses and initiate further subprocesses.

Syntactically a subprocess is similar to a procedure with the word process replacing procedure in the heading. Initiation of a subprocess is similar to a procedure call. It is convenient to provide a language mechanism to indicate whether subprocesses may execute in parallel rather than sequentially. A standard parallel construct is one way of achieving this, e.g:

```
parallel
 begin
    pl;
    p2;
    p3
 end;
p4
```

In this example pl, p2 and p3 may be executed in parallel followed by p4. Often a row of similar subprocesses, each performing similar operations on different data sets is required. A construct of the form:

```
parallel i = 1 to n do
   p (i);
```

initiates n parallel processes parameterised by i.

The parallel statements need not always initiate separate processes. A statement of the form:

```
parallel i = 1 to n do
   f(i)
```

where f is an ordinary function allows the various f(i) ... f(n) to be executed in any order. This can be used even on a single processor to allow processing to be ordered according to data availability.

```
parallel i = 1 to n do f(x[i])
```

could first operate on whichever x[i] were available, given that access to x[i] is controlled by a synchronization mechanism. Sometimes several statements are executed sequentially within one strand of parallelism. Here a sequential construct can be used:

```
parallel i = 1 to n do
        sequential
           begin
              p(i);
              f(i)
           end
```

In this case each f(i) is executed after the corresponding p(i) but all p(i)s may be executed in parallel.

Communication and Synchronization

In the MU6V multi-processor system independent subprocesses can be handled by separate processors. Sometimes a problem can be partitioned arbitrarily into independent processes, in which case it is easy to assign one process to each processor and execute all the processes in parallel. The ordering of the processors in this allocation is arbitrary on MU6V. In most practical problems the processes will not be completely independent since data may need to be shared or passed between processes and also some synchronization of processes may be necessary. For example, if a matrix problem such as a solution to the Laplace equation is shared between a number of processes each operating on a region of the matrix, then the boundary elements of the regions need to be shared between the processes operating on each side of the boundary. Furthermore in some algorithms the use of the shared data must be synchronized. In some systems the software mechanisms for such sharing and synchronization are based on the concepts of shared data and semaphores. Alternatively, the use of explicit sharing may be avoided by using a CSP-like mechanism [9] for communicating information along channels between pairs of processes. This approach is used in Occam [10].

Any of these mechanisms can be implemented in MU6V. When global data is used, each sharing process has its own local copy of the item which is used whenever a read is made. A write to such an item must be interpreted by the hardware as a broadcast of the item to all the processes requiring it. As all copies of the shared data are the same, the programmer need not be aware that the sharing is implemented by multiple copies. The hardware protocol ensures that all copies are updated simultaneously. If the shared data is to be synchronized this may be done explicitly or implicitly. In an implicit scheme every read of a synchronized item would consume that item and every write would update by global broadcasting. This is somewhat restrictive and a mechanism which allows multiple reading of a synchronized data item is preferred. This arrangement corresponds precisely to the hardware mechanism of MU6V. Basing the notation loosely on CSP:

?

is used to accept a new value of V. As all data is identified by the hardware on the global bus the name V is effectively both a channel name and a variable name. The ?V request in the consuming process may occur at the point when V is used but this could lead to the producing process being held up unnecessarily waiting for V to be accepted. If the ?V request occurs as early as possible unnecessary waits can be avoided. The input takes place in parallel with other processing in the receiving process which is only held up if the value of V is not available when the eventual reference is made. A similar technique can be used in Occam.

A frequent action is to consume the available value of V and immediately request a new value for V. In this case the notation V? is a convenient shorthand so:

```
x := f(V); ?V;
```

can be written:

```
x := f(V?)
```

with ? as a postfix unary operator. The prefix operator ↑ is used to indicate production of a result to be broadcast:

```
↑V := expression.
```

The Gauss Seidel Example

The Gauss Seidel method for the indirect solution of a system of linear equations may be used to demonstrate some of the above features. Given a system of equations:

$$A\ x = b$$

and an initial approximation x1 to the solution, new approximations to each element of x, x1 ... xn, are computed in turn using the newest computed values of the other elements of x. Although the algorithm appears sequential all elements of x can in theory be computed in parallel. A simple algorithm for a single row can be expressed as:

```
 process gauss seidel row
  (i:row; arow:slice, x,b:vector, N:dim)

  repeat
     s  := 0, temp := x[i];
     parallel j := 1 to N do
               s := s + arow[j] * x[j]?;
     ↑x[i] := temp - (b[i] - s)/arow[i]
  until converged;
```

while a harnessing procedure might be:

```
 procedure harness;
  parallel
  begin
     parallel i := 1 to N do
       gauss seidel row (i,a[i,#],b,N);
     {compute convergence}
  end
```

The order in which elements become available to gauss seidel row is in fact x[i] .. x[N], x[1] ... x[i-1]. The non-determinism in the parallel statement can be used to avoid unnecessary waiting. Alternatively the parallel statement can be recast to permute j to the optimal ordering. The test for convergence can take place in the harness process. The harness process reads all the computed x[i]s broadcasting a new value of converged after each iterative sweep. "until converged" in gauss seidel row becomes "until converged?", while {compute convergence} in the harness becomes:

```
  repeat
    diverged := true;
    parallel i := 1 to N do
    begin
      diverged := diverged and
                  (x[i]-y[i]<limit);
      y[i] := x[i]?
    end;
    ↑converged := not diverged
  until converged?
```

The timing diagram (fig.4) shows that once this process is initialised very efficient processor utilisation results. Each process broadcasts one result after an inner product calculation and results are available in sufficient time to keep all the processes busy. Usually there will be many more rows than processors, so the solution is reformulated to allow gauss seidel row to deal with a selection of rows of the matrix rather than a single row. This change makes no difference to processor utilisation, except for end effects caused by slightly uneven distribution of work.

Vector Orders.

In a high performance implementation each unit in MU6V will execute vector orders directly, so language mechanisms must be provided to exploit this facility. Straightforward extensions can be made to a Pascal-like language in which array names or slices of arrays appear as operands in array operations. Actus [11] is one language providing this type of facility and much of the Actus notation is appropriate. However in its use of index sets for subscript selection Actus is more oriented towards the control vector approach to element selection than is convenient in MU6V.

The user may nominate the dimension of the array which is to be contiguous in memory using : in the subscript in place of .. e.g. var a,b : array [1:10, 1..5] of real. Here the first subscript is contiguous and varies most rapidly while the second subscript will cause one of 5 vector names to be selected. Some examples of possible array assignments are:

a := b	for the whole array
a[#,2] := b[#,3]	for one column

CONCLUSION

The design principles of a potentially very high performance computer system have been described. Performance is achieved by enabling a multiplicity of vector processors to co-operate efficiently on a single problem. A 68000 based prototype system has been built, capable of accommodating up to 16 processors, of which three have been constructed and brought to an operational state. Approaches to programming this system have been explored and a variety of powerful mechanisms enable the hardware to be exploited to the full in various applications such as direct and indirect solutions of equations. Thus the prototype has demonstrated the viability of the architectural concepts of the system. In an implementation based on current supercomputer technology a system with 16 processors and a 10 ns clock would offer a peak performance of 1.6 gigaflops.

ACKNOWLEDGEMENTS

The authors wish to thank Professor D.B.G. Edwards for the provision of facilities and equipment and N.P. Topham acknowledges financial support from the SERC. The authors are grateful to a number of colleagues for contributions to discussions about the project and among the reseach students who have been involved the authors would particularly like to acknowledge the contributions of Dr Q. Lin and Dr I.O.A. Sanusi.

REFERENCES

[1] R.N. Ibbett & P.C. Capon, "The Development of the MU5 Computer System", Communications ACM, Vol 21, 1978, pp 13-24.

[2] "Control Data Cyber 200 Model 205 Computer System - Hardware Reference Manual", Control Data Corporation, St. Paul, Minnesota, 1981.

[3] W.J. Watson, "The TI ASC - A Highly Modular and Flexible Super Computer Architecture", AFIPS FJCC Conf. Proc., Vol 41, 1972, pp 221-228.

[4] H.T. Kung, "Why Systolic Architectures?", Computer, January 1982, pp 37-46.

[5] R.M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units", IBM Journal of R & D, Vol 11, 1971, pp 8-24.

[6] J.W. Moore, "The HEP parallel processor", Los Alamos Science, No. 9, Fall 1983.

[7] D.B.G Edwards, A.E. Knowles & J.V. Woods, "MU6-G. A new design to achieve mainframe performance from a mini-sized computer", Proc. 7th Annual Symposium of Computer Architecture, 1978, pp 161-167.

[8] G.R. Frank & C.J. Theaker, "The design of the MUSS operating system", Software Practice and Experience, Vol. 9, 1979, pp 599-620.

[9] C.A.R. Hoare, "Communicating Sequential Processes", Communications ACM, Vol. 21, No. 4, 1978, pp 666-677.

[10] D. May, "Occam", ACM Sigplan Notices, Vol. 18, April 1983, pp 69-79.

[11] R.H. Perrott, "A language for array and vector processors", ACM Toplas, Vol. 1, Oct. 1979, pp 177-195.

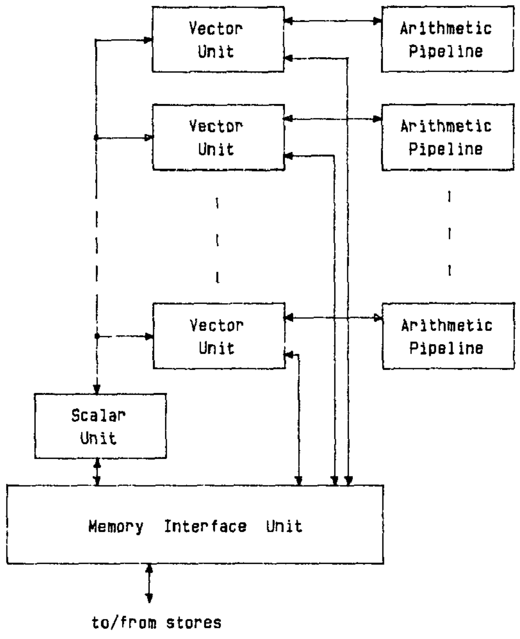


Figure 1 A Multiple Vector Unit System

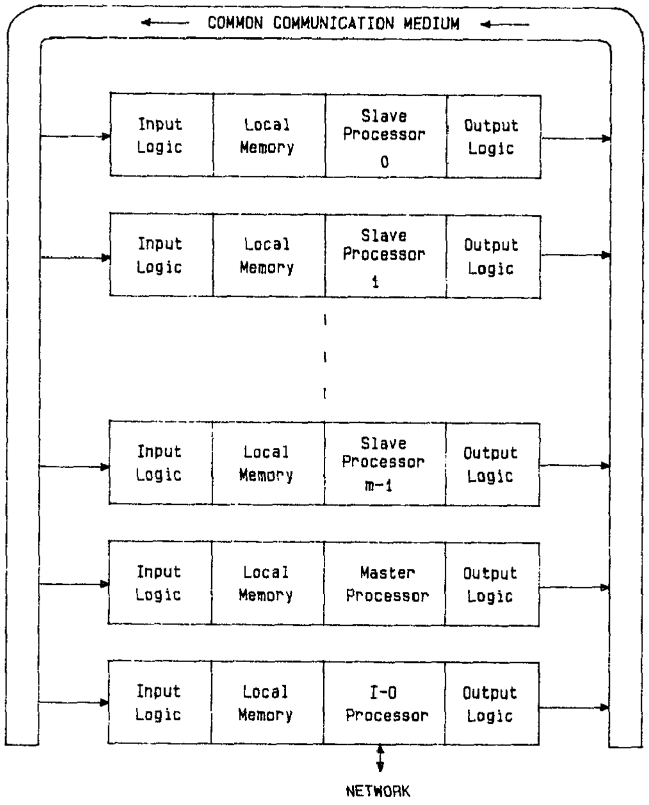
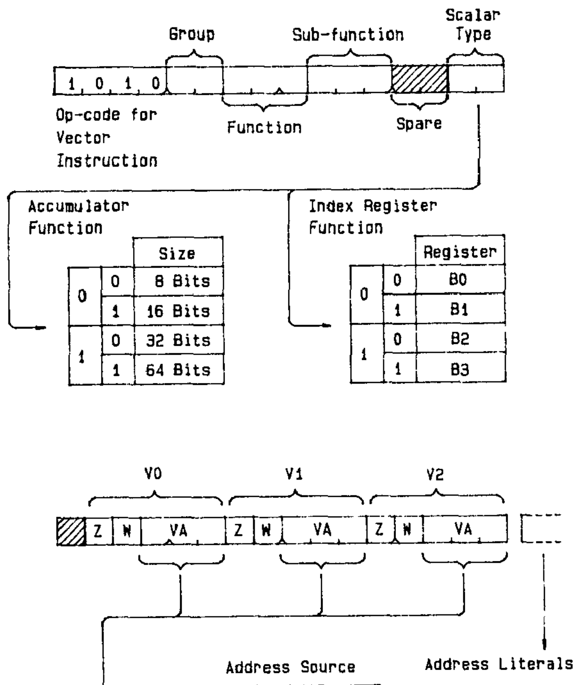


Figure 2 MU6V System Organisation



		Size
0	0	8 Bits
	1	16 Bits
1	0	32 Bits
	1	64 Bits

		Register
0	0	B0
	1	B1
1	0	B2
	1	B3

Address Source

1	X	X	Literal
0	0	0	B0
		1	B1
	1	0	B2
		1	B3

Figure 3a Prototype Vector Instruction Set

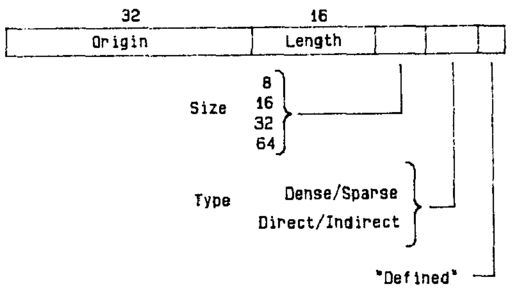
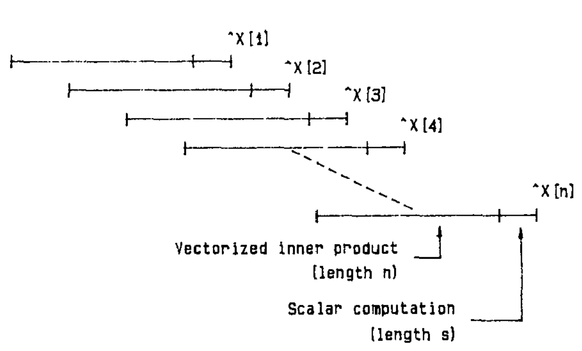


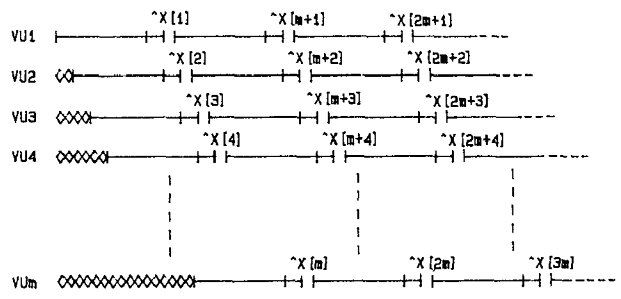
Figure 3B Descriptor Format



^X[1]
^X[2]
^X[3]
^X[4]
^X[n]
Vectorized inner product (length n)
Scalar computation (length s)

Sequential Processing Time = k.n(n+s)
Parallel Processing Time = k.n(1+s)
(for k iterations)

Figure 4a Gauss Seidel Row Calculations



◇◇ = idle time

$$\text{Vector Unit Utilisation} = \frac{\left\lfloor \frac{n(s+1)}{n+s} \right\rfloor}{s+1}$$

Figure 4b Gauss Seidel Performance When m < n