North-Holland
Microprocessing and Microprogramming 30 (1990) 657-664

Vector addressing processor for direct and indirect accesses

J-L. DEKEYSER - Ph. MARQUET - Ph. PREUX

Laboratoire d'Informatique Fondamentale de Lille
Université des Sciences et Techniques de Lille
59655 Villeneuve D'Ascq Cedex
FRANCE
preux@lifl.lifl.fr.eunet *or* preux@frcitl81.bitnet

Abstract

High performance computers are centered around the concept of vector processing. The first super-computer generation is single processor oriented. The last trends introduce parallel processing concepts in this field (e.g. Cray 2). In this context, we propose to associate a specialized processor in vector addressing to vector processors. This one takes in charge the whole computation required to access either a contiguous vector, or a sparsed vector controlled by an index vector (gather/scatter operations). An elementary vector access processor is in charge of vector operand fetching with a limited length. A modular cluster will increase length of fetched vectors.

Keywords

super-computer, vector processing, gather/scatter, vector addressing, contiguous / sparsed vectors, addressing processor, vector memory

Introduction

The majority of super-computers have their own language and their own development environment. To program this type of machines, each manufacturer proposes either an explicit vectorial language [12] (Cray CFT 77 [6], CDC FTN 200 [4]), either a standard scalar language (Fortran 77 [10] or C [15]) or a derived one associated with an automatic vectorizer (VAST 2, EAVE [3]). In spite of their interest in vector handling, the existent explicit vector languages do not facilitate the application portability. Software development using these languages requires super-computer utilization. Vectorizers are only efficient on scalar loops written keeping in mind vector processing [1]. This programming technic facilitates neither the design nor the legibility of programs. Our project leads to the definition fo a language - EVA [8] - with vector explicit handling. The EVA compiler produces DEVIL code, the machine language of the virtual vector computer MAD. For each EVA target machine, including scalar processors, a DEVIL translator will produce different executable codes. This will ensure the portability of programs (fig. 1).

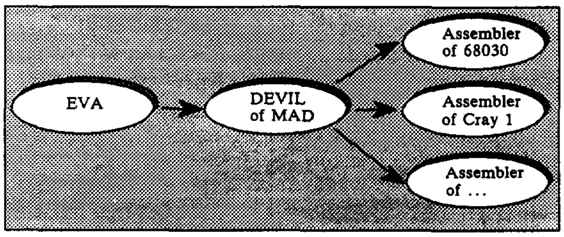

Figure 1 - Program development system

DEVIL handles either contiguous or sparsed vectors. A contiguous vector is defined by its *length*, number of elements, and a zone called *allocation zone* which contains the vector elements. A sparsed vector is defined by a data contiguous vector associated with a description - an index contiguous vector. This description selects some allocation zone elements, which are the sparsed vector actual elements.

Such vectors handling involves consequences on the memory unit (MU) architecture of the machine MAD. A memory unit read/write request contains several informations: for a contiguous vector, the allocation zone address and length; for a sparsed vector, the allocation zone address of the data vector, the address and the length of the index vector (fig. 2). In order to emulate our virtual vector computer

MAD in an efficient way, we propose a solution to process vector element addresses in a vector way.

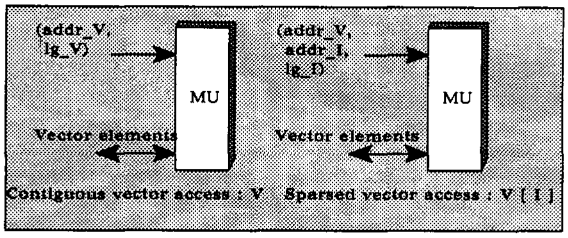

Figure 2 – Vector Memory accesses

With regards to the definition of *intelligent memory* [2], the *vector memory* concept has been introduced into MAD. The vector memory allows to access vectors as entities rather than scalar sequences. Contiguous and sparsed vectors are directly accessible using this memory. A contiguous vector access is realized by specifying its base address and its length. A sparsed vector access is obtained by specifying the base address and the length of the index vector, as well as the allocation zone address. Address computations are realized by a vector memory internal unit.

Each vector element access requires a direct or an indirect address computation. A memory unit internal vector processor (VAP) dedicated to address computations deals with this job. The VAP allows to pre-empt the address computation of any vector elements. Meanwhile, the vector processor does continue its own computation activity. The produced addresses are buffered. A scalar control unit (SCU) triggers actual data transfers when the vector processor is ready. Multi-buses of addresses and data (respectively composed of several buses of addresses and data) ensure the transfers.

Our study aims at associating our VAP to an array processor composed of a limitated number of PEs (of the order of 64). Pipelined processors still fit with the concepts presented here. The first part of this paper specifies the characteristics of the vector addressing processor. The internal architecture of the latter is then detailed. Lastly, a modular extension of this VAP is described.

Vector Addressing Processor specifications

The VAP deals with the access to contiguous vectors and sparsed vectors of limitated size. After briefly introducing our concepts, we discuss vector accesses in super-computer memories, as well as other interesting approaches. Then, the VAP as well as its architectural context are more precisely presented.

Contiguous vector accesses

Contiguous vector accesses are directly obtained by incrementing the allocation zone base address. For that purpose, the VAP is composed of several adders. These adders will make it possible to produce in parallel each vector element address, starting from the base address. All these produced addresses are buffered. An *end of activity* signal (EOA) is emitted by the VAP. At the actual release time of the memory access, each address is positioned on an address bus of the multi-bus of addresses. The VAP sends the signal of *memory request* (MREQ).

Sparsed vector accesses

An operation to access a sparsed vector is composed of (1) the index vector reading – a contiguous vector access, (2) a phasis of indirect address processing – effective addresses of vector accessed components. All these addresses are buffered. Then, vector element effective accesses are made like contiguous vector element accesses. Such an operation on a sparsed vector is a *gather* when reading, a *scatter* when writing (fig.3).

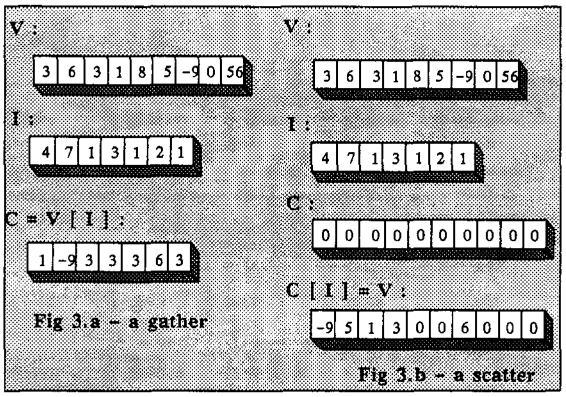

Figure 3 – Examples of gather / scatter

State of the art of gather / scatter processing

The super-computer traditional approach to carry out these operations consists in using the vector processor to realize the vector element address calculations. Furthermore, a vector indirect access V[D] requires three steps: index vector element loading D[i], element effective addresses processing V[D[i]], and vector element effective accessing V[D[i]]. The Cray-1 does not include gather / scatter operations in its instruction set. They must be realised by program [13]. Now, nearly all vector supercomputers provide instructions in their instruction set to realise such operations (Cray X-MP / 4 [5], Cray 2 [14], Cray Y-MP [7], Fujitsu VP-200 [11], Nec SX-2 [16], ...).

A totally different approach consists in representing sparse matrices with two lists: the list of selected items and a list composed of indices of these items in the matrix. Matrix composed of a great number of selected items are contiguously stored. This technic has been implemented in a specialized processor: the machine ESP at University of Edinburgh [9].

Unlike classical vector processors, our vector addressing processor works in parallel with the vector processor. It can prepare a vector access while the vector processor executes its computations.
The sparsed vector element addresses are obtained by adding the allocation zone base address to the index vector elements. This operation breaks up into the three following phases:

- *address computation of the index vector elements*: like contiguous vector accesses, these addresses are obtained by successive increment of the index vector base address. The produced addresses are buffered. The same circuit takes care of this computation and the contiguous vector addressing one.
- *memory access to the index vector elements*: the buffered addresses are positioned on the multi-bus of addresses (MBA). The VAP emits a request of reading towards the memory. It is put on standby. After positioning the data on the multi-bus of data (MBD), the memory emits a signal of end of activity to wake up the VAP.
- *actual address computation*: the VAP recovers the index values on the multi-bus of data. The allocation zone base address of the data vector is added to each read value. Thus, the VAP obtains the actual addresses of the vector elements. These addresses are buffered. The VAP signals its end of activity.

Then, the VAP is ready to realise the actual writing or reading.

Processing requiring successively the reading and the writing of the same vector can avoid the processing of effective addresses for writing. These ones have been latched by the VAP while reading.

Communication management

The cooperation between the VAP and the vector processor (VP) is controlled by the scalar control unit SCU (fig. 4). The SCU executes scalar operations and triggers VP activities for all vector operations. Furthermore, at the time of a vector memory access, it starts the VAP to compute vector component actual addresses. These addresses are buffered inside the VAP. The signal of end of activity is sent to the SCU. Then, for writing, the SCU triggers the transfer of the data from the VP to the MBD and then triggers the actual writing through the VAP. For reading, the SCU triggers the actual reading through the VAP. When the data are positioned on the MBD, the VAP transmits a signal of end of activity towards the SCU.

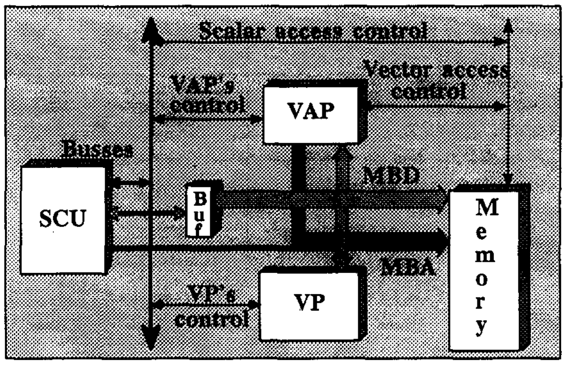

Figure 4 - Architectural model

Then, this latter can start the data transfer from the MBD to the VP (fig. 5). The VAP and the VP never communicate: any communication is taken in charge by the SCU. The communication protocols are based on an asynchronous model.

For a scalar memory access, the SCU uses one of the buses of the multi-buses of addresses and data. It starts the memory activity without VAP intervention. The data are accessed (read or written) by the SCU via a buffer (BUF) connected to the MBD. Therefore, the SCU can access a vector to perform vector processing with scalar result (reduction operations).

Vector operations concerned with internal transfers (of the following types: A [I] = B [J], A [I] = B, A = B or A = B [J] where I and J are index vectors), do not require computations, except vector component effective address computations. So, the vector processor has not to act on such operations. In our model, only the VAP acts on performing these operations, controlled, as usual, by the SCU. The source-vector element address computation is triggered by the SCU. An effective reading request is also sent by the SCU. Once the reading access have been performed, data are buffered inside the VAP. In a second time, target vector component effective address computations can be triggered. When these addresses are available, the SCU triggers the effective memory writing of the buffered data (in the VAP), via the MBD. While such a transfer, the vector processor continues its own activities. If the target-vector is scattered, indexes will circulate on the MBD. So, transferred data have to be memorised in an internal buffer of the VAP. This memorisation allows not to trigger writing right after reading. Other memory accesses, which are non-internal transfers, can be inserted between the reading and the writing.

The VAP's connection

In order to integrate the VAP, we must limit the width of the multi-buses according to the available external connection number. Pin array grid utilization allows multi-buses of four addresses and four data for a sole VAP. The VAP's pins are described in table 1 (fig. 6).

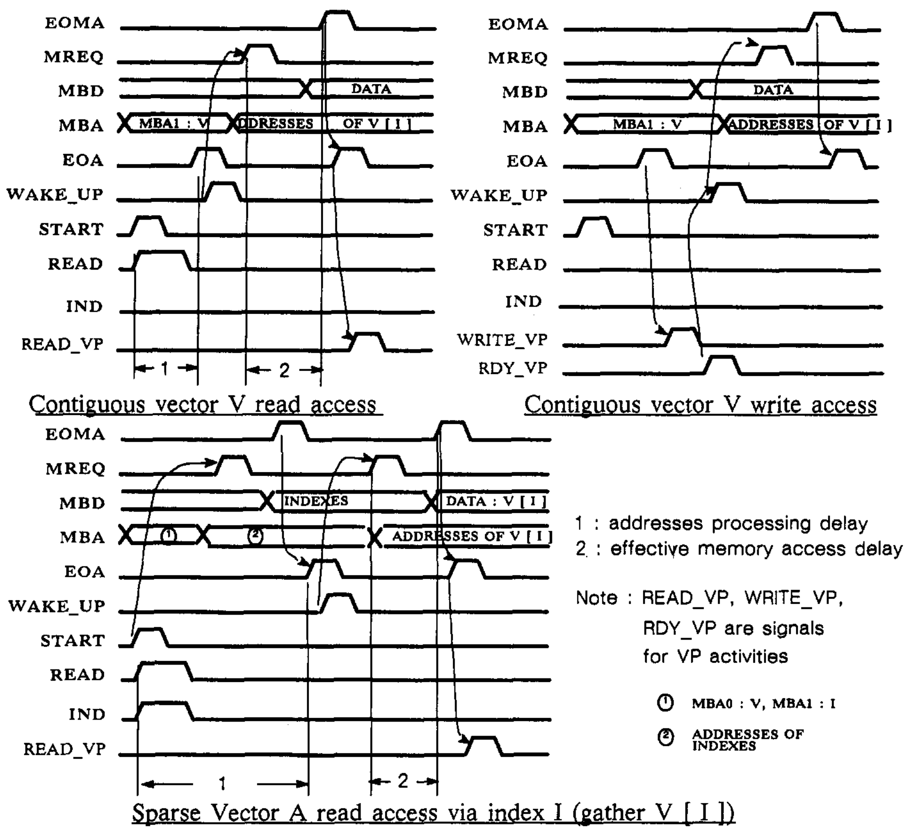
EOMA
MREQ
MBD
MBA
EOA
WAKE_UP
START
READ
IND
READ_VP
WRITE_VP
RDY_VP
DATA
MBA1 : V
ADDRESSES OF V [I]
INDEXES
DATA : V [I]
Contiguous vector V read access
Contiguous vector V write access
Sparse Vector A read access via index I (gather V [I])
1 : addresses processing delay
2 : effective memory access delay
Note : READ_VP, WRITE_VP, RDY_VP are signals for VP activities
① MBA0 : V, MBA1 : I
② ADDRESSES OF INDEXES

Figure 5 - Communication protocols

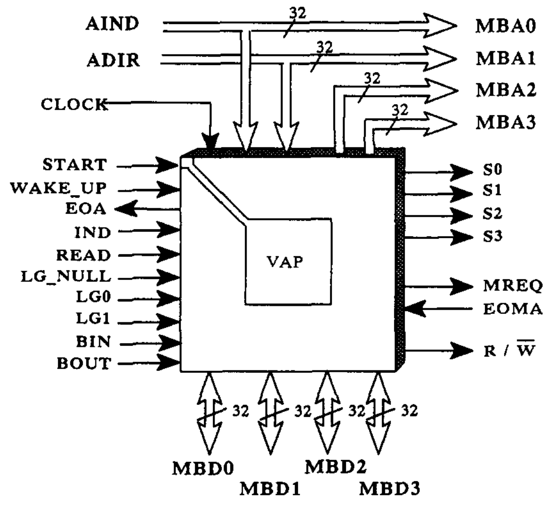

Figure 6 - The VAP chip

The VAP's internal architecture

This part describes the different functionnal units of the VAP.

The VAP is mainly composed of three active units (fig. 7): the control unit (CU), the direct address processing unit (DAPU) and the indirect address processing unit (IAPU). The VAP also includes two storage units: a buffer of data (BD) and a buffer of addresses (BA).

- Control unit:
  It receives the different communication signals from the SCU and the memory. It schedules the VAP internal activities according to the following input signals: LG_NULL, READ, BIN, BOUT, START, WAKE_UP, IND. It also emits the end of activity signal (EOA) to the SCU and the memory request signal to the memory (MREQ).
  According to the length of the processed vector, it activates the adequate validation signals of the MBA buses (S0 ... S3). It synchronizes all the VAP's internal activities. It triggers the loading of the BA with the

START	triggers the actual address computation
WAKE_UP	triggers actual address transfers on the MBA and then activates the memory with a MREQ signal
EOA	after a START: signals computation; after a WAKE_UP: signals the end of activity of the memory
READ	read / write selection signal
IND	direct / indirect addressing selection signal
LG_NULL	validates LG0, LG1 inputs
LG0, LG1	length of vector - 1
BIN	loading of data from MBD to the BD when EOMA will be received
BOUT	loading of MBD from BD when MREQ will be emitted to memory
S0 .. S3	validation of multi-bus buses: Si validates MBAi and MBDi
MREQ	triggers the memory activity
EOMA	signals the end of memory activity
AIND (= MBA0)	input: allocation zone address for a sparsed vector access; output: first MBA's bus
ADIR (= MBA1)	input: base address for a contiguous vector access; output: second MBA's bus
MBA2, MBA3	buses 3 and 4 of MBA
MBD0 .. 3	buses of MBD
R / W	read / write signal to the memory
CLOCK	clock signal of the VAP

Table 1 - VAP's signals

addresses computed by the DAPU or the IAPU.

- DAPU: Direct Address Processing Unit:
  It computes the element addresses for a contiguous vector and the index vector element addresses for a sparsed vector. When the CU receives the START signal, the address ADIR available on MBA1 is latched in the DAPU. Then, three adders compute in parallel the three addresses: ADIR + 1, ADIR + 2, ADIR + 3. The resulting addresses are respectively positioned on the buses $MBA_{int}1$, $MBA_{int}2$ and $MBA_{int}3$. ADIR is positioned on $MBA_{int}0$.
- IAPU: Indirect Address Processing Unit:
  It computes the actual addresses for a sparsed vector. Once the CU receives the EOMA signal, the four indexes (previously positioned on the MBD from the memory) are latched in the IAPU. Four adders compute the four actual addresses in parallel: the allocation zone base address (AIND) is added to the four latched indexes. These four resulting addresses are placed on the MBA_{int}.
- BA: Buffer of Addresses:
  This buffer memorizes the computed addresses present on MBA_{int}. The CU controls loadings and unloadings of this buffer.
- BD: Buffer of Data:
  This buffer memorizes the content of the MBD. The loadings and unloadings of this buffer directly depend on the signals BIN and BOUT.

The VAP's modular aspect

The chip current technology and the connectic have limitated the multi-bus width to 4. To achieve a higher level of parallelism, larger multi-buses are required. By clustering several VAPs, the size of the handled vectors are no longer limitated to four. The strategy to cluster several VAPs will now be described. Afterward, an iterative method to build "clusters of clusters" is introduced.

Principal

Each VAP takes in charge the computation of addresses required to access a vector slice. Multi-buses MBA and MBD are obtained by clustering MBAs and MBDs of the VAPs. Each VAP of a cluster uses certain buses of the multi-buses MBA and MBD. Communication protocols between the VAP and the other units have to be extended to a cluster. In order to keep this interface, some auxiliary circuits are connected to the inputs and outputs of the VAPs of the cluster. Some of these circuits (Address Distributor) split the vector addressing over the VAPs. Other ones (Synchronization Centralizer) take in charge the synchronization between the SCU or the memory and the VAPs.

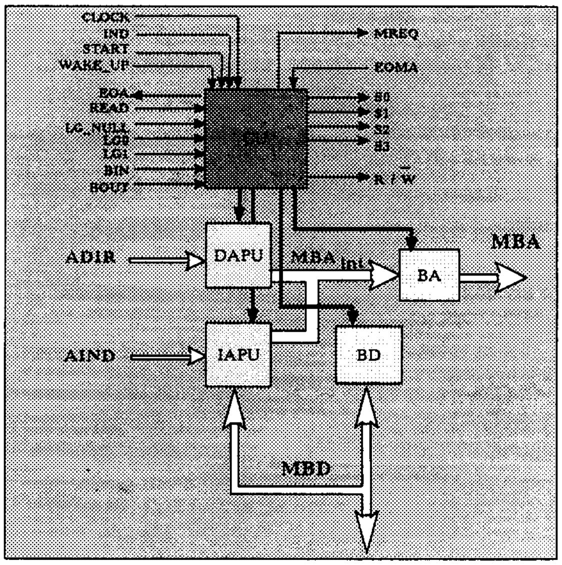

Figure 7 - VAP architecture

The address distributor - AD

An AD chip (fig. 8) splits a vector addressing into four sub-vector addressings. A contiguous vector is addressed via a couple of informations, base address and vector element number. Given this couple, the address distributor generates four couples to address the four contiguous sub-vectors.
A sparsed vector addressing uses the address distributor exclusively to split the index vector. The vector allocation zone address is directly sent to VAPs by the SCU to compute the indirect addresses. Each VAP receives a triplet constituted by the base address, the index vector element number provided by a address distributor, and the allocation zone base address, provided by the SCU.
Some sub-vectors produced by the address distributor may be zero-length. The initial vector is splitted in slices of constant length, rather than four slices of same length (fig. 9).

The activity scheme of the address distributor is the following:

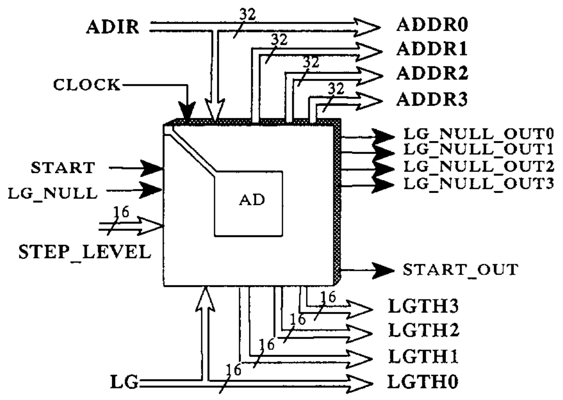

Figure 8 - The Address Distributor chip

AD's input signals	
START	beginning of AD's activities
ADIR	contiguous vector allocation zone base address
LG	vector length
LG_NULL	LGTH invalidation
STEP_LEVEL	incrementing value
AD's output signals	
ADDR0 .. 3	computed address buses
LGTH0 .. 3	computed length buses
LG_NULL_OUT0 .. 3	LGi invalidation
START_OUT	starts depending activities

Table 2 - AD's signals

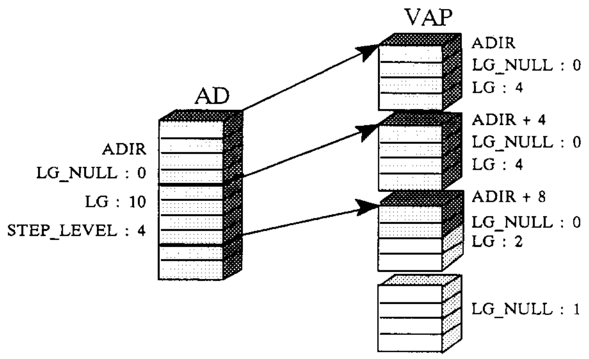

Figure 9 - Example of a vector splitting into four sub-vectors

The signal START releases the address distributor activity. This signal indicates the availability of an address and a length on the buses ADIR and LG

Then, base addresses and lengths of the four sub-vectors are computed. For that purpose, STEP_LEVEL bus inputs a constant value. It is used as an increment value to produce the base addresses of the four sub-vectors. This value also specifies the maximal length of the four produced sub-vectors. These computed addresses and lengths are placed on the four couples of output buses (ADDR*i*, LGTH*i*). Each bus couple links the address distributor to a given VAP.

As previously seen, some generated sub-vectors may be zero-length. For this raison, four output signals (LG_NULL_OUT*i*) respectively validates the lengths of the four sub-vectors (LGTH*i*). For example, if the initial length (LG) is less than 3 x STEP_LEVEL, LG_NULL_OUT3 is active, invalidating the length vehiculated by LGTH3, ... Furthermore, the signal LG_NULL which selects the address distributor invalidates, when active, the four lengths LGTH*i*. In this case, the four signals LG_NULL_OUT*i* becomes active to indicate the invadility of the data which are present on output buses.

Once all the signals are positionned on output, START_OUT signal triggers activities associated to each sub-vector.

In order to optimize the pin number, the ADR0 and LGTH0 buses vehiculates ADIR and LG at the beginning of circuit activity (START).

The Synchronization Centralizer - SC

This circuit is used to manage communications of a VAP-cluster and the memory or the SCU.
It emits a signal when four input signals have been received. On one hand, it centralizes four MREQ signals to trigger the memory activaty. On an other hand, it centralizes four EOA to signal their end of activity to the SCU.

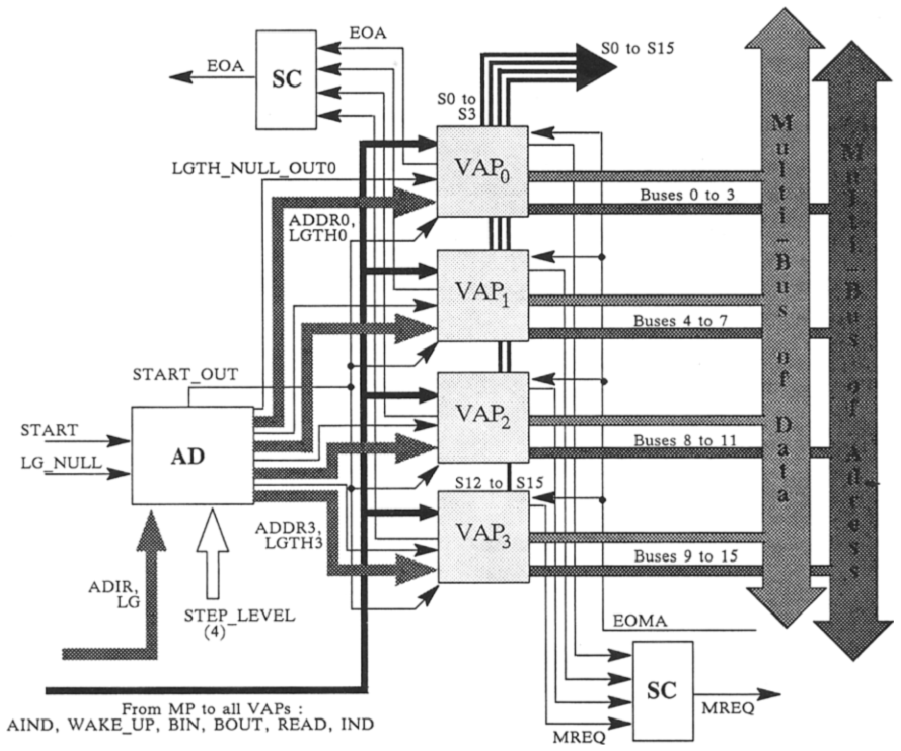

Figure 10 – A VAP-cluster (made of 4 VAPs)

Multi-level modular architecture

A address distributor associated to four VAPs produces a VAP-cluster with a 16 buses width multi-buses (fig. 10). The four VAP-cluster association driven by a fifth AD circuit produces a VAP-cluster of width 64. By iteration, a VAP-cluster of width 4^n can be created with four VAP-clusters of width 4^{n-1} and an AD circuit. In this case, the arborescence of AD circuits realizes the distribution of address computations. Each leaf of this arborescence is connected to four VAPs.

A similar arborescence of SC circuits centralizes all the EOA signals. A second arborescence of SC circuits centralizes the MREQ signals. All signals, which are not transmitted either by a SC or by an AD circuit, are directly emitted to any VAP. The validation signals S0 .. S3 of each VAP represents a part of signals S0 .. Sn of the VAP-cluster.

Conclusion

We have presented a specialized processor in vector accesses (VAP-cluster). This processor is build around three different circuits. It realises the addressing of contiguous and sparsed vectors of 4^n elements. The address computations can be executed in parallel with the vector processing activities.

A simple modification of the VAP would allow the addressing of sparsed vector whose elements are uniformly distributed according to a stride. This type of access is available in DEVIL as well as in the majority of vector processors (e.g. Cray-2, Y-MP, Japanese machines). This kind of access is similar to the contiguous vector addressing. Instead of successively incrementing the base address, we add successively the stride to the base address.
Our processor can not realize bit-vector controlled vector addressing, like compress/extend operations. A second VAP generation should allow this kind of vector accesses.

Bibliography

[1] R. ALLEN, K KENNEDY, Automatic Translation of FORTRAN Programs to Vector Form, ACM TOPLAS, vol 9, n°4 (October 1987) pp. 491-542.

[2] A. ASTHANA, H.V. JAGADISH, J. A. CHANDROSS, D. LIN, S. C. KNAUER, an "Intelligent Memory System", Computer Architecture News, Vol 16, N 4 (september 1984), pp 12-20.

[3] P. BOSE, Interactive Program Improvement via EAVE: an Expert Adviser for Vectorization, Proceedings of the 1988 International Conference on Supercomputing (St Malo, France, July 1988) pp. 119-130.

[4] Control Dat Corporation, FORTRAN 200 Reference Manual, Pub. 60480200 (1986).

[5] Cray Research Inc., Cray X-MP Computer System Functional Description, Pub. HR-3005 (Cray Research Inc., Mendota Heights, MN 55120, USA).

[6] Cray Research Inc., CFT 77 Reference Manual, Pub. SR-0018 1986 (Cray Research Inc., Mendota Heights, MN 55120, USA).

[7] Cray Research Inc., Cray Y-MP Computer Systems Functional Description Manual, Pub. HR-4001, 1988 (Cray Research Inc., Mendota Heights, MN 55120, USA).

[8] J.L. DEKEYSER, Ph. MARQUET, Ph. PREUX, EVA: an explicit Vector lAnguage-an Alternative Language to Fortran 8x, Internal Pub. n°ERA 90-78 of the Laboratoire d'Informatique Fondamentale de Lille (February 1990).

[9] R.N. IBBETT, T.M. HOPKINS, K.I.M. McKINNON, Architectural Mechanisms to support Sparse Vector Processing, Proceedings of the 16th Annual International Symposium on COMPUTER ARCHITECTURE (May 1989), pp 64-71.

[10] A.H. KARP, R.G. BABB II, A comparison of 12 Parallel Fortran Dialects, IEEE Software, September 1988, pp. 52-67.

[11] K. MIURA, K. UCHIDA, FACOM Vector Processor VP-100/200, Proceedings of the NATO Advanced Research Workshop on High Speed Computing, Jülich, W. Germany, 20-22 June 1983

[12] R.H. PERROT & A. ZAREA-ALIABADI, Supercomputer Languages, ACM Computing Surveys, vol 18, n°1 (March 1986) pp. 5-22.

[13] R. W. HOCKNEY, C. R. JESSHOPE, The CRAY-1, in: Parallel Computers (Adam Hilger Ltd, Bristol, UK, 1981), pp. 69-95.

[14] W. SCHÖNAUER, The Cray-2, in: Scientific Computing on Vector Computers (Mc Graw-Hill, 1987).

[15] Cray Research Inc., Vectorizing C Compiler for Cray-2 Computer Systems, Pub. SN-2061, 1987 (Cray Research Inc., Mendota Heights, MN 55120, USA).

[16] T. WATANABE, H. KATAYAMA, A. IWAYA, Introduction of NEC Supercomputer SX System, in: SUPERCOMPUTERS, Class VI Systems, Hardware and Software (North-Holland, 1986)