OPAC: a cost-effective floating-point coprocessor

André Seznec, Karl Courtel

INRIA

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1461

Programme 1
Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

OPAC : A COST-EFFECTIVE FLOATING-POINT COPROCESSOR

André SEZNEC
Karl COURTEL

Juin 1991

RR-1461

IRISA INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTEMES ALEATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CEDEX
FRANCE
Téléphone : 99.36.20.00
Télex : UNIRISA 950 473F
Télécopie : 99.38.38.32

OPAC : a cost-effective floating-point coprocessor*

André Seznec Karl Courtel
IRISA,Campus de Beaulieu
35042 Rennes Cedex, FRANCE
Tel: 99362000
e-mail : seznec@.irisa.irisa.fr

Le coprocessor numérique OPAC

Publication Interne n° 586 - Mai 1991 - 26 Pages
Programme 2

Résumé

Les performances effectives des microprocesseurs RISC sur les applications numériques restent décevantes malgré des performances nominales de crête parfois impressionnantes. Dans ce papier, nous montrons que la pauvreté de ces performances est due à l'adressage statique des registres dans les microprocesseurs. Nous présentons l'architecture du coprocesseur OPAC où des mémoires FIFOs sont utilisées pour ranger les résultats intermédiaires et les opérandes réutilisables. Des performances proches d'une multiplication-accumulation par cycle sont atteintes sur un large spectre d'algorithmes.

Abstract

The *effective* performance of RISC microprocessors on numerical applications remains on the order of a few megaflops/s. In this paper, we show that the static addressing of the registers in standard RISC floating-point coprocessors is one of the main bottleneck for performance. In the architecture of the coprocessor OPAC, we propose an alternative using only FIFO queues for storing intermediate results

*This work was partially supported by the french Ministry of Defense under grant DRET-INRIA No 88.34.191.00.470.75.01 and by the coordinate program of CNRS PRC-AMN

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(U.R.A. 227)
UNIVERSITE DE RENNES I I.N.S.A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(UNITE DE RECHERCHE DE RENNES)

and reusable operands. Performance close to multiplication-accumulation every cycle is expected on a large set of numerical applications at a reasonable hardware cost.

Mots-clés :

numerical applications, floating-point coprocessor, dynamic addressing, FIFO queue

Introduction

In 1987-88, we observed the rise of very powerful workstations, for example those built around RISC microprocessors. On the other hand, powerfull floating-point chips with cycle around 30 ns were off the shelf. Nevertheless, the *effective* performance of these workstations on scientific applications were still of the order of a few megaflops/s even on optimized primitives.

In this paper, we point out that the classical archetype RISC solution relying upon a large set of floating-point registers for exploiting data locality in compute-bound primitives had severe limitations. In section 3, we present the architecture of a prototype floating-point coprocessor OPAC. OPAC has been devlopped at IRISA since the end of 1987. FIFO queues are used for storing intermediate results and reusable operands : these FIFO queues are not used as interface buffers, but as memorie which are implicitly written and read with stride one. OPAC has been designed to perform very efficiently a set of numerical primitives including the whole library BLAS LEVEL 3, FFTs, .. ; the coprocessor OPAC can also execute a larger set of applications and may be used as a classical floating-point coprocessor.

Simulation results have shown that the peak performance of one floating-point multiply-add per cycle expected on the prototype under development will be approached on a large set of *effective* applications such as solving dense linear systems, dense eigenvalue or singular value problems or computing FFTs or correlations, ...

The prototype is built with off the shelf chips which were available in 1988 and a customized VLSI sequencer. But it must be pointed out that a one-chip VLSI implementation of an OPAC-like coprocessor may be possible : it represents approximatively the same hardware complexity as the Intel i860.

1 A set of useful computing primitives

The performance on numerical applications must be accessible through standard high level languages (Fortran or C) or through numerical libraries which ranges from low level kernels (for instance BLAS LEVEL 3 [Do88]) to sophisticated algorithms

(for instance LAPACK [An90]). Low level primitives are used to shield the user from the hardware complexity of the architecture. Here we list some interesting primitives.

1.1 Matrix multiplication and the BLAS LEVEL 3 library

As most numerical computers reach their best performance (in terms of Mflops/s) on matrix multiplications, there has been a particular effort to express linear algebra algorithms in such a way that most of the computations is carried in matrix multiplications [Ja86]. In fact, most of the linear algebra algorithms on dense matrices can be rewritten in block algorithms; here we listed some of these applications :

- Direct methods for solving linear systems : Gauss method, Gauss-Jordan method, LU decomposition, Cholesky decomposition, ..
- Orthogonalization algorithms : Gram-Schmidt method, Polar decomposition via an iterative method [Ph87], ..

It is noteworthy that an efficient matrix decomposition may be useful also for operations on banded matrices and even in a few methods for solving sparse linear systems [Ch87][Ol80].

The BLAS LEVEL 3 [Do88] library also contains a few other very useful primitives such as multiplication of full matrix by a triangular matrix (or its opposite); some other matrix operations can also be interesting such as operations on banded matrices. Most of the subroutines of LINPACK [Do79] and EISPACK have already been rewritten in order to encapsulate the major part of the computations in calls to routines of BLAS LEVEL 3 [An90].

1.2 Fast Fourier Transform

Fourier transform is one of the most popular methods in signal processing and is very efficiently implemented by the FFT algorithm [Co65], nevertheless the FFT algorithm remains compute-bound : 2^n complex data read and written, 2^{n-1} complex coefficients referenced and $5n * 2^n$ floating point operations executed. Moreover, in

many cases, the Fourier transform has to be applied to a set of vectors : coefficients may be read one time, then the asymptotic average number of floating point operations per memory access is $5n/4$.

As other useful primitives, one can also enumerate convolutions and derived algorithms, polynomial evaluation (on a single element or on a vector).

2 Needs for intermediate storage in floating-point coprocessors of RISC microprocessors

Floating-point operator chips which are able to deliver the result of a floating point multiplication/accumulation on every cycle have become commercially available[1]

From now, we consider a floating-point coprocessor of a RISC microprocessor built around such elements. We analyze the size and the kind of intermediate storage which is needed in order to achieve performance close to one multiplication/addition on the primitives defined in section 1; the multiport register file generally used in standard floating-point RISC coprocessors is shown to be unsufficient to ensure performance close to one floating-point multiply-add per cycle (FMA).

Effective memory bandwidth in a RISC microprocessor

On processors designed around RISC microprocessors, the effective memory throughput is relatively low; generally one cannot hope to obtain more than one floating-point data every five cycles as illustrated in the following example :

```
   DO 10 I=1,N
10 A[i]= A[i]+ B*C[i]
```

The loop body may be coded as follows:

1. R0= R0 +R1 % R0 is the address of C[i] , R1 is the storage stride of C
2. LOAD R0 F1

[1]Since the project has been started in 1987, microprocessors with on chip floating-point coprocessors have appeared

3. F2 = F0 * F1 % B has been loaded in F1 outside the loop
4. R2= R2 + R3 % R2 is the address of A[i] , R3 is the storage stride of A
5. LOAD R2 F3
6. F4 = F2 +F3
7. STORE R2 F4
8. R4 = R4 - R5 % decrementation of the loop index
9. if (R4!= 0) JUMP to 1.

Let us consider that as in SPARC implementation, all the instructions cost one pipeline cycle except for the LOAD (2 pipeline cycles) and the store (3 pipeline cycles)[2]. Then the sequencing of this loop will cost 13 cycles (when there is no cache miss).

Limitations of the RISC model

We suppose that the floating-point coprocessor is associated with a RISC microprocessor which is able to deliver it only one memory word of data every five cycles.

Through the example of the FFT on a 2^n elements vector, we analyze some limitations of the standard RISC floating-point coprocessors based on specialized register sets.

For computing a FFT on a vector of 2^n points , 2^n complex elements must be read and 2^n complex results must be stored in memory; at least $4*2^n$ memory words must be accessed for computing a FFT on 2^n elements and $3n * 2^n$ floating-point adds and $2n * 2^n$ floating point multiplications must be computed : $n * 2^{n-1}$ FFT butterflies.

If the processor can deliver (or receive) one data every five cycles and if no extra memory access is executed, at least $5 * 4 * 2^n$ cycles are needed to perform the memory accesses involved in a FFT on 2^n points : then, in these conditions, the floating point adder can not be saturated if $3n * 2^n <= 20 * 2^n$ i.e $n < 7$ or $2^n < 128$.

[2]for simplicity, we consider that the conditionnal jump costs also 1 pipeline cycle

Then, implementing a performant FFT primitive would require a huge amount of registers (approximatively $3 * 2^n$) for all sizes of 2^n: FFT on a large vector may be decomposed in a two-dimensionnal FFT on smaller vectors, but to saturate the floating-point adder, intermediate vector results must not be intermediately stored in the memory if 2^n does not exceed $128 * 128 = 16K$.

The size of the register set used in standard floating-point coprocessors of RISC processors is limited (generally less than 64 registers). The main trouble for standard RISC floating-point coprocessors is not so much the size of the needed register file (we can reasonnably expect that a multiport register file of a few thousands words will be commercially available in a few years); the effective difficulty is due to the static addressing of the registers : each register must be explicitly referenced in every instruction working on it, then it seems that the whole FFT must be unrolled and coded as a sequence : volume of code will exceed $3n * 2^n$ instructions (more than 30K words for a FFT on 1024 elements).

Moreover a different version of the FFT primitive must be written for each value of n.

This example clearly proves that using a standard register file as *only* intermediate storage support in a floating-point coprocessor attached to a RISC microprocessor does not enable to reach performance close to one floating-point multiplication/accumulation every cycle on effective applications : we need a few thousands words of intermediatememory support (an analog analysis for a square matrix updating show that we need more than 800 words) and we need some dynamic addressing of this support.

Synchronous sequencing of the processor and the coprocessor

In standard RISC microprocessors, the processor and the floating-point coprocessor are synchronously sequenced. This may induce an important performance decrease due to caches misses, TLB exceptions, pages faults, ... : the sequencing of the coprocessor is stopped even when the data on which it had to work are present in its file register.

Synthesis

If the effective data throughput of the microprocessor remains low, some relatively large local memory has to be used in the floating-point coprocessor in order to obtain performance close to one FMA per cycle. We have also observed that some kind of dynamic addressing must be provided in order to be able to efficiently use this memory in parameterized numerical kernels.

3 The architecture of OPAC

The OPAC prototype will be interfaced with a standard MIPS R2000 microprocessor (fig.1). FIFO queues are used for buffering operands, results and calls for numerical kernels between the microprocessor and the processor. The sequencing of OPAC and its host will be completely asynchronous. This structure may be compared with the model of "Decoupled Access Execute" [Sm82] : the Execution unit and the memory Access unit are decoupled; in our machine, we decouple the floating-point unit and the rest of the processsor.

3.1 Architecture of the computation block

The architecture of the computation block of the coprocessor OPAC has been studied in order to deliver the result of one multiplication/accumulation per cycle on the set of primitives listed in section 1. Only commercially available chips are used in the computation block. Then to approach our performance objective, the computation block is heavily pipelined, each element in it is supposed to be able to deliver one result per cycle.

Studying these primitives has lead us to the architecture presented in fig.2.

Optimized data paths

In the set of compute-bound primitives which have been listed in section 1, in most cases, the result of a multiply is then accumulated with a previously computed intermediate result. So we have chosen to implement a direct *and mandatory* path

Figure 1: Coprocessor OPAC in its microprocessor environment

tpx, tpy, tpo, tpl : interface FIFO queues

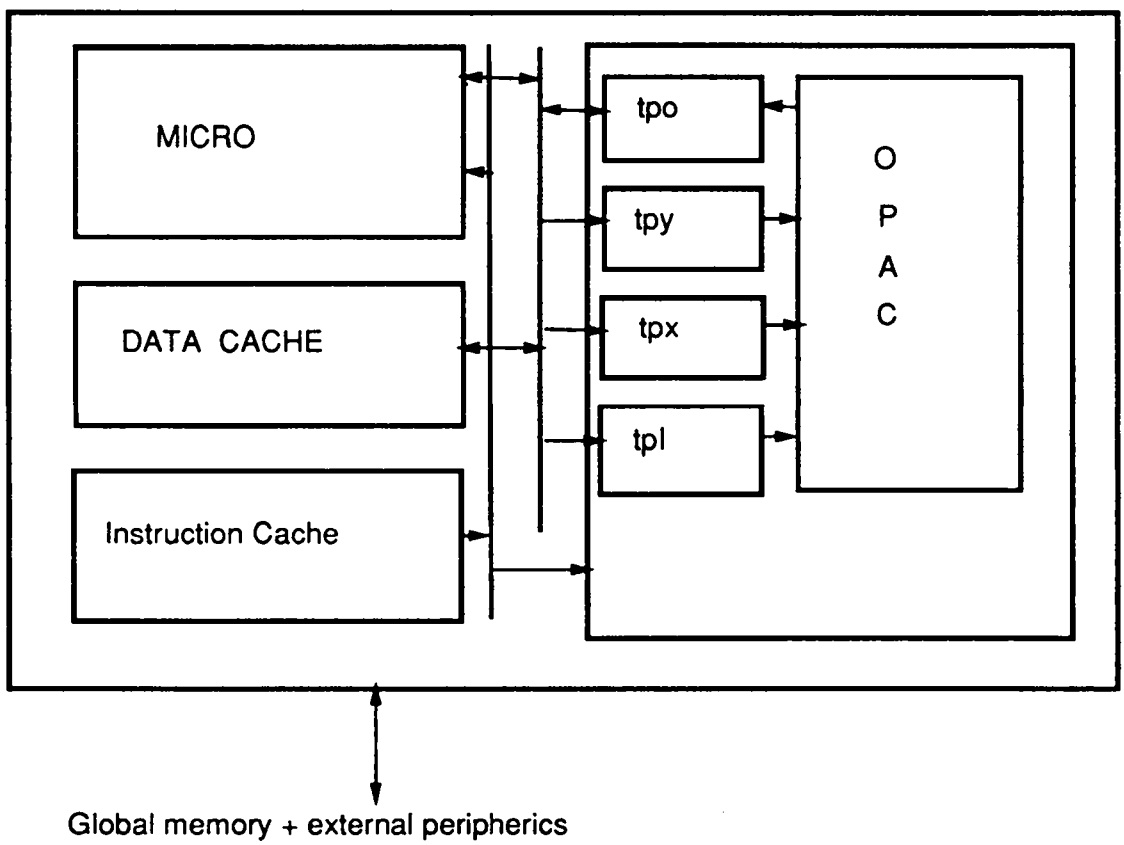
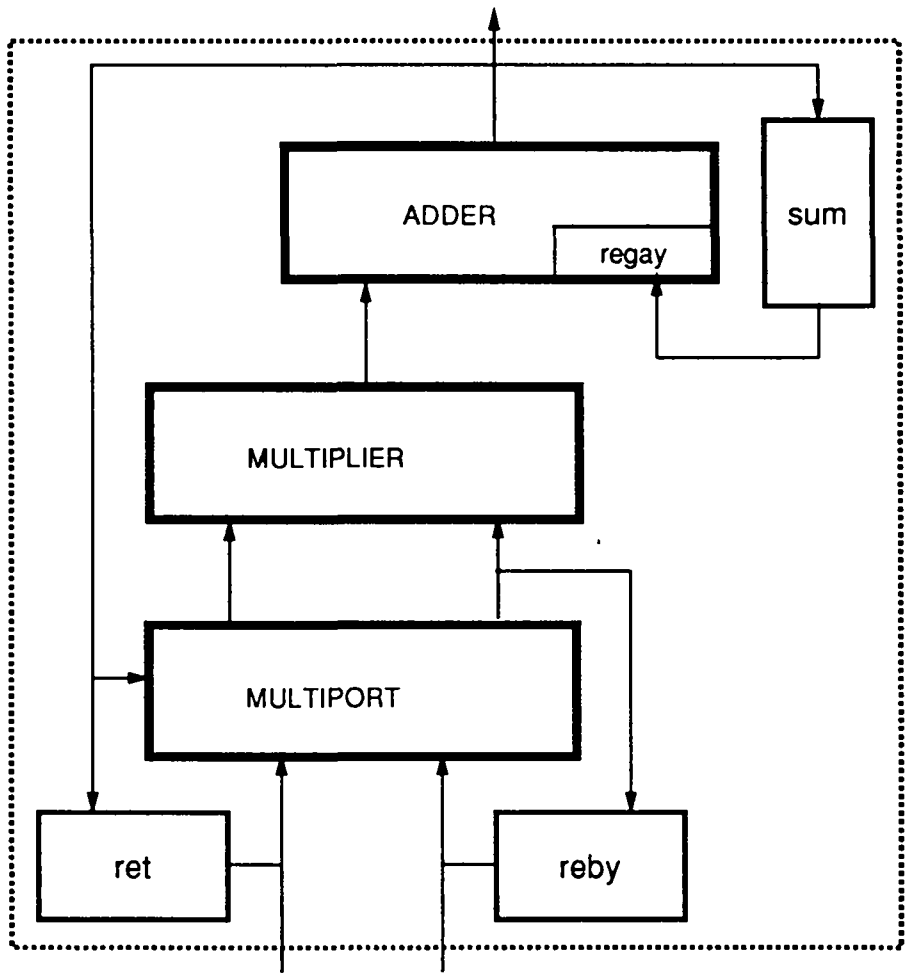


Figure 2: Architecture of the coprocessor OPAC



sum, ret, reby : FIFO queues

from the output of the multiplier to one of the input of the adder[3].

On the other hand, there is no direct path from the input FIFO queues of OPAC to the floating-point adder, because the ability of passing a data through the multiplier without change has been judged sufficient for an efficient implementation of the defined kernel of primitives : adding two operands coming directly from memory is very rare in compute-bound primitives. The scalar addition of two elements can be implemented as follows : the first operand is passed through the multiplier and is recycled on entry y of the floating-point ALU through a direct internal path, the second operand is then passed through the floating-point multiplier and added to it.

Local memorization

As pointed out in section 2, some local and reasonably large (at least several thousands of words) physical support is needed to store intermediate results and reusable operands. We have also shown that static addressing of this memorization support is not a competitive approach, because it does not allow to implement efficient primitives with variable parameters.

We have decided to use FIFO queues for 6 major reasons :

- For each primitive defined in section 1, we have been able to find an implementation where these FIFO queues are used as only local storage support and where no extra external memory access is performed.
- As the microcode has only to contain the READ and WRITE information on the different queues used in the design, then the microcode is quite compact : no explicit information are needed for address generation.
- We shall see in section 3.2 that the use of FIFO queues facilitates the microcode generation and decreases the microcode volume (in number of instructions).
- In vector sections, the difference between a vector register and a FIFO queue

[3]This approach has become very popular since the project was started (IBM RS6000, Intel i860, ..)

is very tiny; but sometimes the use of FIFO queues may be easier because it is possible to dissociate consecutive elements of a FIFO queue (for example when solving a triangular system, at each step of the outside loop, the size of the vector is decremented, and the computation on the first element is different from the computations on the other elements).

- Reasonnably large FIFO queue RAMs were now available in 1988 (at least 2048 words of 9 bits) with fast access times (25 MHZ) and correct characteristics : a data written at t may be available on the output at $t + 80ns$. An decrease of the period and an increase of the capacity has been observed : 50 Mhz 8 Kwords FIFO queues are now available.

- In terms of hardware, the use of FIFO queues is a quite cheap solution : an alternative was the use of dual-port registers:

  1. Static addressing would lead to the difficulties described in section 2.
  2. Addresses are computed during the execution; but this costs a lost of hardware : address generators, data paths for initializing addresses, microcode RAMs for controlling these, etc.

The locations of the three FIFO queues *sum*, *ret* and *reby* are justified by the study of the algorithms listed in section 1. FIFO queues *sum* and *ret* have been introduced to store intermediate results flowing out from the output of the adder respectively to the second entry of the adder and the entries of the multiplier. The FIFO queue *reby* has been introduced to store vectors (or matrices) of data which are used several times as an operand for a multiply.

Synthesis

We have checked that the structure of the computation block allows to compute the primitives defined in section 1 for reasonably large size of parameters without extra memory access; for larger parameters, the whole set of data cannot be intermediately stored in an internal FIFO queue. The primitives have to be splitted into calls to basic kernels working on smaller sets of data (see section 4 for instance).

For enabling the computation block to reach performance close to one multiplication/accumulation by cycle, standard scalar instructions are not sufficient : one instruction is needed at each cycle, so that the processor would not be able to feed the coprocessor in instructions.[4].

At least vector instructions would be needed in order to reach correct performances : a single chip controller would be quite complex. Moreover trivial vector instructions are not be sufficient to implement many interesting primitives (for instance executing arithmetic on complex data or performing the perfect shuffle). Therefore we have decided to use horizontal microcode for controlling the computation block .

3.2 An efficient pipeline management

3.2.1 Horizontal microcoded control

Many existant pipeline machines are controlled via horizontal microcode. The FPS 164 produced during the beginning of the eighties by Floating Point Systems is a popular machine which is representative of this family [Ch81]. In horizontal microcoded machines, distinct FUs may be used in parallel; the instruction contains one instruction parcel for each functional unit; at each cycle, each FU receives a new control parcel. This structure of control allows to initiate a new sub-instruction on each functional unit at each cycle : the classical bottleneck to feed the functional units in instructions does not exist on this family of pipeline machines.

In the OPAC coprocessor, microcode is stored in RAM cells; a customized VLSI sequencer is used to sequence this microcode.

Now we recall some difficulties which are inherent with classical horizontal microcoded machines; then we present an original and very efficient hardware management of the pipeline in the coprocessor OPAC.

[4]In the Intel i860, this problem is addressed by the ability of issuing two instructions every cycle, one instruction for the RISC processor and one instruction for the floating-point coprocessor

Figure 3: Activity in the different FUs on a vector multiply

T	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
RFIFO	1	2	3	4	5	6	7	8	9	10								
REG		1	2	3	4	5	6	7	8	9	10							
MUL1			1	2	3	4	5	6	7	8	9	10						
MUL2				1	2	3	4	5	6	7	8	9	10					
ADD1					1	2	3	4	5	6	7	8	9	10				
ADD2						1	2	3	4	5	6	7	8	9	10			
WFIFO							1	2	3	4	5	6	7	8	9	10		

Labels in each case corresponds to the index of the iteration concerned by the instruction currently executed

MUL1, MUL2 (resp. ADD1, ADD2) are the stages of the multiplier (resp the ALU).

3.2.2 Classical difficulties on classical horizontal microcoded pipelines

As an example, we consider the multiply of two vectors of 10 elements on a very simple model of the coprocessor OPAC : we suppose that all functional units are passed through in one cycle.

Activities of the different elements are represented on fig.3.

On this example, one can notice that three phases can be isolated in the execution :

- Phase 1 : filling the pipeline (from cycle 1 to cycle 6) item Phase 2 : steady state phase (from cycle 7 to cycle 10) :
  at cycle i
  - the FIFO queues works for iteration i
  - multiplexers work for iteration i-1
  - the register file works for iteration i-2
  - the multiplier works for iteration i-3
  - the adder works for iteration i-4
- Phase 3 : emptying the pipeline (from cycle 11 to cycle 16)

If we use classical horizontal microcoded control, then we have to generate three distinct sections of microcode for these three sections. This can be done by software [Ra81][Ei88].

The main trouble here comes from a number of iterations smaller than the minimum number required to reach the steady state phase of the pipeline. If we want to implement primitives which can run for any parameters value, these parameters must be tested and different code branches must be generated for the different cases :

1. Latency of the execution of primitives is increased (test of the parameters and conditional branch at entry of the primitive) : performance for small size of problems will be bad, particulary on primitives consisting in a single vector instruction.

2. Volume of the microcode may become huge.

3. The sequencer must be able to test different values of the parameters.

3.2.3 An elegant hardware management of the pipeline

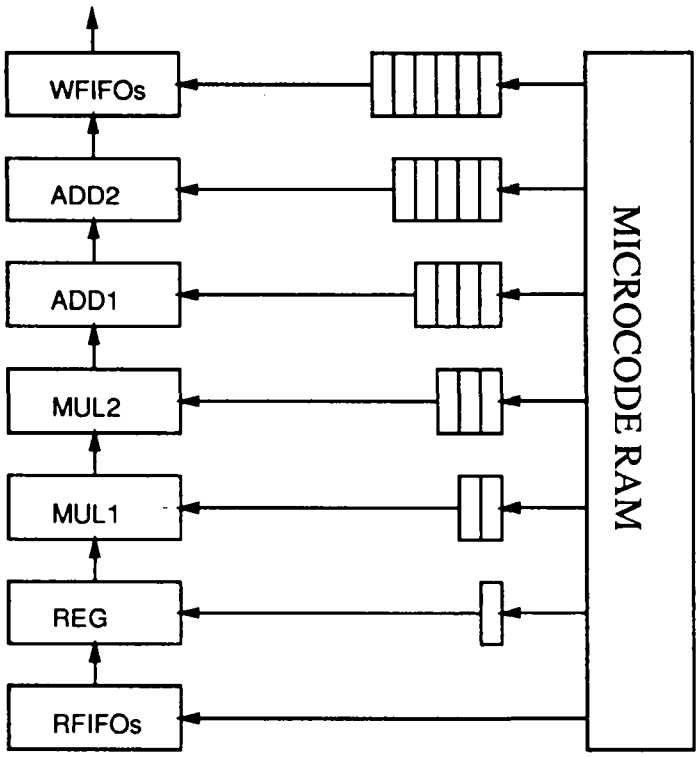
In a previous paper [Je86], we have presented the DSPA model which allows to avoid the previous problem on a pipeline processor designed for general scientific applications; this solution was developped in order to obtain performance on a very large spectrum of numerical algorithms including sparse processing. Unfortunately the hardware implementation of the DSPA model seems quite expensive. For the particular structure of OPAC, we have derived a cheap hardware solution which allows a very simple management of the pipeline.

Let us remark first that the chaining insides the computation block in OPAC are automatic : there is a maximum of one data path from a point in the computation block to another point with the restriction that the path does not cross a FIFO queue [5].

Then if we suppose that the different delays to crossing the elements of the computation block are constant, the delay separating the contributions of two distinct functional in the computation of a single result flowing out from the adder is constant : for example, if we consider the model treated in fig.3, at T+3, the multiplier acts on data which have flown from the FIFO queues tpx and tpy at T.

[5] except the path from the output of the floating-point ALU to the multiport register file

Figure 4: Principles of the delayed microcode control



A natural hardware solution for managing the pipeline is then as follows :

A single word of intruction contains instruction parcels for the distinct elements in OPAC, but all these parcels correspond to a single flow of data. When the address in the microcode flows out from the sequencer, the different parcels of the microcode are delayed so that the control flow and data flow reaches the different elements at the same cycle; this is illustrated fig.4.

If we consider the example illustrated in fig.3, all the activations of the elements corresponding to anyone of the iterations are coded in the same instruction word, but then they are differently delayed : then the instruc-

tion parcels which are attacking the different FUs in fig.4 correspond exactly to the column 5 in fig.3.

We call this hardware management of the pipeline : *delayed microcode control.*

Square root or division are generally longer operations than multiplications on a floating-point multiplier. Nevertheless, the "delayed microcode control" allows to manage these operations, provided that there is only one such operation in progress in the pipeline of the multiplier.

We have had a difficulty with the floating-point operators we have chosen (ADSP-7110 and ADSP-7120 from Analog Devices), their internal architecture does not allow to have the same length of pipeline for single and double precision operations when one wants to reach the optimum performance for both precisions. So the parcels in the microcode are differently delayed depending on the working precision. In primitives where both double precision data and single precision data are used, a single precision operation will cost the same number of cycles as the same operation on double precision data.

The low added cost introduced by the hardware implementation of the "delayed horizontal microcode" has to be pointed out : only multilevel registers.

3.3 Microcode generation

An intermediate language has been defined to implement primitives on the coprocessor. This intermediate language allows to describe the data movements in the coprocessor. A code generator has been developed.

Due to the "delayed horizontal microcode", microcode generation and optimization for the coprocessor OPAC is quite trivial : one can generate microcode as if the whole pipeline was passed through in one single cycle. Then there is no need for microcode compaction and no need for complex algorithms for register allocation.

On the other hand, let us imagine that we had chosen to replace the FIFO queues in OPAC architecture by dual-port RAMs and dynamic address generations. The spectrum of algorithms that would have been possible to run on the coprocessor would have been slightly larger, but the dependency analysis which would have

been necessary to take into account these possibilities would have been very difficult: for some algorithms, an addressable local memory must be viewed as a cyclic FIFO queue in order to avoid the computation of the addresses by "modulos" (the multiplication of a band matrix by a full vector for instance).

Another important advantage of the use of the "delayed horizontal microcode control" appears for nested loops: in that case, the end of the execution of an outer loop iteration is automatically overlapped by the beginning of the next iteration.

4 Some simulation results

A functional simulator of OPAC has been implemented. This simulator respects all the timings which are expected for the prototype.

Here we give some simulation results with the assumption that the host can deliver (or receive) one word of data (which may be a floating-point data, the call of the primitive or a parameter) each five cycles [6].

An asymptotic performance of 0.99 floating-point multiply-add/cycle has been obtained for the multiplication of a matrix $N * K$ A by a matrix B $K * M$ when :

1. K grows to infinity
2. The matrix result can be stored in an internal FIFO queue
3. Time to send a column of A and a row of B is inferior to the time needed to multiply them : i.e $5(N + M) \leq N * M$ (if N=M, $N \geq 10$)

4.1 FFT

The FFT primitive on 2^n elements can be executed on OPAC without extra memory access under the condition that the whole vector operand can be stored in one of the FIFO queue; since the maximum depth of the FIFO queues in the prototype is 2048 words, this means $n \leq 10$; when n is higher, technics described in [Ga87] may be used.

[6] The model of the behavior of the host is quite optimistic for MIPS R2000 or SPARC microprocessor: no cache miss, time for controls not considered ,..

During the execution of the FFT algorithm, the whole vector operand is needed during the first iteration of the outermost loop; the whole vector result is produced during the last iteration of this outermost loop. In the implemented algorithm, OPAC needs one word of data each cycle during the first step and produce one word of result each two cycles during the last step, so that the performance on a single FFT is a little disappointing : the coprocessor OPAC and the host are not busy at the same time. If several FFTs on distinct vectors are computed, then the second vector operand may be sent before storing the first vector result, improving the performance.

Simulation results are given in number of cycles needed to execute one FFT in Table 1. Av is the average number of cycles for a FFT butterfly during an "extra" FFT.

2^n	1 FFT	extra FFT	cycles per FFT butterfly
8	271	220	17.92
16	566	420	12.69
32	1186	820	9.85
64	2526	1700	8.85
128	5436	3777	8.43
256	11736	8414	8.22
512	25336	18683	8.11
1024	54525	41269	8.06

Table 1

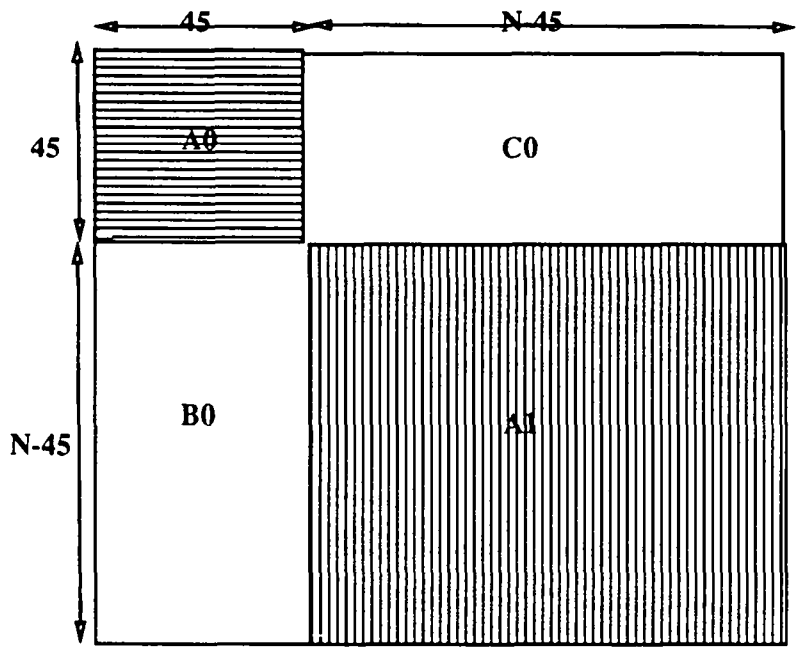
For small sizes of vectors, the performance is essentially bounded by the memory bandwidth of the host : $5 * 2^n + 4$ memory accesses are executed (send of the primitive call and three parameters + $5 * 2^n$ memory accesses for data and coefficients). For greater sizes of vectors, the performance is very close to one FFT butterfly each 8 cycles; as the FFT butterfly has been implemented with 8 additions, (except for the first step only one addition and one substraction), this performance is quasi-optimal.

4.2 LU decomposition

The LU decomposition algorithm on an arbitrary size of matrix illustrates the way OPAC may be used on large matrix algorithms.

The LU decomposition of a matrix N*N can be performed on the coprocessor OPAC without any extra memory access when the whole matrix can be stored in an internal FIFO queue of the coprocessor i.e $N^2 \leq 2048$ or $N \leq 45$ for our prototype. Nevertheless larger LU decomposition can be implemented using a block algorithm

Figure 5: block decomposition of the LU algorithm



illustrated in figure 5.

1. LU decomposition on the $45 * 45$ submatrix A0.
2. update of the $(N-45)*45$ submatrix B0 (resolving (N-45) triangular systems $Tx = y$ with the same matrix T).
3. update of the $45*(N-45)$ submatrix C0 (multiplication of a triangular matrix by a full matrix)
4. update of the $(N-45)*(N-45)$ submatrix A1 by $A1 = A1 - B0 * C0$
5. LU decomposition of the submatrix A1

The primitives corresponding to the steps 2 and 3 can be implemented on OPAC with no extra memory access [7]; for step 4, if the whole matrix A1 cannot be stored in a FIFO queue, a block version of this matrix updating is also implemented.

In table 2, simulation results are given for different sizes of the matrix. These results show that with the OPAC coprocessor it is possible to obtain performances in close to one FMA per cycle on *effective* applications such as solving full linear systems.

N	cycles	FMA per cycle
10	1471	0.226
20	6836	0.390
45	50436	0.602
100	527081	0.632
200	3546001	0.752
300	11140096	0.808
500	48529951	0.859

Table 2

5 Hardware devlopment

In november 1990, the hardware devlopment of the OPAC prototype is in its final phase :

- The whole floating-point coprocessor board using off-the-shelf chips has been designed, simulated, realized and tested at 10 Mhz frequency.
- The VLSI sequencer has been realized and tested at 10 Mhz frequency.
- The interfacing of OPAC with a MIPS box is under realization.

6 Conclusion

The performance of first generation RISC microprocessors on numerical applications remained quite poor (on the order of a few megaflops/s). To improve them, we have

[7] These primitives are in BLAS LEVEL 3 [Do88], their implementation requires the intermediate memorization of the whole triangular matrix in an internal FIFO queue. In the considered case, there is no problem; for a general implementation, a block algorithm using matrix updating is used.

shown that a reasonably large intermediate storage support is needed; but static addressing of this support is not sufficient.

In this paper, we have presented the architecture of the prototype floating-point coprocessor OPAC which has been developped at IRISA since the end of 1987. FIFO queues pnnrovide local storage support for reusable vector operands or intermediate results. A large set of numerical primitives including the library BLAS LEVEL 3 [Do88] and FFT primitives will efficiently run on this coprocessor.

The use of OPAC may allow a win of an order of magnitude on performance on effective numerical applications on dense data structures besides standard RISC workstations at a reasonable added hardware cost.

An optimized library has to be implemented to use OPAC. Automatic generation of calls to OPAC vector primitives from vectorized FORTRAN seems easy; but to obtain effective performance on applications coded in standard FORTRAN, a software tool will be needed to detect reusable vector and matrix operands or results in nested loops.

Some discussion on current superscalar microprocessors

Since, the project was started at the end of 1987, the so-called superscalar microprocessors (Intel i860, IBM RS6000) have become available; their theoretical peak performance is one FMA per cycle. At least for the IBM RS6000, it has been shown that performance close to the peak performance may be obtained on BLAS 3 algorithms [Ch90] :

- Due to post-incremented memory access and zero-delay branches, the effective data memory throughput may be closed to one data per cycle when good data locality is observed (i.e very high ratio of cache hits).
- Some limited asynchronous sequencing of the floating-point processor is possible.
- A very short pipeline has been implemented and allows to reach good performance without using a large degree of unrolling and a large set of floating-point

registers.

Nevertheless our arguments still remain available :

- Performance decrease due to cache misses may be huge for example when vectors are accesssed with large stride.
- In order to reach optimal performance, register blocking technics must be used, then particular cases must be treated.

Acknowledgements

The authors are endebted to William Jalby of IRISA for carefully reading the paper and many valuable comments.

References

[An90] E.Anderson & al "LAPACK : A portable linear algebra library for high-performance computer" Proceedings of Supercomputing '90, New-York, Nov. 1990

[Ch81] A.E.Charlesworth, "An approach to scientific array processing : the architecture of the AP120B/FPS 164 Family" Computer, Sept. 1981

[Ch90] D.Chen "Hierarchical blocking and data flow analysis for numerical linear algebra" Proceedings of Supercomputing '90, New-York, Nov. 1990

[Ch87] A.T.Chronopoulos, C.W.Gear "Implementation of s-step methods on parallel vector machines" Illinois University, 1987

[Co65] J.W. Cooley, J.W.Tukey "An algorithm for the Machine Calculation of Complex Fourier Series" Mathematics of Computation, Avril 1965

[Do79] J.Dongarra, J.Bunch, C.Moler, G.W.Stewart LINPACK Users' Guide, SIAM, 1979.

[Do84] J.Dongarra, FG. Gustavson, A.Karp "Implementing linear algebra algorithms for dense matrices on a vector pipeline machine" SIAM Review 26.1 (1984) pp. 91-112.

[Do88] J.Dongara, J.DuCroz, I.Duff, S.Hammarling "A set of level 3 Basic Linear Algebra Subprograms" Argonne Technical Report May 1988.

[Ei88] C.Eisenbeis, "Optimization of horizontal microcode generation for loop structures" 2^{th} International Conference on Supercomputing, July 1988, St-Malo

[Ga88] K.Gallivan, D. Gannon, W. Jalby "Strategies for Cache and Local Memory Management by Global Program Transformation", Journal of Parallel and Distributed Computing, vol.5, 1988 pp 587-616

[Ga87] D. Gannon, W. Jalby, "The influence of memory hierarchy on algorithm organization: Programming FFTs on a vector multiprocessor" in *The characteristics of parallel algorithms*, L. Jamieson, D. Gannon and R. Douglass, MIT Press 1987.

[Go70] G. Golub, C. Reinsch, "Singular value decomposition and least squares solution" in Num. Math., vol. 14, pp. 403-420, 1970

[Ja86] W.Jalby, U.Meier, "Optimizing matrix operations on a parallel multiprocessor with a memory hierarchy" Proceedings ICPP 1986

[Je86] Y. Jégou, A. Seznec "Data Synchronized Pipeline Architecture : Pipelining in Multiprocessor Environment" Journal of Parallel and Distributed Computing Dec.1986

[Ol80] D.P.Oleary, "The block conjugate gradient algorithm and related methods" Linear algebra and its application, 29, pp293-322, 1980

[Ph87] B.Philippe "An algorithm to improve nearly orthonormal set of vectors on a vector processor", SIAM Journal on Algebraic and Discrete Methods, Vol.8, No.3, 1987

[Ra81] B.R. Rau, C.D. Glaeser, "Some scheduling techniques and a easily schedulable horizontal architecture for high performance scientific programming" IEEE/ACM 14^{th} Annual Microprogramming Workshop Oct. 1981

[Sm82] J. E. Smith, "Decoupled Access/Execute Computer Architectures", Proceedings of 9th Annual International Symposium on Computer Architecture, pp 112-119, April 1982

[St71] H.S. Stone, "Parallel processing with the perfect-shuffle" IEEE Transactions on Computers, Feb. 1972.

LISTE DES DERNIERES PUBLICATIONS INTERNES IRISA
1991

PI 580 DESCRIPTION ET SIMULATION D'UN SYSTEME DE CONTROLE DE PASSAGE A NIVEAU EN SIGNAL
Bruno DUTERTRE
Mars 1991, 66 Pages.

PI 581 THE SYNCHRONOUS APPROACH TO REACTIVE AND REAL-TIME SYSTEMS
Albert BENVENISTE
Avril 1991, 36 Pages.

PI 582 PROGRAMMING REAL TIME APPLICATIONS WITH SIGNAL
Paul LE GUERNIC, Michel LE BORGNE, Thierry GAUTIER
Claude LE MAIRE
Avril 1991, 36 Pages.

PI 583 ELIMINATION OF REDUNDANCY FROM FUNCTIONS DEFINED BY SCHEMES
Didier CAUCAL
Avril 1991, 22 Pages.

PI 584 TECHNIQUES POUR LA MISE AU POINT DE PROGRAMMES REPARTIS
Michel ADAM, Michel HURFIN, Michel RAYNAL, Noël PLOUZEAU
Mai 1991, 10 Pages.

PI 585 TOWARDS THE CONSTRUCTION OF DISTRIBUTED DETECTION PROGRAMS, WITH AN APPLICATION TO DISTRIBUTED TERMINATION
Jean-Michel HELARY
Mai 1991, 24 pages.

PI 586 OPAC : A COST-EFFECTIVE FOATING-POINT COPROCESSOR
André SEZNEC, Karl COURTEL
Mai 1991, 26 Pages.

PI 587 ON FAILURE DETECTION AND IDENTIFICATION : AN OPTIMUM ROBUST MIN-MAX APPROACH
Elias WAHNON, Albert BENVENISTE
Mai 1991, 24 Pages.

PI 588 BOUNDED-MEMORY ALGORITHMS FOR VERIFICATION ON THE FLY
Claude JARD, Thierry JERON
Mai 1991, 14 pages.

ISSN 0249 - 6399