# A self timed asynchronous router for an heterogeneous parallel machine

Eric SENN

Centre Technique des Moyens d'Essais
Laboratoire G.I.P.
94114 Arcueil Cedex, FRANCE

Bertrand ZAVIDOVIQUE

Institut d'Electronique Fondamentale
Université Paris XI
91405 Orsay Cedex, FRANCE

**Abstract**

*This paper describes the implementation of the self timed asynchronous router in a parallel machine. The heterogenous architecture of the machine is outlined, then the need for asynchronous operations is explained, and the interest of an asynchronous network control. The specification and VLSI design of the router are exhibited with its measured performances.*

## 1 Introduction

Currently, the throughput of parallel machines rarely exceeds 20 to 30 percents, far behind the one of vector machines: usually 80 to 90 percents. Vector machine architectures did not evolve that much the last few years, so current vector compilers are efficient enough. On the contrary, the diversity of parallel architectures makes parallel compilers harder to build and stabilize. Anyway, such a compiler by itself would never fill the gap between processors' power and data transfer rates along communication networks. Better rely on specific hardware ressources, like in vector machines where independant *load and store* units feed vector registers for the processor to keep computing, in achieving a data prefetch. The efficiency of the technique grows with the network speed. Then, the bandwidth between memory and processor has to be maximized.

Let us define a *network hierarchy* regardless from any technological level. In classical hierachies, the so called processor hierarchies, upper layer units are still interconnected by the network, involving a strong bottleneck. In a network hierarchy, bandwidth requirements are handled over the whole memory hierarchy. This implies several independant communication levels with respect to processors, in charge with data transfering and reordering in the memory, while computing. To check the concept we seek efficiency on a wide class of algorithms, in both scientific calculation and image processing. So, an high granularity MIMD machine with a bulky global memory seems to be a reasonnable choice provided high speed communication ressources allow for data parallelism. The latter programming model fits low-level image processing requirements - array data structures - whereas control parallelism rather suits high-level operators - intricated graphs for recognition. A bit of scalability should ease and widen further aplications.

To sum up, our aims are to:

- build an high granularity MIMD machine with a bulky memory
- use an on-the-shelf processor
- share memory
- take into account technological constraints since the very design stage

Hence:

- optimize the throughput that could be bad if computing ressources wait for informations
- improve the bandwidth, by reducing memory access and network access latencies

Hence:

- separate computing functions from adressing functions, i.e. data transfer and reordering
- use prefetch
- ensure the overlaying of calculation / adressing ( prefetch ) / communications

And that means:

- actually test a prototype of **asynchronous architecture**

## 2 Phenix architecture

The Phenix heterogeneous architecture results merely from putting together harmoniously classical solutions to each subgoal listed above. Its nodes will so include calculating ressources, a huge memory, a cache, and routing ressources. Caching is necessary to profit from data locality, routing ressources to free the processor from communication and networking tasks.

0-8186-8409-7/98 $10.00 © 1998 IEEE

A classical cache drops with vector calculation. Vectors mostly spread over the whole local memory, with poor locality, and non continuous assignments. Besides, a huge static memory would imply voluminous expensive power supply and cooling devices, would be voluminous itself ( 4 to 6 transistors / cell) and expensive. We thus prefer dynamic memories which have none of these drawbacks. In order to mask their latency, they are interleaved. The next problem is to build a long bus, as long as the memory is large, and a fast one, connecting memory banks to the cache. A pipelined bus will do, separating sections by registers, should furthermore improve the scalability for memory size expansion.

To improve the peak to raw power ratio, an adressing controler prefetches and feeds the cache, dealing with data coherency, in relation with several adressing processors, one per memory node. Likewise, routing ressources are added to relieve the main processor, and distributed over the machine architecture to reduce the network access latency, as shown on figure 1. Routing processors are called routers.

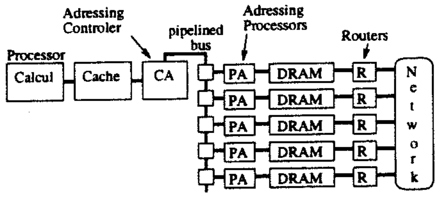


Figure 1: A calculation node of the machine

This set made of one processor, cache, adressing controler, a set of adressing processors, memory banks and associated routers, makes a *line* of the machine, equivalent to the pair *processor plus router* in a classical parallel architecture. Considered separately, this complete computing unit offers strong facilities for vector processing, especially because datas in cache can be arranged in vector registers. The overall architecture is completed in interconnecting the memory banks of these vector lines into a *plane* through a 2D-torus ( a cyclic mesh ), then in stacking several such planes, interconnected by an *omega pyramid* vertical network, see figure 2 [15].

## 2.1 The network

A network is as more efficient as its node number, $N$, is closer to the theorical *Moore's* maximum, given its diameter $D$, and degree $\Delta$.

$$N(\Delta, D) \leq \frac{\Delta(\Delta-1)^D - 2}{\Delta - 2} \; for \; \Delta \geq 3 \; ; \; N(2, D) \leq 2D+1$$

Furthermore, $D$ is kept small to minimize distances traveled by messages, and $\Delta$ must be equally small, to

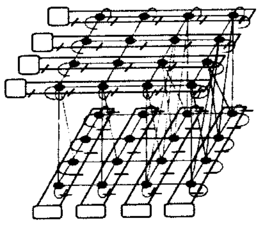


Figure 2: Two planes, four lines each

maximize the bandwidth on each vertice of the graph, since the number of connections on boards is limited. The omega pyramid is designed as shown on figure 3 [4]. From one stage to the next, alternately rows, then columns are interconnected as an omega network ( the horizontal torus not represented here ). The last stage is connected to the first one.

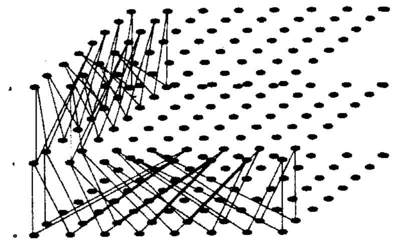


Figure 3: 2 stages interconnection

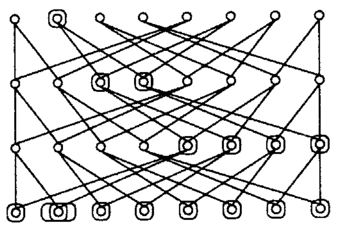


Figure 4: Construction

This network shows several interesting properties. It is universal, it can route any permutation in three revolutions, so blocking doesn't matter. Considering $2Log(n)$ planes of $n^2$ nodes, its diameter is less or equal to $3Log(n)$, for a total of $2n^2Log(n)$ nodes. In a plane, the diameter is $2Log(n)$ for $n^2$ nodes. This topology is scalable by symetry, with an invariant degree of 8, then achieving good compromise according to *Moore*. Only two layers can emulate the whole machine, thanks to isomorphism between pairs of layers. Figure 4 shows that any node is the top of a pyramid, the base of which is the complete plane including this top node, after one turn. This omega pyramid then contains pyramidal topologies and trees, and bears embedding many classical data structures, without any bottleneck at the top. Again, the programming model of a plane is data parallel. The network efficiency relates to the graph plunging compute efficiency over the network graph. Data structure plunging explains the succes of classical topologies such as trees, meshes, pyramids, and hypercubes, and was a kee concern in designing the omega pyramid.

The vertical network bears long distance communications such as permutations routing, associative queries (maximum calculation,...), broadcasting, gossiping or scans, with complex data movements, and possibly heavy processing that triggers control parallelism between stages. The horizontal network 2D-torus carries simpler communications in a plane, supporting data transfers between close neighbours in the computing scheme. It is *polymorphic*, allowing for more complex communication structures, so called *stencils* [3].

## 2.2 The "line" computer

As already mentioned, a line (see figure 5 ), the basic unit of the machine, is equivalent to the calculation node of a classic parallel machine, but in addition can be seen as a vector machine.

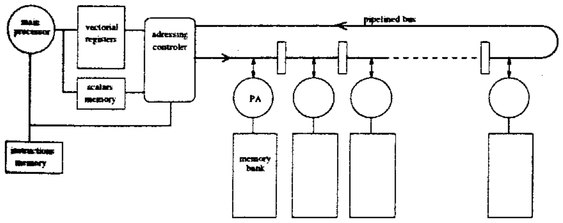


Figure 5: The line vector

Obviously, we will gain from the pipelined bus by an interleaved repartition of vector elements in the memory banks. A convenient *stride* should be the number $n$ of memory banks on line. $n$ vector elements are loaded in one read of the pipelined bus, accordingly dividing the acces time to one element. Operating on these vectors requires vector buffers nearby the main processor, both super-scalar and pipeline. Indeed, vector machines generally display several pipelined units, working in parallel on the set of vector registers. Most current RISC processors do that now. But a scalar processor handles vector datas by mean of logical loops that slacken calculations, enough that the vector machines efficiency never be reached. In addition, algorithms with uneven stride reduce the data interleaving interest. Nevertheless, scalar computing must not be neglected, for the scalar parts of vectorized algorithms, and for actually unvectorizable ones [11].

Adressing processors load and store datas under the control of the adressing controler at the head of the line. Simultaneous acces to data memories is by FIFOs or dual-port devices.

## 2.3 The memory node

The structure of a memory node is given figure 6.

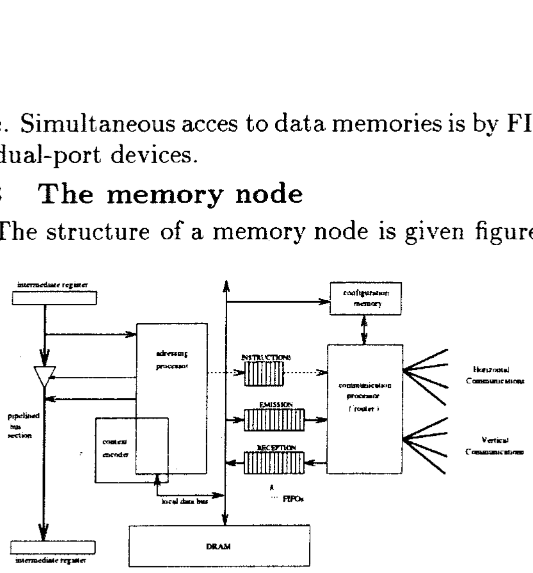


Figure 6: A memory bank, on a Phenix line

The *adressing processor* receives instructions from the pipelined bus. It loads and stores data, refreshes memory, computes adresses for indexing strides. The *context encoder* skips useless accesses and processing of vaccuous vector elements. This processor also performs read and write from and to the *communication FIFOs*. These FIFOs support computing, adressing and communication overlays. They are doubled to make horizontal routing independant from vertical routing. The vertical routing is of the *virtual cut-through* type, the best compromise between communication latency, *flits* number, and *inter-blocking* probability. The vertical router remains classical, processing on aligned message flows from different sources.

Horizontal routers execute complex stencils, and ensure polymorphy by switching messages. The plane control is SPMD: every main processor executes the same program, and gets to know the data repartition all over the mesh; the adressing space is unique. The main processors are all connected to a dedicated bus that eventually permits synchronisation. The execution of stencils, a *communication session*, can be divided in several phases, depending on the routing algorithm. For pre-compiled stencils, associated switching configurations are stored in the *configuration memory* of each router. A session begins with sending a session number from every adressing controler to each router of its line. Then the routers emit, receive, or transmit data, according to the configurations they read. *Asynchronous* functioning is then essential because it avoids a strong synchronization of all the routers in the plane at the end of each communication phase. As soon as a router ended a phase, it can begin the next one independant from the state of its neighbours. Identically, sessions can be chained by a router in an asynchronous manner, if several are stored in its *instructions FIFO*.

It is all the more usefull as the configurations, their number, their chaining, are particular to each router, depending on its location in the plane. The main processor knows the ending of a session when it has recovered every data it was waiting for. The synchronization is then implicit, not global anymore. Horizontal routing - *circuit switching*, *wormhole* - has to be fast, it is mainly devoted to regular data transfers, stencils. It supports logical computing graph plunging, which the need for adapted plane communications control stems from.

## 3 Asynchronous control

Several practical difficulties arise from actually building the machine. In the plane, the correct transfer of informations between routers has to be secured. The machine is assembled on several printed circuit boards, plugged into a rack, interconnected by several back rack busses. A *synchronous control* would ask for the distribution of a global clock all over the machine, to the boards supporting parts of a same plane, along significant busses, through many connectors. A fast clock would involve building of complex, thus expensive, distribution tree to minimize the *clock-skew* effects [7]. Indeed, bus speeds are limited to about 60 Mhz, inside a single board, and for basic busses. The difficulty and cost of such a control mode should grow with the topology complexity, the size, and especialy if some scalability is seeked for.

*Asynchronous control* then suits better because there is no more need for a global clock. For instance, the *synchronized asynchronous control* means a global clock per component only ( see figure 7 ). On every input, synchronizers [8], minimize the failure probability due to *metastability* [6]. The cost of this extra hardware is an increase of the circuit surface, and a loss in performances, extra time beeing necessary for synchronizers to resolve [1].

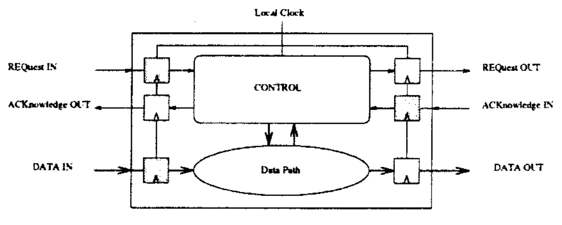


Figure 7: Synchronized

*Self timed asynchronous control* doesn't need any clock at all ( see figure 8 ). The activity of circuits is driven by the incoming data itself. Significant gain in power consumption because the circuits actually do nothing when they have nothing to do. Clocking

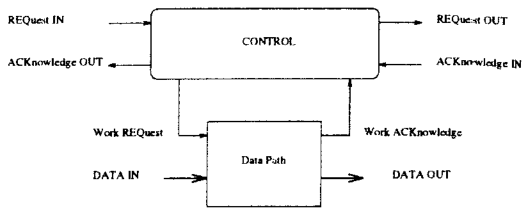


Figure 8: self timed

causes the clock lines to load and unload continuously, consuming power. Furthermore, register commutations produce peaks of current that polute the power supply lines. "No clock" goes with lower supply voltage without hampering the circuit functionnality, but for its performances [12]. In the asynchronous control case mean performances are obtained, not worst-case performances as in the synchronous control case. Consider the chain of processes figure 9.

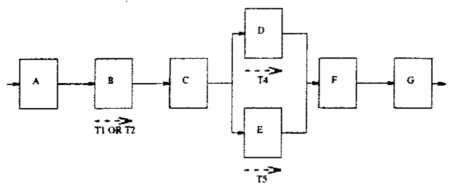


Figure 9: Chain of processes

The clock period for a synchronous control should be greater than the slowest response time of all the processes. In such case, the performance over the chain is always to the worst case. On the contrary, under asynchronous control a process gives its result as soon as it finished working. If T1 is slow but runs more rarely than faster T2, the response of all the chain is faster most part of time, similarly for T4 and T5, then achieving mean performances. This example suggests that, among other advantages, asynchronous control offers some insensitivity to characteristics dispertion. It makes sense to optimize only portions of circuit that are widely used, or to have an heterogeneous technological evolution in the case of systems [16].

Synchronized asynchronous machines are designed like synchronous ones. On the other hand, specific methods must be used for self timed design to be hazard free [5]. Those methods depend on the model of delays in the implementation: *bounded delay*, *delay insensitive* ( i.e. unbounded in wires and logic operators ), or *speed independent*, called *quasi delay insensitive* too, (i.e. delay insensitivity plus *isochronic forks* [9] ). We try and compare results of the first and the last one in our design ( see section 4 ).

Asynchronous control involves extra control lines to implement a communication protocol. This protocol can be four, or two cycled [14]. The first solution remains simpler hardwarewise.

We focuse on the optimisation of the controler circuit performances. Our design method is based on a low level description, *state graph*, and *signal transition graph* at the higher level [2, 10]. We defined several properties to ensure the reliability like the *integrity* of the *state crossing* and its *best covering*, and we set the *constraint* of signal transition graph to take care of *out of the cycle* signals [17]. Eventually, asynchronous control is a natural support of asynchronous operating.

## 4 VLSI Implementation

The figure 8 shows that particular data paths are added with self timed asynchronous control. Any data path element must be able to begin upon request, and to signal the completion of its work. Such elements can't be found in standard libraries from ASICs manufacturers, so they have to be built. The asynchronous cells we need for our data path are *stability detector*, *asynchronous comparator*, *asynchronous FIFO*. The asynchronous FIFO uses specific *Muller cells*, and *switch invertors* that we have designed too.

Muller cells are also found in the control parts of our self timed circuit ( see figure 8 ) [13]. We need two different types of Muller cells, the first is a regular one, the second gets an inverted input and a supplementary inverted clear input. They are designed in static logic, as the switch invertor cell. On the contrary, the asynchronous comparator and stability detector rely on differential dynamic logic, involving preload phases and the evaluation of both the function and its complement. As examples, we give figure 10 and 11 the structures of a Muller cell and the asynchronous comparator.

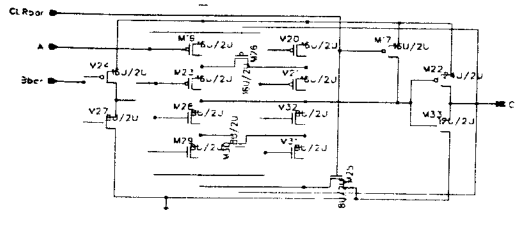


Figure 10: Muller cell

Stability detection runs whith standard data path elements whenever it is possible, for instance with a

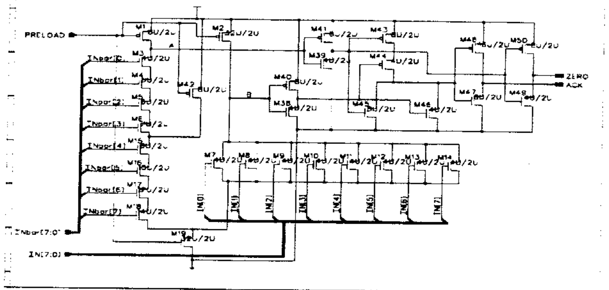


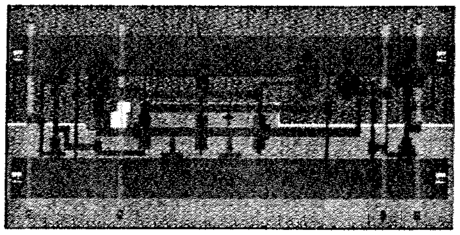
Figure 11: Asynchronous comparator

counter. These elements are made of standard cells from the manufacturer's library. In our circuit, standard cells also do for switching devices, multiplexors and demultiplexors, and several security and configuration modules. As a result, the major part of the design calls for standard cells, and the floorplan includes large standard cells zones. If our customized cells were outside such zones, multiple long wires would interconnect them, and it would results in a lesser speed. To include our custom cells in standard cells zones, we designed them following the guidelines of the standard library in use. Moreover, our cells are recognized by the automatic placing and routing CAD tools. Figure 12 shows the layout of a Muller cell and the asynchronous comparator. The technology is $0.7\mu$ CMOS from ES2 manufacturer, and COMPASS CAD tools.

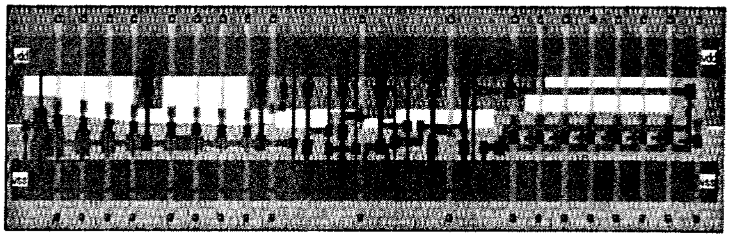
We made retroannoted simulations of those cells and obtained the results table 1, for a load of $0.05pF$.

The horizontal network is four connected, with full duplex links between neighbours. This mean two ports, one input and one output per neighbour, hence eight ports per router. A router is able to open several communication channels concurrently; it can emit on a port, receive on an other, and transmit between two separated pairs. All these tasks are achieved by four distinct asynchronous modules. A configuration module to read the instructions ( session numbers ), to read configuration memory, the configuration of switching devices, and correct operations through phases and sessions. Extra security hardware is necessary to avoid electrical conflicts on busses in case of bad configuration. The layout of the full circuit is given figure 13.

On this evaluation version, the number of ports is halved. Two different versions of every asynchronous module are included, with a special input added to make choice. The first version is fully speed independent, the second merely considers that some data paths are delay bounded, allowing to evaluate the cost of work completion devices. The total circuit area, in-



(a) advanced muller cell



(b) asynchronous comparator

Figure 12: Cells layout

| | Muller cell | Muller cell 2 | As. comparator | Stability det. | Switch inv. |
|---|---|---|---|---|---|
| tpLH | 0.89ns | 1ns | 1.36ns | 1.22ns | 1.64ns |
| tpHL | 1.12ns | 1.2ns | 2.55ns | 1.42ns | 0.99ns |
| trise | 0.67ns | 0.55ns | 0.47ns | 0.5ns | 1.24ns |
| tfall | 0.62ns | 0.55ns | 0.72ns | 0.5ns | 0.6ns |
| tprecharge | | | 0.3ns | 0.3ns | |
| area | 3072 $\mu^2$ | 4669 $\mu^2$ | 6758 $\mu^2$ | 3317 $\mu^2$ | 2334 $\mu^2$ |

Table 1: Full custom cells characteristics

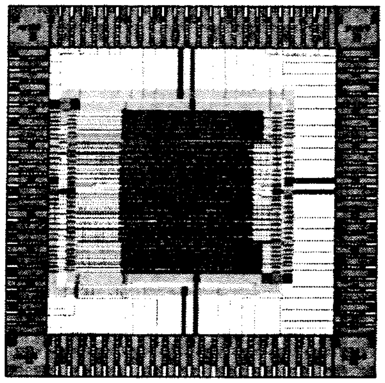


Figure 13: Circuit layout

cluding pad ring, is 24.11 $mm^2$ for about 4000 equivalent gates. The circuit power consumption is $11.34\mu A$ at rest, when no data incomes, and grows to $120\mu A$ while running actually. The measured timings of the circuit are displaid table 2.

$l_{com.}$ and $l_{com.2}$ are the latency of the asynchronous FIFOs for emission and reception. $t_{trans.}$, $t_{rec.}$ and $t_{eme.}$ are the crossing times of a transmitter, a receiver, and an emitter module. $T_{trans.}$ and $F_{trans.}$ are the transmission period and frequency. These measures are about 10% better than the simulation results on the whole circuit. Simulation appears then reliable to validate the correct operation of our circuits and to estimate their performances. Comparing the transmission times, about $15ns$, to the transmission periods, about $30ns$, shows that half the time is wasted by the modules for reinitialisation of control lines, due to the choice of a four cycled protocol. The next version of these modules will use a two cycled protocol, and should give a transmission period near the transmission crossing time, and then a transmission frequency close to $60MHz$, for the same technology. The difference between the "full speed-independent" and the "some bounded-delay" circuit performances shows that the influence of work completion circuitry, i.e. of the asynchronous data path elements, is not significant; barely a 10% degradation in performances.

## 5 Conclusion

Once given the machine architecture and its programming model, we show a need for asynchronous operation in data transfer all over the network. Physical constraints in assembling the machine leads to the implementation of an asynchronous control for routers. Asynchronous control naturally supports asynchronous operation. Our concerns about optimisation and the search for performance leads to the design of a self timed circuit, speed independent for more reliability. This involves the design of specific cells for both asynchronous data paths and control. These cells are drawn according to the standard cells guidelines, to make easier their placing and routing on the floorplan, and to guarantee better performances. The circuit contains several asynchronous modules working concurrently, and extra logic for configuration, security, and switching. The measured performances are close to simulation results, that proves simulation re-

| | $l_{conf.}$ | $l_{com.}$ | $t_{trans.}$ | $t_{rec.}$ | $t_{eme.}$ | $l_{com.2}$ | $T_{trans.}$ | $F_{trans.}$ |
|---|---|---|---|---|---|---|---|---|
| some bounded tplh | - | 46ns | 17ns | 22ns | 10ns | 38ns | 31ns | 32MHZ |
| some bounded tphl | - | 40ns | 14ns | 20ns | 10ns | 31ns | 31ns | 32MHZ |
| speed indep. tphl | - | 46ns | 19.5ns | 27ns | 15ns | 38ns | 35.7ns | 28MHZ |
| speed indep. tplh | - | 40ns | 16.2ns | 21ns | 13ns | 31ns | 35.7ns | 28MHz |

Table 2: First evaluation version measured performances

liability. They indicate that the choice of a two cycled protocol would improve in a factor of about two the transmission frequencies. This protocol is currently implemented in the next version of the circuit. The measurements also show that the cost of speed independence for total reliability would not exceed 10% of the global performances.

## References

[1] T.J. Chaney. Measured flip-flop responses to marginal triggering. In *IEEE Trans. on computers*, December 1983. vol c-32, number 12.

[2] Tam-Anh Chu. Synthesis of Hazard-Free Control Circuits from Asynchronous Finite State Machines Specification. *Journal of VLSI Signal Processing*, pages 61–84, February 1994. Volume 7, Numbers 1/2.

[3] P. Fiorini. Une étape dans la conception de l'architecture de la machine Phénix. Technical report, DGA/CREA/ETCA, Labo SP, March 1994. Rapport interne.

[4] P. Fiorini. *L'Omega Pyramide.* PhD thesis, Université de Paris-Sud, U.F.R. Scientifique d'Orsay, June 1996.

[5] S. Hauck. Asynchronous design methodologies: An overview. In *Proceedings of the IEEE*, January 1995. vol 83, number 1.

[6] L. Kleeman and A. Cantonni. Metastable Behavior in Digital Systems. In *IEEE Design and Test of Computers*, December 1987.

[7] S.D. Kugelmass and K. Steiglitz. An upper bound on expected clock skew in synchronous systems. In *IEEE Trans. on computers*, December 1990. vol 39, number 12.

[8] W.Y-P. Lim and J.R. Cox. Clocks and the performance of synchronisers. In *IEE. Proc*, March 1983. vol 130, Pt E, number 2.

[9] A.J. Martin. The limitation of delay-insensitivity in asynchronous circuits. *in Advanced Research in VLSI*, 1990. MIT Press.

[10] T.H.-Y. Meng, R.W. Brodersen, and D.G. Messerschmitt. Automatic Synthesis of Asynchronous Circuits from High-Level Specifications. In *IEEE Transactions on Computer-Aided Design*, pages 1185–1205, November 1989. Volume 8, Number 11.

[11] P. Michaud. Quel RISC pour Phénix. Technical report, DGA/CREA/ETCA, Labo SP, 1994. Rapport de DEA.

[12] L. S. Nielsen, C. Niessen, J. Sparsø, and C. H. van Berkel. Low-power operation using self-timed and adaptive scaling of the supply voltage. *IEEE Transactions on VLSI Systems*, 2(4):391–397, December 1994.

[13] O. Petlin and S. Furber. Designing C-elements for testability. Technical Report UMCS-95-10-2, Computer Science, University of Manchester, 1995.

[14] C.L. Seitz. System timing. In *Introduction to VLSI systems*, chapter 7. Addison-Wesley P.C., 1980.

[15] E. Senn, P. Fiorini, P. Hoffman, and T. Porcher. Proposition pour une architecture hétérogène. In *RenPar6, 6èmes Rencontres Francophones du Parallélisme*, June 1994.

[16] E.P.A. Senn. Horloges et Synchronisation. In *Rapport de Stage de DEA*, September 1993.

[17] E.P.A. Senn. *Conception pleinement asynchrone: méthodologie et application à la réalisation d'un processeur de communication pour machine parallèle.* PhD thesis, Université de Paris-Sud, U.F.R. Scientifique d'Orsay, 1998. in preparation.