A R T I C L E

MAD, A FLOATING-POINT UNIT FOR MASSIVELY-PARALLEL PROCESSORS

A. Bartoloni, C. Battista, S. Cabasino, N. Cabibbo, F. Del Prete, F. Marzano, P.S. Paolucci, R. Sarno, G. Salina, G.M. Todesco, M. Torelli, R. Tripiccione, W. Tross, P. Vicini and E. Zanetti

Abstract

We describe in detail the architecture and implementation of the MAD chip. It is a floating point unit, used as the elementary processing element of the APE100 array processor. The design has been accurately tailored to the requirements of a SIMD floating point intensive machine.

1. Introduction

Dedicated processors have recently emerged as an important element in the numerical simulations of Lattice Gauge Theories (LGT). In fact, a fair fraction of LGT simulations have been performed on dedicated processors in the last 5 years.

Typical dedicated machines for LGT (LGT engines, for conciseness) exploit the rather unusual features of the relevant numerical algorithms and are able to achieve sustained performances exceeding those of state-of-the-art commercial supercomputers [1]. This is possible for the following two reasons. First, LGT algorithms are homogeneous and local, so that massively parallel arrays of processing nodes with limited inter-node connectivity achieve performances growing nearly linearly with the number of processing nodes. Second, LGT calculations exhibit an unusually high ratio of floating-point operations over required data operand (of up to ten to one). This high ratio helps to keep very fast processing nodes busy in spite of limited I/O bandwidth capabilities. Moreover, limited bandwidth requirements make large arrays of processors easier to assemble.

It is interesting to note, however, that actual programs to design, build, and operate LGT engines have been triggered essentially by technological developments in the field of micro-electronics. Indeed serious (that is, faster than commercial machines) LGT engines appeared only in the mid eighties when floating point chips were developed by industry. Later developments have essentially concentrated on the fairly limited increase in the number of processing elements and their upgrade to the better performing devices that have been developed.

In this paper, we report on a first attempt to resort to a different, albeit related, technological development to substantially increase the performance of LGT engines by nearly two orders of magnitude. This progress is made possible by the use of custom-designed integrated circuits, which can be accurately tuned to the rather unusual needs of LGT engines.

The plan of this paper is as follows: sect. 2 schematically describes the architecture of the APE100, a massively-parallel LGT engine whose processing node is based on one custom-integrated circuit, the main arithmetic data (MAD) chip. A discussion of the substantial advantages offered by custom design for parallel LGT engines is given in sect. 3. Section 4 describes the architecture of the MAD chip, while sect. 5 discusses the technological constraints within which the design of our custom-integrated circuit had to be performed. Section 6 presents details on the actual implementation of the device. Section 7 contains some remarks on our project methodology and is followed by our conclusions.

2. The APE family of parallel LGT engines

This section is a brief introduction to LGT engines in general and to the architecture of a specific family of parallel processors known as the APE family. The interested reader is addressed to the proceedings of the Lattice Conferences of the last few years for the history of the development of dedicated computers for LGT and to ref. [2] for a detailed description of the APE100.

LGT engines were proposed early in the eighties, when it was realized that parallel processing was an efficient technique to provide the computing power (typically, thousands of Cray level CPU hours) necessary for realistic simulations of field theories on the lattice. Indeed, LGT codes can be run on large processor arrays with relative performance close to one, as long as the number of processors does not exceed the number of physical sites on the lattice (of the order of 30^4, or larger). The basic reason why such unexpectedly high performance can be reached is now explained.

Field theories are discretized on a four-dimensional lattice of points and dynamical variables corresponding to the physical degrees of freedom are defined at each lattice site. A statistical Monte Carlo procedure is used to provide equilibrium configurations of the dynamical variables on which expectation values of operators are measured. "Equilibrium" means that each configuration appears with a probability inversely proportional to the exponential of its weight in the action. The evolution in Monte Carlo time of the variables attached to each lattice points depends only on the configuration at the point itself and at its neighbours since actions are local, therefore the evolution of several lattice points can be evaluated in parallel. Consequently, it is possible to use several processors that operate concurrently on different sublattices. Action locality ensures that communication overhead between processors, one of the main factors in performance degradation in parallel processing, will be limited. Load balancing among processors, another key factor to achieve high performance in parallel systems is ensured by translational invariance of the theory, so that each lattice point requires on average the same amount of processing. Note also that since all lattice points undergo the same evolution and measurement algorithms, the simplest form of parallel processing, i.e. SIMD (Single Instruction Multiple Data) processing is adequate for LGT.

The APE family of processors was proposed in 1985 to reach the target of 1 Gflops in LGT simulations. APE [3] was a small-scale parallel processor of nodes arranged along a ring; machines with 4 to 16 nodes were built. All nodes operated in SIMD node and accessed non-local data across a barrel-shift switching network. It was decided to keep the number of nodes rather small, and to squeeze as much processing power as possible inside each node. Processors of 64 Mflops were built, incorporating a hardwired complex number floating point unit. A SIMD structure was used since it was simpler from the hardware point of view and easier to program.

APE has been used for LGT simulations since 1986 (1987 for the 16-node machine). The APE100 project was started in 1988 to provide a substantial increase in floating point performance and a somewhat broader spectrum of applications for the machine. This time, it was decided to set up a massively-parallel machine, since assembling a large number of nodes is easier (if each node is simple, compact and reliable) than building an appreciably more powerful processing node. This point is explained in detail in later sections.

The new project is a rather straightforward upgrade of the original APE architecture. It provides a massively parallel array of nodes (in excess of 1000) running in lock step mode and steered by just one controller (SIMD architecture). There is a major difference, however, in the topology of the processing nodes. We have adopted a three-dimensional mesh, particularly well adapted to the simulations of physical systems in three or more dimensions. A further difference is that APE100 nodes perform real (as opposed to complex) number arithmetics, making the new machine much more versatile.

The actual structure of the machine is based on a processing element containing one floating point unit and one memory bank. Nodes can be assembled as one, two or three-dimensional arrays of, in principle, arbitrary size with direct data links to nearest neighbours in all dimensions. All nodes execute the same instruction at each clock cycle. Sequencing of program instructions is ensured by a master controller, which also provides the common address to all memory banks. The peak performance of the processing node is of the order of 50 Mflops (as explained in detail in later sections). About 2000 nodes are necessary to reach our 100 Gflops target. They can be assembled as, for instance, a $16 \times 16 \times 8$ mesh of processors.

3. Custom chip design and massively-parallel processors

This section describes the reasons that led us to base the actual implementation of the APE100 architecture on two custom chips, incorporating respectively the floating point unit of each node and the data link to the neighbour nodes.

It might seem that the assembly of some appropriate sets of commercially available integrated circuits (floating point building blocks, in our case) is the obvious way to design a complex computer system. In the case of the APE100, a better choice was available, as is explained below.

Floating point devices have been generally available for several years now, with different levels of performance and integration. They can be broadly divided into:

(a) coprocessor devices for standard microprocessors,
(b) floating point subsystems embedded within a high-performance microprocessor, and
(c) stand-alone floating point units.

None of these proved to be the best choice for our machine. Class (a) is immediately ruled out for its low performance (in the Mflops range), while class (b) (exemplified by the Intel i860) is unsuited for the APE100, since it would require a complete redefinition of the architecture of the machine. Class (c) must be considered with more attention. Indeed, the floating point unit of the original APE machine was assembled with commercial ALUs (Arithmetic and Logic Units), multipliers and register files, controlled at each clock cycle by an appropriate command word. Several improved devices are now available, incorporating arithmetic units and registers on the same chip, with clock rates up to ~ 40 or 50 MHz, designed as floating point blocks for high-performance workstations or graphics engines. These devices are not the best choice for the APE100 for both architectural and electrical reasons, as discussed in the next paragraph.

In a massively-parallel machine it is essential to keep the I/O bandwidth for the computing nodes as low as possible without compromising the floating point performance. This is not an important design requirement in standard devices. A typical routine for a graphical workstation might be a 3-d coordinate transformation of a set of points. An optimized, possibly hand-coded routine would first load the elements of the rotation matrix $A\langle i,j\rangle$ and the displacement vector $D\langle i\rangle$ onto the register file. The procedure would then load in succession the coordinates of the data points $P\langle i,n\rangle$, perform the coordinate transformation $P'\langle i,n\rangle = A\langle i,j\rangle\, P\langle j,n\rangle + D\langle i\rangle$ and store the new values.

The above calculation requires 18 floating point operations, 3 data inputs and 3 data outputs, that is, 3 floating point operations per I/O operation. We call R the value of this ratio. In practice, available floating point devices are tuned for a much smaller value of $R \approx 0.5 - 1$, because they are designed to run code generated by high-level (e.g., FORTRAN or C) compilers. These compilers are not very efficient in register optimization and often save onto memory all intermediate results of the calculation, requiring a larger bandwidth. The strategy behind the design of typical floating point devices is now clear: a small number of registers (typically 32) is accompanied by one or even two 64-bit data buses that can be used for input or output at each clock cycle. This degradation of R is sustainable in a workstation that uses fast memory devices and perhaps a cache, but would be prohibitively expensive on a massive parallel machine. In this case the most economical strategy consists in providing each node with a large number of registers and in using an efficient compiler.

The actual number of registers required by this strategy can be estimated by considering one of the most I/O intensive LGT kernels, the product of a Wilson spinor by an SU(3) matrix [4]. The working set includes one gauge matrix (identified by two of its columns, a total of six complex numbers), one set of input fermion fields (4×3 complex numbers) and one set of output fermion fields, for a total I/O activity of 60 data words moved from/to memory. The actual matrix multiply requires three complex multiplies and 2 complex adds for each of the twelve numbers. Altogether, 212 real operations are required, yielding an R of ~ 3.5, not very different from the previous example. It must be stressed, however, that we can achieve such a large value for R only if a large working set (~ 70 words) can be kept inside the register file. A smaller register file would lead to more frequent data moves, that is, to a smaller value of R, even for very accurately tuned routines. Extensive simulations on a variety of LGT algorithms have shown that this strategy actually requires ~ 100 registers.

From the electrical point of view, standard floating point units have large buses and high-clock rates, resulting in high-power rates. They also require complex glue logic. Glue logic has a bad impact on a massive system, since it usually requires a small number of gates assembled on several chips of "smallish" scale of integration. For instance, an error detection/correction circuit contains about one thousand gates, but needs nearly as much board space as one full floating point unit.

The above considerations lead to our picture of the ideal floating point unit for a parallel engine: a dedicated device with a large register file of ~ 100 registers, floating point arithmetic circuits and all glue logic required to keep the node at work. The data bandwidth can be kept rather low, compatible with a value of $R \geq 2$, but maximum flexibility must be provided to control the chip, since it is anticipated that very efficient programming is needed to sustain high performance.

Note finally that, although this type of device is accurately tuned towards the requirements of LGT simulations, it can be efficiently used in all kind of floating point compute-intensive algorithms.

A device with the desired mix of features was not available, and we had to develop it from scratch. Its structure is presented in sect. 4.

4. The architecture of the MAD chip

This section presents a description of the architecture of the MAD chip, with emphasis on the accurate tuning of the device to the particular needs of both LGT calculations and embedding in a SIMD parallel system.

The core of the device is a floating point data-path, with ample register space (128 registers), a floating point multiplier and a floating point ALU with equal performance, since multiplies are roughly equally frequent as adds in scientific computations. These basic building blocks must be wired together in such a way as to: (a) maximize performance and (b) make automatic code optimization as easy as possible. A complete discussion of these requirements would lead us far away, but can be summarized by these two points:

(a) the configuration must be such that the multiplier and the ALU can operate concurrently,

(b) for each operation there must be a unique source of operands and a unique destination for results.

The second requirement frees the compiler from the burden of choosing the best strategy for intermediate storage. In MAD, all arithmetic operations are register-to-register. An optimal layout within the above constraint is shown in fig. 1, where it is understood that each port can be accessed at each clock cycle.

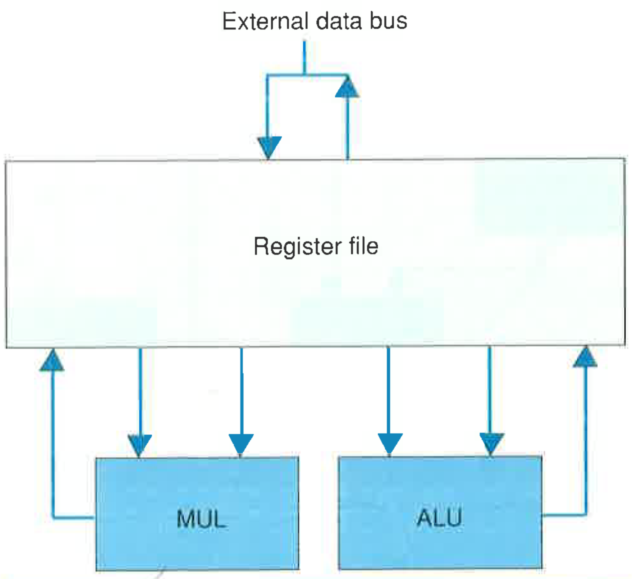

FIGURE 1

A possible optimal architecture for the MAD chip.

This layout allows completely independent operation of both floating point devices and of all I/O activity, with obvious advantages for performance and code optimization. A serious drawback is that a large number, 8, of register ports are necessary, requiring a very complex register file design and a wide bit field for register addressing. We have therefore chosen the layout of fig. 2.

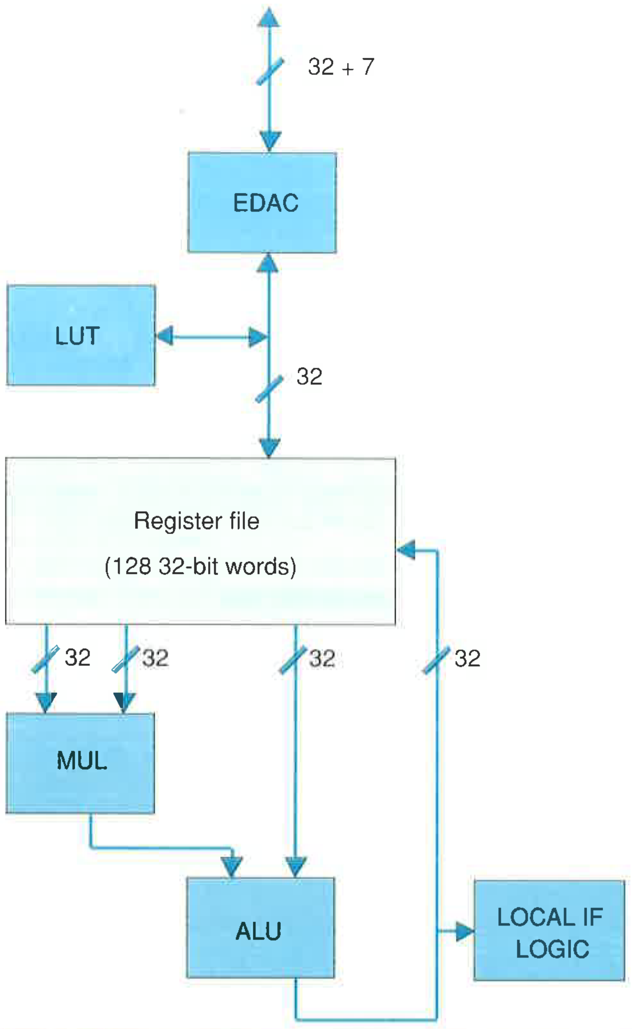

FIGURE 2

Simplified block diagram of the MAD chip.

This structure requires only six register ports, two less than in the previous example. It does not guarantee that the multiplier and the ALU can be kept busy at all times, but this is ensured for several frequent algorithmic structures (e.g. matrix multiply, of real or complex numbers), in which triadic expression must be evaluated.

The main arithmetic data path is pipelined, so that a new operation can be started at each clock cycle. The pipeline length is chosen as a reasonable balance of performance and latency and is heavily dependent on technology considerations [5]. Details on this point will be given int sect. 5. Note, however, that the pipeline delay is the same for all arithmetic operations, a very important element in achieving maximum performance with relatively simple compilation techniques.

It is convenient to include additional hardware to the basic structure discussed above, in order to guarantee reasonable performance for those operations that, although less frequent, would require long and cumbersome expansions in terms of adds and multiplies. A list of these additional devices includes:

(a) Look-up (LU) circuitry encoding first approximations of the functions $1/x$ and $1/\sqrt{x}$. Correct inverses and inverse roots are calculated with an iteration procedure. One over square root is traditionally preferred because it is more frequently used than square root.

(b) Look-up circuitry for first approximations of the $\log(x)$ and $\exp(x)$ functions, which are widely used special functions in Monte Carlo algorithms. This circuitry is necessary because bit-manipulation instructions used in standard routines to compute logs and exps are not available on MAD.

All look-up operations are again register-to-register, but no dedicated ports are available on the register file for this purpose. The register-file ports to/from the data bus are used instead. This configuration has been chosen in order that the main arithmetic path can be kept active during LU accesses. Contention on the I/O bus is not a problem, since when an LU is accessed, a very compute-intensive operation is being performed, so traffic on the I/O bus is expected to be low.

The arithmetic capabilities of the MAD chip have been discussed so far. Let us now consider those features of the device used to: (a) interface it to the outside world and (b) support SIMD parallel operation.

Point (a) is straightforward. Just one bi-directional 32-bit data bus is used for all data I/O activities. In the actual APE100 system, this bus is directly connected to the memory bank and is spied by the communication interface to the neighbour nodes. Single and double-bit error detection and single-bit error correction circuitry are also provided (the check word is 7-bit long), since it is crucial to keep the reliability of the whole system within reasonable limits.

A final feature of the MAD chip is a condition code handling scheme that enhances the basic SIMD architecture, introducing a parallel conditional structure (called WHERE(cond) THERE in APE jargon, for instance ref. [6]). All instructions inside the WHERE block are executed only in those processing nodes where the relevant condition is true. The remaining nodes must perform no-ops instead. This is obtained by freezing all memory and register writes on these nodes and is controlled by a condition-code stack, which allows nested conditional constructs. The latter is included in a simple stack-machine, which allows the computation of complex conditions (e.g. WHERE $(A > B$ and $C < D)$ THERE, etc.).

A last point to consider is the strategy used to control the device. The usual approach for this purpose starts with the definition of a set of machine-level instructions used in assembly programs. Machine instructions are decoded and expanded into elementary steps inside the device. A simpler approach provides control bits to all sub-components at each clock cycle. The break-up of each macro instruction into elementary actions becomes in this case a software task, performed at some level of the compilation chain. This approach has two main advantages: first, the hardware structure is simplified, since instruction decoding is no longer necessary. More important, very efficient code optimization can be performed in this way. The large size of the instruction word (48 bits, in our case) is not a serious problem in a SIMD machine, where all processing elements share the same code.

5. Technology issues for the MAD chip

This section discusses the technology constraints that have been taken into account in the design of the MAD chip, and presents some estimates of the level of performance expected for the circuit.

The first point concerns the electronic technology chosen to implement the MAD chip. We have selected a CMOS technology for several reasons. It offers the highest gate density available and is the easiest to integrate in a complex system, thanks to its high-noise immunity level and low-power consumption. CMOS technologies also offer the crucial advantage that simple but reliable simulation models have been developed, so the behaviour of a complex circuit can be tested before being fabricated. Finally, efficient semi-custom design tools are available for this technology.

In principle, once the basic architecture of the chip has been selected, chip performance has to be maximized. Generally speaking performance can be increased with the use of more complex (more parallel) circuitry, and increasing the number of pipeline stages. On the other hand, as chip complexity is increased, the area of the device becomes larger and interconnection delays also grow, until some limits are reached that are set by the technological process at hand. These figures also depend strongly on the design style (full custom vs cell based) that is used. Full custom design maximizes speed and minimizes area, while cell-based design is slower and uses more silicon area, but allows a very fast design style. A reasonable compromise has to be set on all these points.

Our target performance has been set by the requirement that the MAD chip interfaces easily with standard 4-Mbit DRAM (Dynamic Random Access Memory) chips that have an access time of ~ 80 ns. This requirement sets the clock rate of the device at $12.5 \times N$ MHz, where N is an integer number (preferably a power of two). In principle, N can be made rather large, if enough pipeline stages are inserted in the data path. A reasonable value of N is suggested by some rough estimates of a few simple figures of merit for the used technology [7].

Let us assume that the average switching time for a typical gate is T_0, and that the clock period is $T_c = n\, T_0$, that is, about n gate levels can be squeezed in each pipeline stage. Some time is also used up in register transfer between stages. Let us write the register transfer time as $r \times T_0$. Then only $n - r$ gates are left for useful work. This means that the gate efficiency of the device is

$$\varepsilon = (n-r)/n\ , \tag{1}$$

while the required silicon area increases (under the optimistic assumption that interconnections scale as the gate count) by

$$R = n/(n-r)\ . \tag{2}$$

Also, if the longest combinatorial path traverses L gates, the number of pipeline stages required is

$$k = L/(n-r)\ . \tag{3}$$

These simple relations show that there is a practical performance limit inherent to the used technology, beyond which a small increase of speed requires a much larger silicon area and much longer pipelines (when eqs (2) and (3) start to depart from linearity (fig. 3)).

We use a 1.2 µm CMOS technology[(*)] with $T_0 \approx 0.9$ ns, and $r \approx 9$. Corresponding values for ε, R and k are given in table 1.

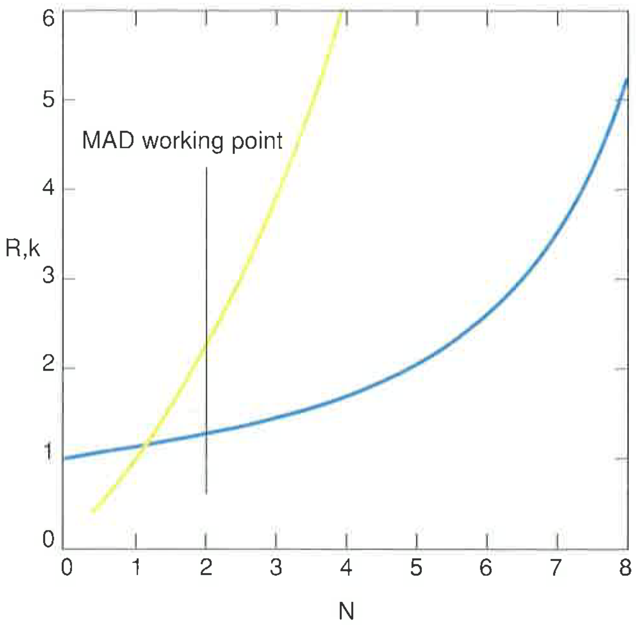

FIGURE 3

Estimated silicon area and pipeline count vs clock frequency.

(*) Our chips are manufactured by European Silicon Structures (ES2) Paris.

Table 1

N	ε	R	k (normalized)
1	.899	1.11	1.00
2	.797	1.25	2.25
4	.595	1.68	6.04

These numbers clearly show that the best-suited clock frequency for our technology is at some value of N slightly larger than 2, but definitely smaller than 4 (fig. 3). These considerations suggest a clock period of 40 ns. Reasonable values for the pipeline length of the main arithmetic blocks are obtained with this clock rate, as detailed in sect. 6.

Let us now consider the design style for our circuit. In our 1.2 μm technology, the smallest inverter gate can be built in an area of ~ 250 μm^2, that is, 800 k transistors/cm^2. In real life, a full custom design might reasonably be half as dense, while a cell-based design could have between 100 k and 200 k transistor/cm^2.

A cell-based design seems appropriate for MAD, since a floating point unit is not a very regular design and the estimated gate count for all main blocks is not too high (for example the multiplier and ALU require ~ 20 k transistors each) if single precision only is implemented.

This is not true, however, for the register file, which has been designed as a full custom block. On the one hand, the register file is a very regular structure that can be built by abutment of a small number of basic layouts. On the other hand, minimum dimension transistors, not available as standard cells, are required in the memory core. A rough estimate shows that a cell-based register file would be nearly ten times larger than our custom design.

6. The MAD implementation

This section presents some details of the actual implementation of MAD as a CMOS integrated circuit.

A detailed block diagram of the MAD chip is shown in fig. 4. Basic elements of the circuit include the main arithmetic data path, the register file, the look-up tables, the status register, the condition-code block and the interface to the external bus, including the error detection and correction circuitry. A representative selection of these elements is described in the remainder of this section. Full details can be found in ref. [8].

6.1 The MAD path

The MAD path is a 3-input block, containing a floating point multiplier and ALU. The standard operation performed by this block (called the "normal" operation, in the APE100 assembler [6]) is $d = a \times b + c$.

The multiplier always computes the product of a and b, while the ALU is able to perform all eight combinations of additions and subtractions (and the corresponding absolute values) of two inputs.

Floating point arithmetics follows a consistent subset of the IEEE floating point standard [9]. The published standard is not fully adhered to, since:

(a) rounding to nearest is always performed,

(b) denormalized numbers (smaller than the smallest representable value) are considered as true zero,

(c) no distinction is made between the IEEE representation of infinity and the so-called not-a-number.

Full details on the arithmetics performed by MAD are given in ref. [8]. Here, we only remark that we have succeeded in maintaining almost complete consistency with the IEEE standard for non-exceptional inputs and results, while discarding all those features needed for general exception handling. They are not needed in our case, and would greatly increase circuit complexity. At an early stage of the project we considered the possibility to build a combined multiplier adder with rounding applied only after the adder stage. This scheme is simpler and faster from the hardware point of view and also provides marginally higher numerical accuracy, but we decided to implement independent rounding after both multiply and add, in order to keep compatibility with old APE arithmetics and standard IEEE hardware.

The multiplier has five pipeline levels, while the ALU requires just four stages (the operand to result round-trip is eleven stages long, including register access). These numbers do not appreciably degrade the performance of typical floating point intensive blocks of program.

6.2 Register file

The register file used in MAD contains 128 32-bit registers. Register contents can be read from three ports at each clock cycle, to provide operands for the arithmetic block. Results can also be written at each clock cycle on one write port. These four ports are used concurrently during the execution of typical programs. Additional and independent write and read ports are used to load or store data from/to the external bus.

The register file is the single largest block of the MAD chip, in term of transistor count (it contains ~ 50 000 transistors). It had to be designed as a full custom block, developed for us by ES2, in order to keep its size within reasonable limits.

6.3 Look-up circuitry

The basic ideas for the design of the look-up circuitry are given here. Full details can be found in ref. [10].

In the case of the inverse and inverse-root functions, MAD provides a look-up table for a first approximation to the result, with 3-bit accuracy in the mantissa. A correct result (± 1 least-significant bit) is obtained with four iteration steps. This choice represent a substantial saving in hardware cost compared to standard solutions, which use 8 bits into 8-bit PROMs.

The exponential function is evaluated in the following way. Note first that exp(x) is equal to 2^z with $z = x \log_2(e)$. If we write $z = \mathrm{Int}(z) + \mathrm{Frac}(z)$, then $2^z = 2^{\mathrm{Int}(z)} \times 2^{\mathrm{Frac}(z)}$.

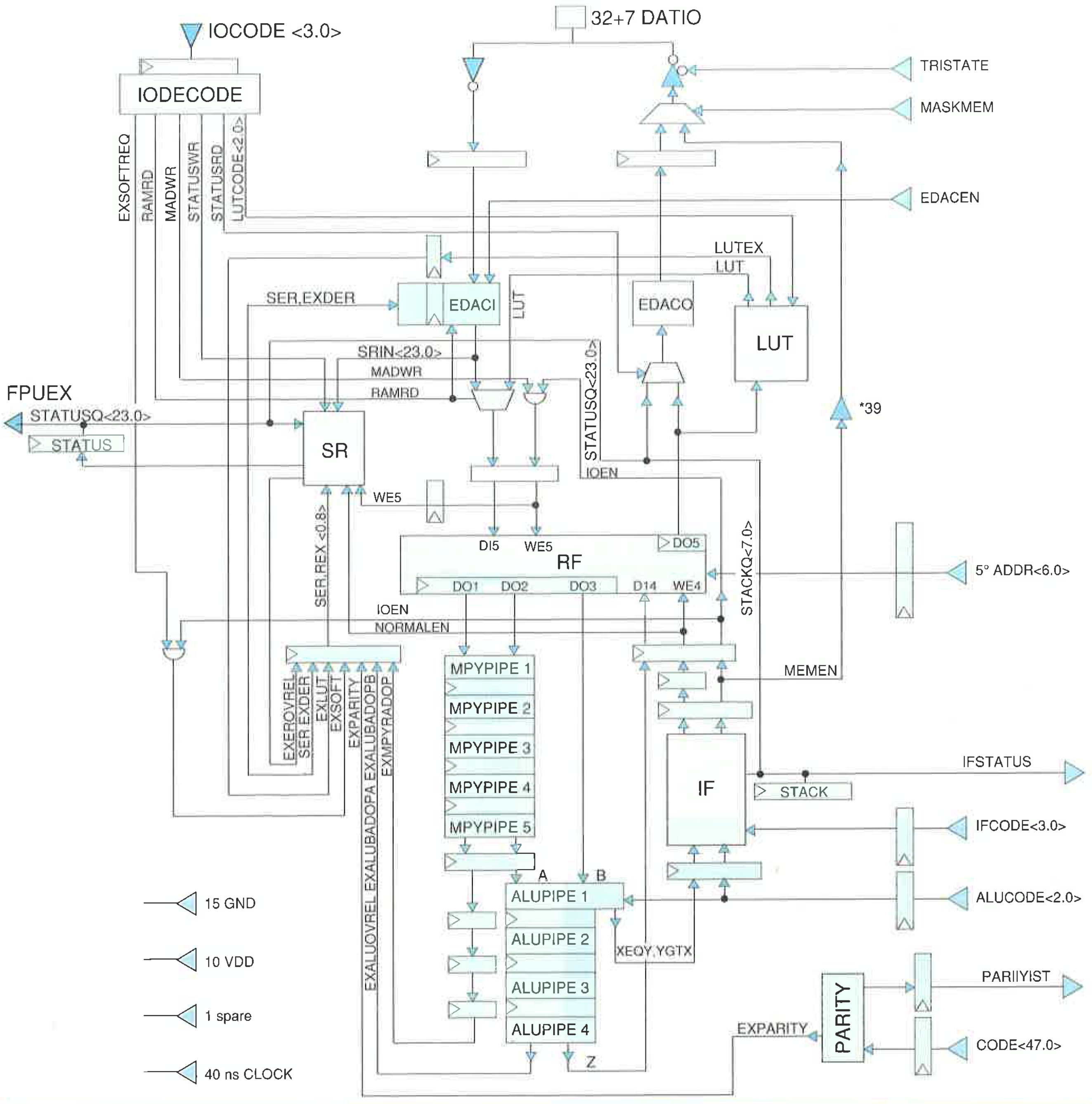

FIGURE 4

Detailed block diagram of the MAD chip.

The floating point representation of $2^{\mathrm{Int}(z)}$ is simply the integer value of Int(z) + 127, shifted in the exponent field of the floating point representation. It can be readily seen by inspection of the IEEE standard that the required integer value is contained in the lowest-significant bits of the floating point number $F = z + M$, where M is a magic number corresponding to the floating point representation of $(2^{23} + 127)$. It can also be seen that $\mathrm{Int}(z) = F - M$, and $\mathrm{Frac}(z) = z - \mathrm{Int}(z)$.

The computation can be completed at this point by an improved series expansion for $2^{\mathrm{Frac}(z)}$. Similar tricks are used for the log function.

6.4 Error detection and correction

Data words are written to (read from) memory together with a 7-bit check pattern, allowing single-bit error correction and double-bit error detection (this is usually known as EDAC capability). Correction codes are necessary in the APE100, since the mean time between failures for uncorrected soft memory errors is estimated at ~ 80 h. Our EDAC strategy raises this number to the level of more than 100 days. The EDAC strategy is compatible with the industry standard algorithms used for instance in the 74LS632 integrated circuit. Inclusion of the EDAC circuitry in MAD is a typical example of tailoring our device to the needs of a massively-parallel processor. Indeed, we have added some badly-needed additional circuitry to our chip (adding just a few percent to the total gate count or silicon area) that would require almost as much board space as MAD itself.

MAD chip prototypes were produced in fall 1990. The final chip layout has a size of ~ 9.4 × 9.4 mm^2. It contains ~ 140 000 transistors and 20 000 standard cells and uses 118 pins, including ground and supply lines. The register file uses ~ 30% of the chip core, while the rows of a standard cell cover about half the core area. Finally, 15% of the device is used for routing avenues.

Figure 5 shows the cell array and fig. 6 depicts the two metal interconnect layers.

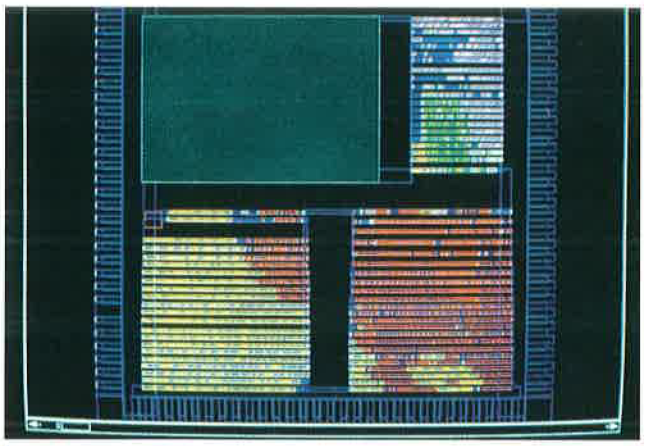
FIGURE 5

Cell layout of the MAD chip. Colour codes are: grey register file; red multiplier; yellow ALU; pink EDAC; and green condition-code stack machine.

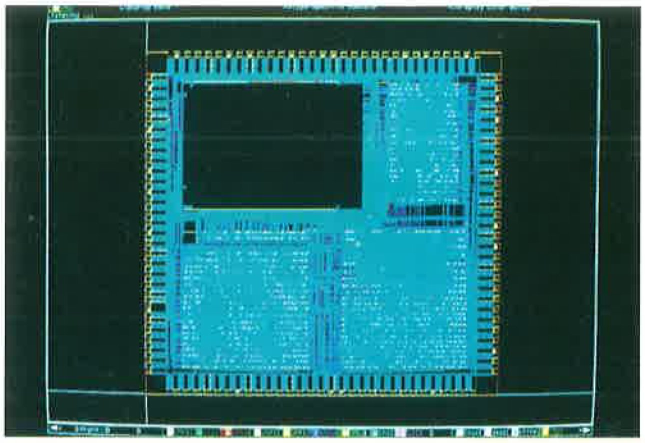
FIGURE 6

Layout of the MAD chip, showing the two metal interconnect layers.

7. Design methodology

In a complex project such as MAD it is essential to be able at each step to gain confidence in the correctness of the design, and in its functionality. In the case of a programmable device, the best strategy is that of building a software model, a simulator, which can be used to execute real-life programs. At a first stage, the behaviour of the simulator can be used to assess the merits of particular design choices. In this respect, it was essential that a compiler for the new machine, derived from the original APE compiler, was available at an early stage of the development. We have thus been able to run long programs (tens of thousands machine instructions), many of which actually are chunks of LGT simulation programs. The output has been compared bit by bit with those of corresponding code running on an IEEE compatible workstation.

At a second stage the simulator was used to check the consistency of the electrical design and to prepare and verify a set of test vectors also produced by compiling real-life programs. In this way we have a triple set of cross checks: the test vectors corresponding to a given program can be fed to a logical simulator of the electrical design. The resulting output can be compared with both the results produced by the software simulator and those obtained by a commercial machine. Furthermore, we can compare the state of internal signals on the electrical design and on the software simulator.

8. Conclusions

Given the powerful design tools now available, including libraries of standard cells, it is now possible for a small group to design powerful custom chips.

The lesson to be learned is that in our case a custom processor proved the most economical solution for obtaining the building block for our massively-parallel machine. Any standard solution would have required four or five chips per processing element, and we would not have been able to assemble many nodes (eight) on a single board. The same approach will certainly prove useful for other applications, especially in high-energy experimental physics, in the frequent cases where nonstandard processors are required in a large number of copies.

The CAD design techniques now available have made it possible to complete the MAD design in ten months, with an effort of about four man-years. A crucial part in the success of the work has been the availability of a compilation chain for the new device which has been derived from that written for our previous APE parallel machine, which shares most of the key architectural features with the APE100. The effort dedicated to the software development represents an excellent investment, since the compiler used for testing during the design stage will be the basis of that used for the real machine.

Acknowledgements

We cannot even try to list all those people that have contributed suggestions and ideas for this project. E. Marinari, J. Pech and F. Rapuano have contributed to the early phases of the project, while L. Elman and A. Vadillo have been precious for their work within ES2. Finally, we warmly thank G. Parisi who helped shape this project with his many ideas and continuous support.

References

[1] R. Tripiccione, Nucl. Phys. B (proc. suppl.) 17 (1990) 137.
[2] The APE Collaboration, Roma preprint 733 (1990);
W. Tross, Status of the APE project, to be published in the proceedings of LATTICE 90.
[3] M. Albanese et al., Comp. Phys. Comm. 45 (1987) 345.
[4] P.B. Mackenzie, Nucl. Phys. B (proc. suppl.) 17 (1990) 137 (and references therein).
[5] H.S. Stone, High-performance computer architectures, ed. Addison Wesley (1987).
[6] The APE Collaboration, preprint A100/DOC/S01 (1990).
[7] C. Mead and L. Conway, Introduction to VLSI systems, ed. Addison Wesley (1980).
[8] The APE Collaboration, preprint A100/DOC/H01 (1990).
[9] ANSI/IEEE Standard for Binary Floating Point Arithmetic, No. 754 (1988).
[10] The APE Collaboration, preprint A100/DOC/G01 (1991).

Addresses:

A. Bartoloni, C. Battista, S. Cabasino, F. Marzano, R. Sarno, G.M. Todesco, M. Torelli, W. Tross and P. Vicini.
INFN, Sezione di Roma and Dipartimento di Fisica
Università degli Studi di Roma, La Sapienza
Piazzale Aldo Moro, 2
I–00185 Roma (Italy)

N. Cabibbo, P.S. Paolucci and G. Salina.
INFN, Sezione di Roma and Dipartimento di Fisica
II Università di Roma, Tor Vergata
Via Orazio Raimondo
La Romanina
I–00173 Roma (Italy)

F. Del Prete, R. Tripiccione and E. Zanetti.
INFN, Sezione di Pisa
Via Livornese, 582A
San Piero a Grado
I–56010 Pisa (Italy)

Received on 17 April 1991.