

Design and implementation of a massively parallel architecture for high-level electronic measurements

Alessandro Gandelli* and Vincenzo Piuri†

Dedicated architectures are often a basic requirement to match massive computation and throughput of an increasing number of real-time applications in electronic measurements, in particular for robotics and power applications. Also, a high-level programming view of the architecture and an interactive interface become important to allow easy implementation of complex algorithms, reasonable access to measurement experts without a specific expertise on computers, and a flexible adaptation to the specific application environment. This paper presents the design and the implementation of an advanced measurement system based upon a multiprocessor architecture. Particular attention is given to the programming environment.

Keywords: multiprocessor dedicated architecture, high-level electronic measurements, multiprocessor design

Nowadays, many applications in electronic measurements need massive computation and throughput, in particular for real-time environments¹ such as in power-distribution systems, in fully automated factories and in robotics. Since traditional computer architectures are not able to satisfy these impressive constraints at reasonable costs, it is necessary to adopt new strategies based upon dedicated hardware/software architectures. The use of DSP processors, possibly in parallel configuration, can be considered to provide the required computational capabilities; however, their cost and the complexity of system control are quite high and their capabilities are not fully exploited in the envisioned application areas. Recent advances in VLSI integration technologies and in architectural design

provide the bases for an effective dedicated approach, whose costs are acceptable with respect to the specific application environments. Since experts in measurement (both at theoretical level and in industrial applications) are not always experts in computer architecture and programming, it is mandatory that advanced measurement systems based upon computers have high-level interfaces to hide details of the underlying computational system and present a flexible interactive environment for measurement.

The design of a new integrated environment for real-time measurement of electrical signals has been investigated by the authors to match all the requirements briefly described above. The overall architecture and the user interface are presented in this paper: the main focus is on the characteristics of the software environment, on the flexibility of the system and on the programmability of the measurement algorithms. As test-beds of our measurement system we considered some of the most critical, highly computing applications: namely, the global management of different kinds of sensors in advanced robotic systems, and the real-time analysis and control of perturbations in power-distribution networks.

From the architectural point of view, our measurement system is composed of a SIMD machine for parallel computation and a front-end computer to support user interface and multiprocessor control. The use of a SIMD architecture was suggested by the characteristics of the specific applications envisioned in our research; in fact, our measurement systems have to deal with multiple data flows by performing the same sequence of instructions on each such data flow in parallel with the others. The software environment has been designed as a hierarchical structure presenting different views of the multiprocessor system at increasing abstraction levels. Hardware parallelism guarantees massive throughputs for high-level measurements,

*Department of Electrical Engineering, Politecnico di Milano, piazza L. da Vinci 32, I-20133 Milano, Italy

†Department of Electronics and Information, Politecnico di Milano, piazza L. da Vinci 32, I-20133 Milano, Italy

Paper received: 9 August 1993. Revised: 15 January 1995

while the user-oriented programming environment provides a friendly and efficient interface which fits the knowledge and the practice of different users.

The hardware architecture is briefly described in the next section, while the overall structure of the software system is introduced in the following section. The three-layer software architecture has been designed and implemented to provide three different abstraction views of the multiprocessor through a flexible programming environment. The low-level programming approach is then presented: it is based upon the multiprocessor assembler EMMA and the run-time mathematical library. Optimization of the assembler code may be important to improve the performance of the measurement system by fully exploiting the available parallelisms; the following section discusses the identification of sequences of independent instructions and proposes two approaches for automatic parallelization by considering the specific multiprocessor architecture. One of these techniques considers adjacent instructions in the source code to verify the capability of concurrent execution. The second approach is based upon restructuring of the sequences of instructions through inspection of the computational graph which can be associated to the user program.

A second level of the user interface is provided by a traditional high-level programming language, as described in the next section. At the moment an interface based upon the C programming language has been developed, and work is in progress for the Fortran language. Optimization is still available at assembler level: we are also studying features offered by an optimization approach to a higher programming level, i.e. dealing with a higher abstraction view of the system. Libraries for the most popular operations and transformations are provided to simplify multiprocessor programming.

A higher view of the measurement system and the computational paradigms is the graphic interface GPE. The next section presents in detail the characteristics of this interface, which constitutes an original approach to system programming. Computation is in fact defined through iconic programming: the user selects the functional building blocks of the transfer function of the virtual measurement instrument and connects such blocks in the proper computational graph. In this way the user defines the basic operations which must be executed in the measurement algorithm, the data flow and the activation order. Graphic description of the measurement program is performed by using an interactive interface based upon icons and guided menus. Functionalities of the virtual instruments are then mapped automatically on the structures and operations of the multiprocessor architecture. This approach is very powerful since it allows fast and easy definition and experimentation of new virtual instruments.

Finally, we briefly review the interactive environment for a user-friendly implementation of a specific application. We present two applications of our multiprocessor architecture to experimental real-time systems: the first concerns electronic measurement and control of an automated plant for power distribution, while the second is directed to multisensor management in advanced robotics.

DEDICATED MULTIPROCESSOR SYSTEM

The multiprocessor architecture presented in this paper has been designed to match the specific requirements of advanced measurement applications. The processors have been studied to achieve the best performance not only for some traditional arithmetic operations, but also for a number of high-speed digital signal processing computations, which are typical of most of the measurement algorithms.

The overall structure of our architecture is given in *Figure 1*. It consists of a multiprocessor subsystem for parallel computation and a front-end computer for user interaction and control. Our architecture behaves basically as a co-processing unit of the front-end computer, since it may be used to extract some characteristic figures from a large amount of input data by means of traditional signal processing algorithms. This pre-processing activity greatly reduces data that must be treated by the front-end computer, in particular for applications in electronic measurements and automatic control. Details of the hardware architecture were presented in Reference 2.

The front-end computer is a microcomputer based upon a set of Motorola's 68020 boards with a standard Unix environment to support user activities. It provides a high-level human interface for designing, testing and executing the application-specific measurement algorithms. In particular, the front-end computer has been designed to support loading of the measurement algorithms in the multiprocessor memory and controlling the parallel computation. The multiprocessor subsystem and the front-end computer are connected to a standard VME bus and to a dedicated control bus for data and control exchange.

As already said, the multiprocessor subsystem is a SIMD architecture, in which a single instruction is performed at the same time by all the processing units²⁻⁴. The main goals we achieved in designing this multiprocessor architecture are performance, modularity, expandability, flexibility and adaptability. Performance is an obvious requirement to implement structures for massive computation. Modularity makes it possible to achieve a compact standard architecture in which the component can be modified and improved according to the requirements of the specific applications. Simple and fixed interfaces,

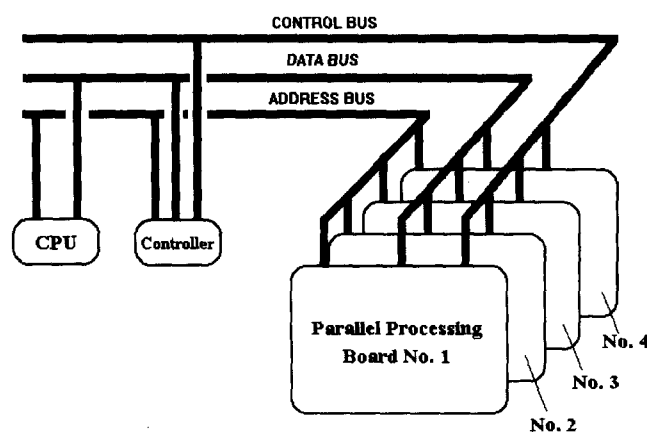


Figure 1 Multiprocessor architecture

interconnections and protocols guarantee the expandability of the structure (at least in a reasonable range) by the addition of identical modules to match the algorithm characteristics. Flexibility and adaptability of the architecture to a large class of measurement algorithms is needed to implement a unique computing structure that may be used for a number of different applications. Adaptability should hold also at run-time by means of software reconfigurable interconnections to guarantee a high availability and flexibility of the system. Many of these features have been achieved by adopting a simple, clear, high-speed, modular architecture based upon standard components.

The multiprocessor subsystem is composed of identical processing units and a controller unit. The processing units provide the parallel computational power by working on different data sets contemporaneously. The control unit has been introduced to overview the multiprocessor activity by imposing the sequence of instructions that must be performed by all processing units in parallel. The number of processing units depends on the specific measurement algorithm required by the user.

Each processing unit is composed by two boards: the ALU (arithmetic logic unit) board (see Figure 2a) and the local memory board (see Figure 2b). The ALU board is the heart of the computational structure to implement high-speed algorithms for electronic measurement. The memory board is basically used to store measurement data coming from the external environment, during execution of the application-specific algorithm. Efficient data transfer inside each board is guaranteed by an internal high-speed bus for data and addresses. Our implementation of the experi-

mental multiprocessor, discussed in this paper, considers arithmetic units for 16-bit data and operates at 10 MHz to avoid hardware complex structures and to provide high flexibility, powerful computational capabilities and high-throughput data management at reasonable costs. The low-frequency system clock (low with respect to the current standards used in microprocessor systems) has been adopted to satisfy both the requirements of the AMD 29000 device family, the maximum data transfer rate for the digital input channels, and cost minimization. On the other hand, a higher clock rate does not seem—at this time—useful for our specific applications.

The ALU board is a custom structure for execution of basic arithmetic operation and some complex algorithmic kernels of digital signal processing. It is based upon AMD's 29501 ALUs and AMD's 29517 multiplier. Two three-port VLSI ALUs perform the computation on 16 bit data using a multibus and multiregister architecture. A high performance 16×16 bit multiplier completes the arithmetic hardware and provides the required computational support for a high-speed implementation of FFT algorithm. The two ALUs can perform multiple data operations during the same clock cycle using their multiport structure (two input-output ports and one input port) and the six register operating independently by the internal ALU and MUX registers.

The memory board has been designed to obtain the best performance and to reduce data transfer time without limiting system capabilities. This board is strictly connected to the ALU board. Nevertheless, it has high autonomous capabilities for data management. A double memory bank has in fact been developed to allow contemporaneous memory management from both internal processor resources and system ALU. Every bank is formed by four 8×32 kbit memories connected to the internal 32-bit bus. These two buses can be alternatively switched to one of the three possible connection paths (ALUs, direct I/O, system bus).

The multi-access structure and management of input data and memory banks are the most important novelty of our architecture. Two independent data paths are provided by the memory board in order to exchange information with the external environment:

- the main data path, which is connected to the system bus;
- the secondary data path, which provides a direct input bus for data acquisition by using the DMA technique.

While processors are working on odd banks, the host computer performs data transfer operations from/to the main memory for the even banks, and *vice versa*. The switching system provides the control of active paths in order to avoid bus conflicts. Read operations (performed by the front-end computer) and DMA acquisition from a secondary data port can be performed while the processor is working on the other bank because their execution time is smaller than the time required to complete a Walsh or a Fourier transform. The various possibilities of overlapping data input and internal computations greatly increase the throughput and the computational power of the system to

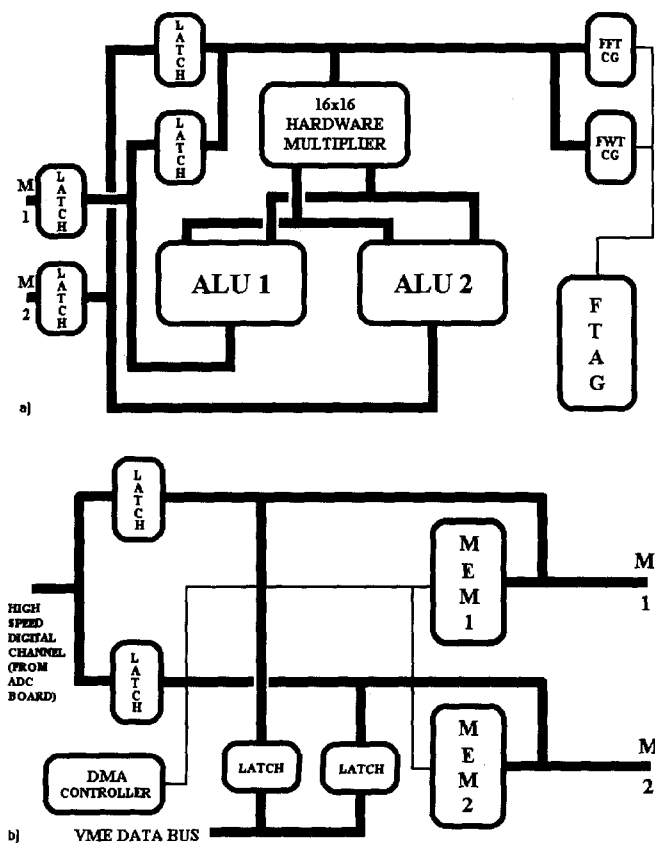


Figure 2 Structure of the processing unit: a, ALU board; b, local memory board

match the massive requirements of real-time applications in measurement and control. While one half of a processing unit is computing the nominal algorithm, the other is supporting the input/output mechanism.

Parallel computation and management of the processing unit boards must be performed by activating the proper control signals on the dedicated control bus. Due to the complexity of the different activities that can be executed in parallel and to the number of control signals, direct control from the front-end computer is not feasible. For this reason we introduced a control unit to supervise and manage all processing units and we defined a set of instructions that can be executed by our architecture. Each instruction corresponds to the compact coded representation of the control signals that must be activated to impose execution of the desired operation. A program for the multiprocessor system consists of a sequence of such instructions.

The control unit is composed of the program memory and the processing controller. The program memory contains the sequence of instructions, while the processing controller performs the fetch and decode operations of the multiprocessor program.

The processing controller is an AMD 2910 microcontroller with its own control circuits, private memory and latches. The microcontroller fetches and decodes the operations for the multiprocessor. For each new instruction, it acquires the coded representation from the program memory and identifies the current instruction. Then, it generates the proper control signals, which correspond to the current instruction, by looking at them in its own private memory. Finally, the microcontroller moves to the next instruction. Due to the modularity of the multiprocessor architecture, in the second release of the control board, the microcontroller will be substituted by a microprocessor (in particular, a Motorola 68040); when properly programmed this microprocessor will be able to generate autonomously sequences of multiprocessor instructions. In this case the controller unit will behave as a traditional microprogrammed unit.

The program memory is composed of 12 8 × 4 kbit high-speed static CMOS RAM. In this memory, we store the sequence instructions for the multiprocessor, which are 29 bits long. The memory is loaded by the front-end computer through the system bus before computation is started. The 96-bit control signals, generated by the program controller in correspondence to each multiprocessor instruction, are imposed on every processing unit by using the dedicated control bus.

The front-end computer loads the desired control program in the memory of the multiprocessor subsystem; then, the front-end computer enables the autonomous computation of the multiprocessor by stating the proper signals to the control unit. Results computed by the multiprocessor are stored in the local memory of the multiprocessor itself: the front-end computer may read them through the VME data bus and provide subsequent manipulations (e.g. presentation to the user, storing in mass-storage units, generation of control activities in automatic control systems).

PROGRAMMING ENVIRONMENT

The efficiency of advanced multiprocessor architectures is often greatly reduced since the user is generally not familiar with parallel programming and, as a consequence, is not able to exploit fully the architecture capabilities. To avoid this drawback and to give practical and effective access also to non-expert users, an integrated user-oriented environment has been designed and implemented.

To achieve this goal, we designed a flexible user-friendly interface which has been hierarchically structured into layered tools. Each layer has been introduced to satisfy the requirements and habits of different users, with particular attention to their knowledge of the multiprocessor architecture, their capabilities of parallel programming and their ability in computer programming. Three programming levels are available:

- the assembler view,
- the view from a high-level programming,
- graphic programming.

They correspond to an increasing abstraction of the view of the hardware architecture. At the lowest level the complete details of the structure and functioning of the system must be accurately understood to program correctly the sequences of operations. At the highest level, the computation is viewed only as execution of transfer or transformation functions on the input data and on the internal state. On the other hand, abstraction implies a possible reduction in optimization capabilities or, at least, expensive design and implementation of optimization tools. The user can, however, choose the level that best matches his knowledge, his requirements and the characteristics of the specific application.

In each layer we provide different tools to support design, development, testing and execution of algorithms on the multiprocessor architecture. Details and features of each tool will be discussed in the subsequent sections.

A general schema of the software environment is shown in Figure 3. At assembler level we provide the following tools:

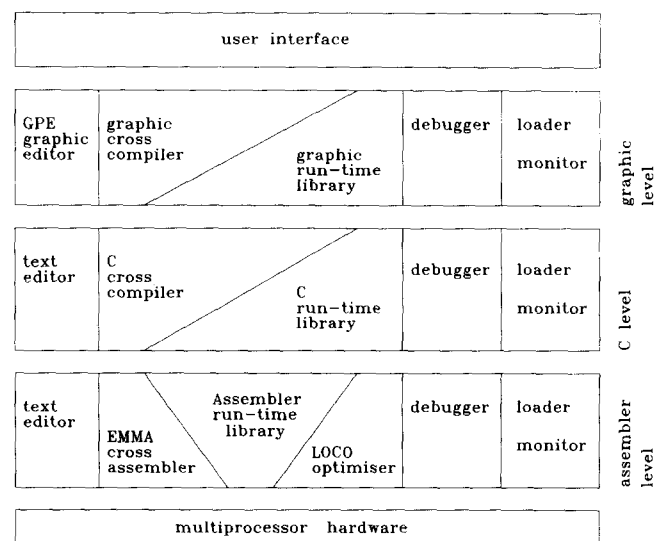


Figure 3 Software architecture

- the multiprocessor assembler EMMA and the assembling program,
- the run-time library which guarantee full, optimized access to the architecture,
- the application loader and monitor,
- the assembler debugger to check interactively the execution of the algorithm.

At the intermediate level, the high-level programming language supplies an abstract view of the architecture still considering traditional techniques of computer programming. At this level, we introduced:

- a subset of the C programming language and the corresponding compiler,
- the run-time library for C language,
- the program loader and monitor,
- a compact version of a symbolic debugger for the C language.

At the most abstract view, namely the graphic programming, we designed the following innovative tools for measurement systems:

- the graphic editor GPE for graphic definition of measurement algorithms,
- the graphic compiler to generate an executable version of the algorithm,
- the graphic library for algorithm definition,
- the program loader and monitor.

Since neither segmentation of the program memory nor dynamic storage allocation for data are available, no dedicated management tool for segment loading and memory allocation is provided. Also, since dedicated applications are considered, the software environment on the multiprocessor architecture is a traditional monoprogrammed system: neither multiprogramming nor multitasking onto the SIMD multiprocessor is presently available. Multitasking is meaningless for the applications of our architecture since data cannot be mixed and distributed to different concurrent algorithms. Conversely, multiprogramming could be useful to apply different algorithms iteratively to the input data stream in order to extract different characteristics, without reloading all programs: in this case, it could be included in the second release of the software environment, when the microprogrammed release of the controller unit will be able to support this feature.

To simplify the user interaction with the software environment, we introduce an integrated presentation framework in which it is possible to select, execute, monitor and observe different activities by means of guided menus and windowing. All previous tools were developed and integrated in the standard Unix environment of the front-end computer: the multiprocessor architecture is only the executor of programs loaded by the front-end computer.

ASSEMBLER VIEW

Implementation of algorithms for the multiprocessor architecture can be realized by sequences of instructions written in the multiprocessor assembler language EMMA (Electro-

nic Measurement Multiprocessor Assembler), a dedicated language which is particularly tailored to exploit the available parallelisms in our multiprocessor architecture. A complete definition of the assembler is given in Reference 5: in this paper we briefly review only the basic characteristics and features.

As in traditional assembler languages⁶, an assembler instruction is the n -tuple (*op-code*, *op-1*, *op-2*, ... *op-n*) where *opcode* is the operator (i.e. the symbolic identifier of the desired operation), while *op-i* are the operands. For simplicity, all instructions have at most two operands: the source and the destination. When two-operand instructions are considered (e.g. addition or multiplication), the source specifies one of the operands, while the destination identifies the second operand and the storage for the result at the same time.

In the present implementation of EMMA, we defined instructions to execute both traditional computing activities, architecture-related instructions and measurement-specific operations. Instructions may be divided in the following groups:

- data-transfer operations allow for storing and retrieving data in the memory storage, for moving data between internal registers, for writing and reading data of input/output channels;
- arithmetic-logic operations;
- trigonometric and logarithmic operations allow for generating efficiently the basic results and coefficients for digital signal processing;
- FFT operation generates the data addresses for the transformation algorithm;
- instructions for program control flow.

One of the main goals in designing the EMMA assembler was the identification of the minimum set of operations which guarantee complete exploitation of the hardware characteristics of the multiprocessor architecture, while preserving the simplicity of programming and the reduced number of instructions. To such purpose, we minimized the number of different instructions by grouping similar operations. For example, all data transfers are executed only by the instruction MOVE, without regard for the characteristics of source and destination.

Most of the assembler instructions are translated into a single machine operation. However, some basic operations (e.g. data transfer from a location of the local memory to another location) cannot be directly supported by the hardware in a single step, due to the characteristics of the processing units and the interconnecting buses. Since the present release of the controller unit is not microprogrammed and behaves as a sequencer, it is necessary to translate these instructions into a sequence of single operations.

Different addressing modes are introduced in the assembler to guarantee enough flexibility both at architectural and at programming levels. At the moment, we support the following definitions inside each processing unit: immediate data, register data, direct addressing in memory, indirect addressing by using the value stored in registers or storage locations, and automatic addressing identifying the operands by using the FFT address generator.

For readability reasons, the user is allowed to define symbols to substitute numeric values. Two applications of the use of symbols are allowed: the identification of an instruction (i.e. a program address) and the definition of constant values.

Translation of the source program into the machine code (executable by the multiprocessor) may be achieved by using the translator (the assembling program) on the front-end computer. This tool also allows for creating a complete listing of the program itself, a symbolic cross-reference table, the allocation map and the executable image supporting debugging.

No optimization of the parallelism is automatically performed by the assembling program. However, the grouping assembler directive (i.e. the symbol '+' at the beginning of the instruction that should be grouped with the instruction above) allows for directing the assembling program to compact a group of assembler instructions into a single executable instruction, if there is no interference on the required hardware resources.

To improve efficiency and development time of new application programs at assembler level and to exploit the capability of parallel execution, we designed and implemented a modular, expandable run-time library. It contains the most used basic mathematical functions (e.g. trigonometric, exponential and hyperbolic functions) and advanced DSP operations (e.g. DFT, FFT, HFT, convolution, correlation).

Loading, monitoring, debugging and executing the machine code may be performed by interacting with the front-end computer through a user-friendly environment.

OPTIMIZATION OF THE ASSEMBLER CODE

The instructions available in the assembler language are simple and similar to those of traditional monoproductors. In the assembler language we did not include directly any multiple instruction, i.e. any set of instructions that can be executed in the same cycle since they require disjoint sets of hardware resources. Moreover, it often generates wasting of time due to the additional programming efforts required by the user. To avoid such drawbacks and to simplify multiprocessor programming, we studied the automatic optimization for compacting sequences of instructions and to exploit the capabilities of concurrent execution.

Several approaches have been proposed in the literature in the last two decades to afford optimization of compiled code in traditional, parallel and vector architectures (e.g., see References 7 and 8 for an overview). All of them are based on (or can be reduced to) the analysis of the program computational graph derived from the sentences of the program itself (possibly at different abstraction levels, e.g. source code in a high-level programming language, intermediate code, executable code). Partitioning and restructuring of the algorithm operations, in accordance with the specific hardware architecture constraints, lead to the identification of the capabilities of concurrent execution. The main differences consist of the specific techniques adopted for optimization and graph analysis.

First, for our multiprocessor system, we consider a very simple approach based on compression of adjacent instructions^{7,8}, since it has been proved quite effective while it has a straightforward implementation. The basic idea consists of scanning the assembler program sequentially and of trying to 'squeeze' each instruction into the same execution cycle of the previous one, if no conflicting access occurs and the required data are available. Following this approach, we implemented the local compactor LOCO.

At the beginning of the program scanning or whenever a group of concurrent instructions has been completed, the compactor views all hardware resources marked as available. It observes the first instruction that should be executed at the current position in the program sequence. For such instructions, the compactor marks 'busy' the hardware resources required by the instruction itself. If the instruction is a control-flow instruction, no hardware resource in the multiprocessor is strictly required. However, the current group of concurrent instructions, which the compactor is building, must be considered completed. In fact, the execution of an imperative control-flow instruction (a call or a jump without conditions) implies implicitly the non-execution of the instruction immediately below in the program source code. Besides, when a conditional control-flow instruction (a conditional jump) is concerned, we cannot predict statically (i.e. before executing the program) if the instruction below will be executed; therefore, the group of concurrent instructions must be considered as completed. Then, the compactor examines the next instruction in the program source code: the same activities are applied to the new instruction. If busy hardware resources are required, the instruction cannot be executed concurrently with those previously inserted in the current group. This group is considered completed and a new group is opened. Such searching continues until all the instructions of the program have been examined.

Due to the characteristics of the hardware architecture, it is possible to verify that we can define groups containing at most eight concurrent instructions: this upper bound can be achieved only for some combinations of instructions. Experiments have shown that usually we can compact two (at most three) instructions in a single cycle and the number of instructions may be reduced of about 30%.

To identify the capabilities of parallelism in LOCO, we adopted a Petri net model⁹ of the hardware resources and instruction requirements. As we show partially in Figure 4,

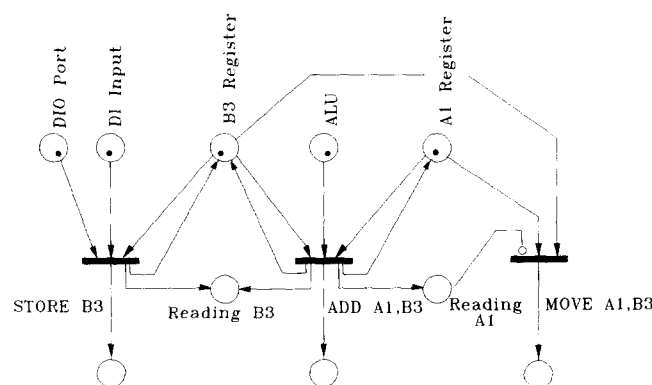


Figure 4 Extract of the Petri net model for LOCO

the hardware resources are represented by using input places, the instructions are given by transitions, and the possibility of concurrent execution is associated with the output places. Input places are loaded with one token for each available resource. If the desired instruction can be executed in parallel with the one previously considered, the corresponding transition is enabled to fire. After possible firing of such transition, the presence (or absence) of a token into the output place states the capability (or the impossibility) of concurrent execution.

The main drawback of the previous approach is the strict locality of the program scope during optimization. This leads usually to a good optimization with reasonable effort, but we do not often achieve a global optimum. A different approach we are considering and experimenting with is automatic identification of concurrency by starting from a wider view of the computational graph of the program, as discussed in Reference 7 for restructuring subsequences of sentences. In this case the instructions of the program are represented by nodes in a directed graph; arcs give the sequential dependencies between instructions. Branch nodes are introduced to represent alternative sequences that must be executed according to the result of a test. Convergence nodes represent junction points where more branches converge into a unique sequence of instructions. An example is shown in Figure 5a.

Optimization tries to remove unnecessary sequential constraint (i.e. the arcs) between instructions. This operation is performed by examining the characteristics of the single instruction. In particular, two instructions are necessarily sequential if the outputs of the first instruction are used as inputs of the second one. When an arc is removed, the instructions are concurrent. Both of them are used to label the unique node to which they become associated after compaction (see Figure 5b).

In this case, optimization is not only localized to adjacent instructions: it involves the whole sequence of instructions starting from a branch node until a convergence node. To achieve this goal, the instructions belonging to one of these sequences may be reordered to allow for removing unnecessary arcs. Since reordering may influence the final identification of concurrent instructions, we are experimenting with different techniques to generate the best ordering. At the moment we adopt a traditional branch-and-bound approach in the delivered optimizer. For long sequences, the performance of such techniques is not satisfactory, both for the computation time and the memory storage which are required; for this reason, we are

experimenting with heuristic hill-climbing algorithms¹⁰. In particular, we considered an efficient prototype tool for code optimization based upon simulated annealing. The final solution is identified by using a constrained search, which is characterized by random movements around the actual solution and by a probabilistic acceptance rule: the new solution is accepted if it is better or at least not worse than the previous one. Acceptance of hill-climbing movements is progressively reduced as iterations proceed.

In the optimized graph, nodes represent the maximum sets of concurrent instructions; arcs are the necessary constraints onto sequential execution. From such representation it is possible to generate optimized code for the multiprocessor program. Branches are reproduced as in the original program by using jump instructions. Each node gives a multiple instruction obtained by verifying the capability of concurrent execution, as in LOCO. When instructions of a node cannot be compacted into a single multiple instruction since they use the same hardware resources, a sequence of multiple instructions must be generated. Partitioning of instructions belonging to the same node into a sequence of multiple instructions may be not unique and the alternative sequences may have different length (i.e. number of multiple instructions). Also in this case we are currently using a branch-and-bound technique to identify a partitioning with the minimum length, even though we are running experiments to adopt a heuristic hill-climbing technique¹⁰.

Other approaches to code optimization could consider a complete restructuring of the whole program, e.g. involving rewriting of branches. Also, since some instructions of a multiple instruction may be considerably faster than the whole multiple instruction, some resources become available earlier than others in the multiple instruction. In this case we could start other instructions which use such resources as soon as they become available. Both of these optimizations are very complex: therefore, they are not presently considered in our research.

HIGH-LEVEL LANGUAGES FOR MULTIPROCESSOR PROGRAMMING

A higher-level programming interface may be defined by adopting a standard high-level programming language and custom libraries for multiprocessor operations. This approach makes it possible to avoid a large number of details about the hardware architecture, e.g. the number of registers, the data transfer between memory locations, the use of jump instructions. The user has a view of the system which is closer to a procedural description of the computation and which is built with more complex control structures.

In the present version of our programming environment we adopted a subset of the C standard programming language¹¹. The multiprocessor hardware is assumed to be equivalent to the standard architecture of a monoprocessor (central processing unit, memory, input/output channels). Our multiprocessor differs in fact from monoprocessors only because it presents one control unit for a number of

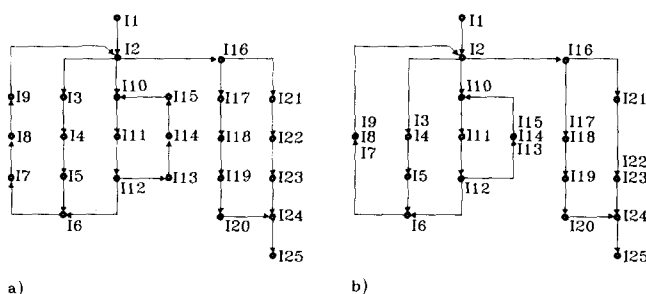


Figure 5 Computational graph: a, before and b, after optimization

processing units working in parallel, while the mono-processor has one control unit for the unique processing unit. In the multiprocessor the same operation is imposed by the control unit to all processing units at the same time.

Different types of variables are allowed on the multiprocessor C: integer, real, and char. Arrays and structures are supported for defining structured data. Pointers are available only for indirect addressing, not for dynamic memory allocation. Input and output standard files are defined to acquire input samples from the external environment and to deliver results to the front-end computer. No dynamic memory manager is presently supported. Since the processing units do not manage automatically a stack structure in memory, only global variables are allowed. Traditional control structures are defined: two-way conditional statement (if-then-else), multiple-choice statement (switch), conditional loops (while-do, for-do), procedure call. Since no stack is present on the controller unit, a calling mechanism is supported only for one level of depth. A number of improvements and elimination of some constraints of the present implementation will be available with the second release of the controller unit and the related software environment: in particular, procedure and function calls will be completely supported by using the on-board stack and call facilities.

To write the C source code of the multiprocessor program, the user can use any traditional text editor available on the front-end computer, as for the assembler programs. The output of such editor is a usual text file. No specialized support is provided for this phase of software development.

Fast development of new measurement algorithms needs adequate support from a run-time library of mathematical functions and digital signal processing algorithms. The user can include references to such library as if it were a standard C library. He therefore views the multiprocessor system as a traditional monoprocessor computer; during execution, the run-time library guarantees the proper data transfer and program execution on the multiprocessor architecture.

Translation of the user program for the multiprocessor is obtained by means of the C cross-compiler we developed on the front-end computer⁶. We designed a cross-compiler on the front-end computer since in Unix there are a number of tools for fast software development that are not available on our multiprocessor. As a usual compiler, our cross-compiler scans the C source code to verify the correctness of the statements from the point of view of the C grammar (with our restrictions summarized above). Then, it generates the executable image of the program as the assembling program. Such an operation is a two step translation: in the first step the cross-compiler creates the assembler translation of the C program in a temporary file, while in the second step it runs the kernel of the assembling program to produce the executable image. If the user needs the assembler translation of the executable image, a compiler directive forces the generation both of the executable image and the source code in the assembler language as the translation of the C program.

Extensions of loader and monitor facilities for the assem-

bler programs are also provided to perform these tasks from the view of high-level programming. Work is in progress for a high-level debugging tool for the C language. No support is presently given to separate compilation: the cross-compiler is actually a compiler-linker which generates the executable image in subsequent steps including the linking phase.

Since the FORTRAN language is still widely-known and used for numerical computation, we are developing a FORTRAN version of the programming tools previously presented for the C language. The functionalities of such tools will be identical to those here discussed.

PROGRAMMING WITH GRAPHICS

Computer programming and, above all, parallel programming may be an obstacle to the diffusion of advanced architecture for measurement, in particular when users have no deep knowledge of programming techniques or they do not want to become experts on computers. To avoid constraints on the efficacy of our multiprocessor and effective adoption in real applications with users who are not experienced in computer programming, we introduce graphic programming¹² as an advanced programming approach for our environment.

This new technique is based upon a particular view of the computation which comes from functional programming. We consider the computation performed in the multiprocessor as a mathematical function that we apply to the input data to produce the final results and to extract some characteristic figures of the input set. A similar view may be found in the electrical sciences with the network functions, or in automatic controls with transfer functions.

In graphic programming the definition of the computation consists essentially of describing the computation by means of the mathematical functions that must be applied to data. The user can create his own transfer function of the whole computation by:

- selecting the basic mathematical operations from a predefined list, and
- connecting the basic operations defining the data flows.

In other words, from the graphic point of view, the user must select a set of basic building blocks to which the basic operations are associated. Building blocks may be as simple as addition, or very complex such as Fourier transform. Then, the user must define the computational graph associated to the desired algorithm, i.e. he must connect the building blocks to describe where the inputs of the multiprocessor must be carried, where the output of each block must be delivered and which data constitute the final output of the computation. In *Figure 6*, an example of graphic programming is shown for the convolution algorithm; multipliers, adders and registers (variables) are the basic building blocks, while interconnections define the evaluation flow of the weighted input summation and the data propagation.

These activities are directly supported in our environment by the Graphic Programming Editor GPE, a dedicated

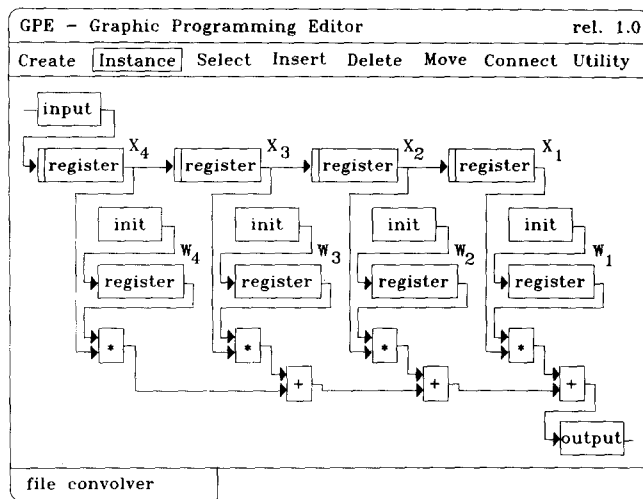


Figure 6 Example of algorithm described by graphic programming: convolution

graphic environment for multiprocessor programming. We believe that a graphic description of the new transfer function is more simple, effective, fast and easy to be created than similar descriptions produced with traditional approaches. The user can in fact concentrate his attention on the design problems, not on syntactical problems related to textual description. On the other hand, a guided choice of commands through pull-down menus and icons avoids a large number of typing errors, improving design speed and correctness.

The graphic editor makes it possible to create a graphical description of the computation by selecting the basic functional blocks in a given list, as shown in Figure 7. Blocks are represented by icons with the symbolic description of the corresponding function. To simplify interaction, pointing devices (mouse or tablet) are supported both for command selection and option definitions. Once selected, one block can be placed anywhere on the screen (i.e. in the description of the transfer function). Then, the user must define the interconnections between blocks to create the correct data flow. Only when all connections have been given, can the design of the new program be considered completed.

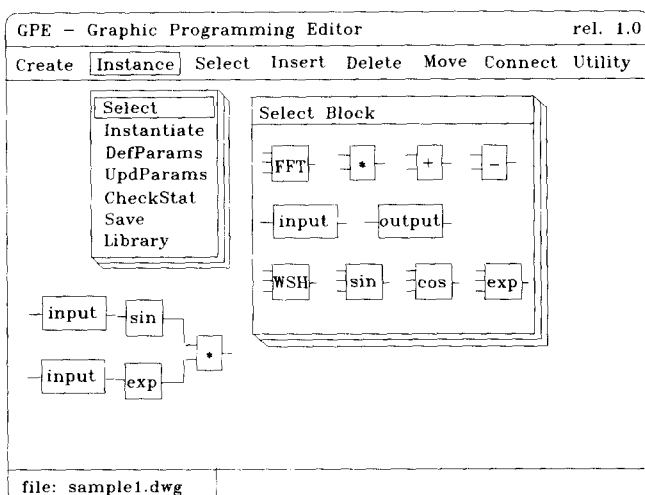


Figure 7 Graphic programming environment

The graphic editor provides the following functions:

- File manipulation. These commands make it possible to load, store, print, copy and delete files containing programs which have been written using the graphic editor itself.
- Block selection. This subsystem provides the user-friendly interface to block selection. A list of building blocks is presented and the user is prompted to choose one of them: to simplify selection, icons may be used to show the transfer functions instead of their symbolic name. The basic library supports arithmetic operations, logic operations, Fourier and Walsh transforms, convolution, decimation, evaluation of basic statistic estimators (mean, variance, linear regression). Possible parameters may be defined to customize the generic block. After selection, the user must place the block onto the drawing plane.
- Block placement. Commands are available to select move, copy and delete blocks which lie on the drawing plane.
- Interconnection. Building blocks, which have been already placed onto the drawing plane, must be connected to define the computational graph of the algorithm that will be executed by the multiprocessor architecture. By selecting the interconnection function, the user can describe the data transfer procedure during the computation. In particular, he can define the destination of each output computed by a building block, the primary inputs of the computation and the final outputs. Multiple interconnections may be defined to broadcast the output of one source block to more destination blocks. Compatibility checking is performed upon the connected data.
- Interconnection maintenance. Commands are available to delete the modify interconnections.
- Block definition and library manager. The user can improve the available library of building blocks by defining new blocks. To such purpose, the user must draw a transfer function as it was the complete computation that should be performed by the multiprocessor: then, he must save this function as a new block. The description of a user-defined block is completely equivalent to the description of a pre-defined block.

Commands are available to view, modify, copy and delete user-defined building blocks. The internal data structure of the graphic editor has been particularly tailored to achieve the best performance during user activities. Data associated with each block are collected in a block object and dynamic lists are automatically updated to guarantee the optimum efficiency in data retrieval during design. For example, to such purpose, we introduced a virtual grid on the editor screen: at each crossing of the grid we associate a list of the objects which lay in the surrounding of the crossing itself. At each moment it is easy and fast to identify the position of objects, or to perform graphic operations on them (insertion, deletion, move, mirroring, rotation, copying).

When a program for the multiprocessor has been completely described by the user through the graphic

editor, it can be definitely saved in a file. This description is not suited for direct execution on the multiprocessor architecture, but it must be translated into an executable image, as for C or assembler programs. This operation is performed by the graphic translator in a two step procedure. At first, it scans the graphic description of the computation for generating an internal representation of the computation. During this phase, it associates the assembler constructs and macro procedures with the building blocks used and with interconnections. Then, it activates the kernel of the assembling program to generate the executable image. From the graphic translator it is also possible to obtain the assembler code corresponding to the graphic description.

Loading and monitoring onto the multiprocessor architecture may be achieved by using an extension of the corresponding procedures for the lower programming levels. Unfortunately, no debugging tool is presently available to verify the correctness of the program behaviour by operating on the graphic description.

Experiments have shown that graphic programming gives much freedom to the measurement experts and opens new perspectives since it makes it possible to define new powerful instruments to measure the different quantities and characteristics, without the constraints of knowledge of computer programming. Moreover, we can also easily create virtual instruments which could never be implemented efficiently by using traditional analogue or analogue-digital approaches.

INTERACTIVE ENVIRONMENT

The complexity of the software environment and the number of operations that must be executed to generate and execute a program on the multiprocessor architecture may not be easily acceptable and attractive for users who are not expert programmers. Often the graphic programming style is also not enough to overcome user resistance and hostility against a complex working environment. To completely remove these feelings that reduce the productivity and efficiency of the users, we designed and implemented an integrated environment. The tools and interactions in such an environment are summarized in Figure 8.

All activities on the multiprocessor architecture (program design, compilation, execution, monitoring and control) may be executed in the IEN interactive environment. This tool provides a unique, user-friendly interface for system management. It is essentially a shell for a simple and homogeneous guided interaction with the environment for multiprocessor programming which runs on the Unix front-end computer.

The user is aided in his work by a graphic interface based upon context sensitive pull-down menus, icons and multiple windows; as soon as a selection is entered through the keyboard or mouse, currently available alternatives and options are presented for subsequent operations. By selecting the proper functions, the user can activate any of the tools and operations described in the previous sections, possibly one for each of the windows currently opened. An example is given in Figure 9.

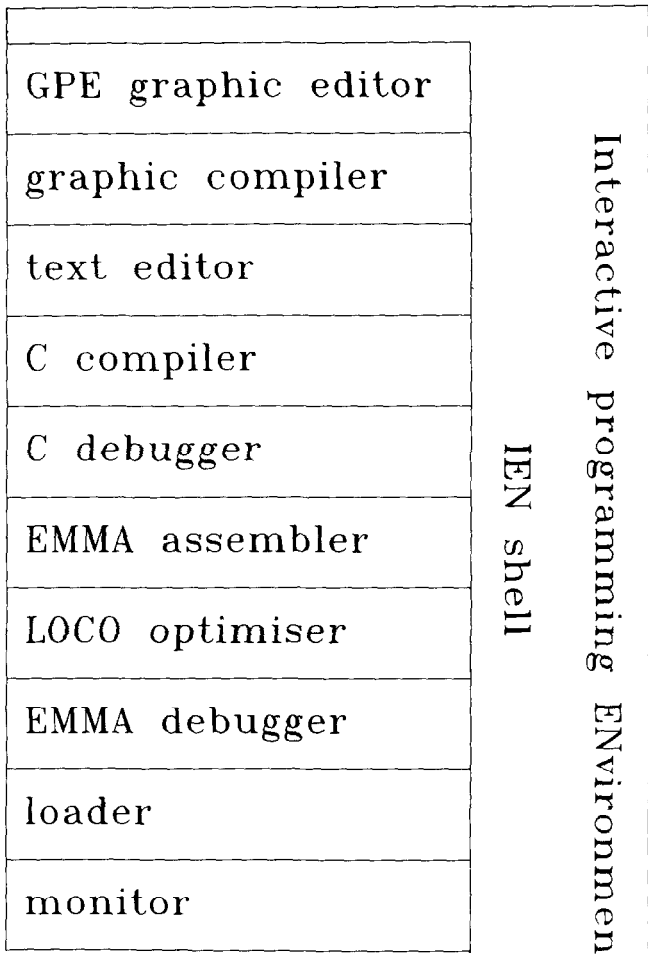


Figure 8 Integrated interactive environment

The integrated environment also provides automatic synchronization and coordination between the sequences of operations that must be performed on the multiprocessor, both during the development of a new application and the on-field execution of application programs.

IEN provides the following functions:

- Selection of the programming style (assembler, C, graphics). This makes it possible to configure auto-

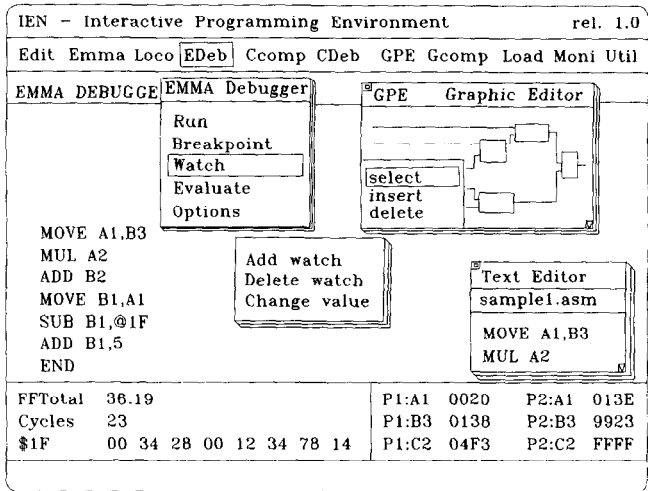


Figure 9 Working session in IEN

matically all tools of the integrated environment to support the preferred programming view directly: no further selection of the abstraction view will be required during the subsequent steps.

- Activation of the editor for designing the application program. It starts the preferred editor, according to the currently selected programming style. If graphics has been adopted, the graphic editor is activated; otherwise a text editor (preconfigured by the user) is used.
- Activation of the cross-compiler. The executable image of the program is generated starting from the source description. The translator used by the system is chosen according to the current selection of the programming style and the type of the considered source code. The final output is automatically linked to the run-time libraries.
- Program loading from the front-end computer onto the multiprocessor subsystem. This tool allows selection of the executable image that will be executed by the multiprocessor architecture: the program is automatically transferred from the file system of the front-end computer into the program memory of the controller unit.
- Program monitor. This starts and controls the execution of a program which has been loaded into the control unit. Continuous execution, step-by-step execution and conditional execution are available to support program development and on-field execution.
- Program debugger. This is designed to allow interactive verification of program correctness: it uses the control features of the monitor facility to interact with the multiprocessor system. On-line execution may be observed and controlled by modifying the actual values of data stored in the multiprocessor system. At the moment, it is available only for the assembler view.
- File and directory management. These commands provide the usual facilities to move through the directories tree in the Unix environment, to create directories, to copy and remove files, to print and show files and directories.

In IEN the user is fully guided through the sequences of operations required to design, implement and execute

application programs. For example, when the user starts the development of a new program, the system proposes the use of one of the editors for program definition, according to the user's expertise. Then, when the program has been written, the graphic translator, the cross-compiler or the assembling program is suggested to generate the executable image. Finally, loading and monitoring functions (and possibly debugging) become available to verify the program behaviour by direct execution and result observation. All these features are achieved by running and controlling the proper system tasks, which are spawned as concurrent processes by IEN.

MULTIPROCESSOR APPLICATIONS

Massive computation on large input data is often required to extract the basic characteristics in electronic measurements, in particular for real-time control of complex systems¹³⁻¹⁷. Our multiprocessor architecture is an effective solution for a large class of industrial applications, since it can be implemented at reasonable cost by using standard components. The research which suggested and supported the development of this architecture is one of the most recent and important applications in the field of electrical engineering: the real-time accurate control of high-voltage (HV) transmission and distribution systems.

The basic goal of this application regards the optimization of electrical power distribution with load balancing. For such purposes, it is necessary to implement a real-time measure of the electrical quantities of the distribution system and compensate the harmonic contents of current and voltages due to the characteristics of the distribution system and to the electric loads. Real-time control is important to provide continuously updated data that close the feedback loop onto the harmonic compensator to suppress unexpected frequencies and to correct active and reactive power.

The classical arrangement of a power node of an electrical distribution network is shown in Figure 10. The HV power station contains the following components:

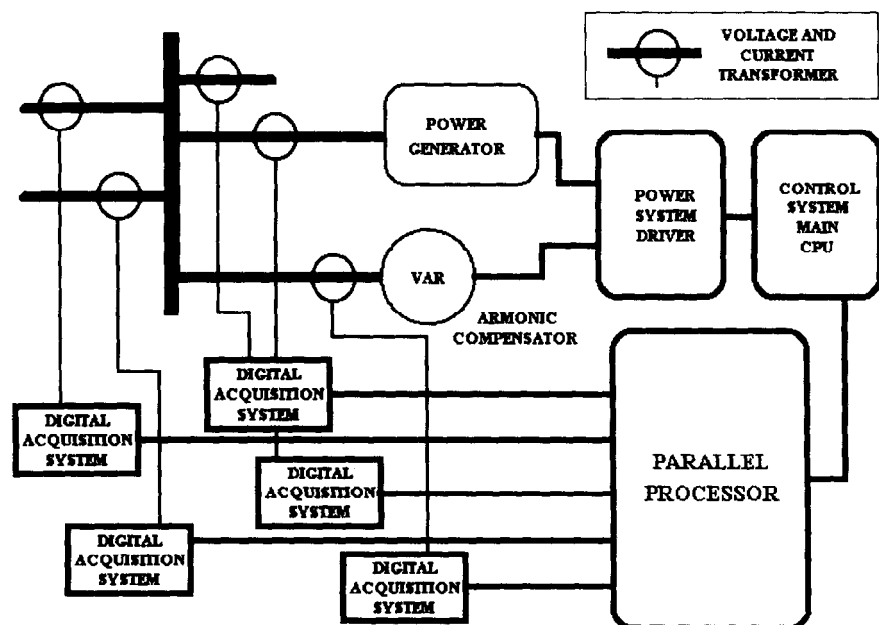


Figure 10 Typical distribution node for electrical power

- a central set of station bars, which provide electrical connection between the power sources and the destination lines for power distribution,
- the power plant which generates the electrical power for the distribution network,
- the HV measuring system, which provides automatic measurement of the instantaneous electrical parameters (voltages and currents) in the station bars,
- the harmonic compensator, which adjusts the harmonic contents of electrical parameters,
- the station control subsystem, which extracts the characteristic figures from the sampled data and imposes the relevant signals on the power plant and on the harmonic compensator to optimize the power distribution.

The distribution lines, connected to the HV station, are continuously monitored by measuring the six basic electrical quantities of a three-phase system for power distribution: the three line currents (one for each phase) flowing into the station bars, and the three line voltages across the bars themselves.

From these measurements we extract the spectrum contents of frequency components, to verify the presence of undesired frequencies and to optimize power distribution. Due to the structure of the power systems, only the amplitudes of the lower harmonics need be considered. Therefore, information required by the control algorithms is highly reduced with respect to the amount of sampled data. For example, if we consider the 30 most significant components of the spectrum, the characteristic figures that must be extracted from each of the six sampled quantities (i.e. currents and voltages) are 30, while the total input figures required by system control are 180.

On the reduced data, the control subsystem is able to evaluate the modifications that must be imposed on the working conditions of the harmonic compensator and the power plant to achieve the best performance. On-line adjusting of the working points is immediately actuated by the central dispatching centre (CDC). Feedback on the drivers of the power plants may be realized by using both analogue and digital approaches. The first techniques are well-known and usually applied in present standard power systems. The second solution represents the new era of electronic control systems: it assures a more flexible, efficient control procedure and can improve the accuracy of feedback operations under heavy perturbations in dynamic transients. Intensive computation, exploitation of parallelism, overlapping of computation and data transfer, and massive-input throughput are the main characteristics of our architecture which make it possible to guarantee a very low response time in driving the electric power system for harmonic compensation and, thus, to achieve an effective and optimal power distribution. Data compression is very useful to reduce drastically the amount of input information that must be considered to perform the control activities: only the integral form of the inputs and the Fourier transform are required for average and spectral analysis. Control algorithms are then executed on compressed data by a traditional computer structure to generate the control signals.

The above application is the first case of a dedicated

parallel processor for power network control that is presented in the literature. Parallel processing is extensively used in power systems to study load-flow and system dynamics; sometimes, off-line measurements also require parallel systems. Dedicated real-time processing and control were generally implemented by using lower-level hardware resources and limiting control parameters and network free-dynamics. Performance evaluations of our system suggest that our approach optimizes costs and benefits in network sites in which the installed power is greater than 50 MW. Some new DSP-based processing systems built and commercialized in the last two years are promising solutions that could be more convenient in the near future.

A second class of applications we experimented with is in advanced robotics. Interaction of robots with the external environment requires advanced sensing functions, able to condense much elementary information into high-level parameters. Visual and tactile recognition belongs to this class of sensing functions, where a large number of very simple sensing elements (video or tactile pixels) provide elementary signals which, only after specific mathematical transformations, can be summarized in highly-condensed information. Only dedicated computational structures are able to perform such signal processing activities in hard real time. The parallel processor presented in this paper has been specifically studied also for several robotic applications in the field of tactile sensor signal processing.

Our specific robotics application deals with a pair of tactile matrices (4×4 taxels each), mounted on two opposite fingers of a dextrous manipulator. Every matrix provides 16 different signals sampled at the frequency f_c . The processor receives 16 data blocks: each of them contains 1 kbyte of samples and represents the time behaviour of a signal over the period T . Data blocks are geometrically correlated to each other by their position in the finger structure. Signals are processed by taking into account their mixed time-space correlation. Primitive features characterizing the evolution of the grasp are extracted and sent to the high-level microprocessor to provide the necessary feedback to the robotic hand and finger drives.

This arrangement differs from the classical solutions which adopt dedicated signal processing of the sensor signals in the processing unit both for pre-processing and feature extraction. In our solution, these two operations are separated onto the parallel processor and the front-end in order to enhance the computational time and the global performance. Figure 11 shows the structure of a two-processor tactile system. Two dextrous fingers (equipped with a 4×4 matrix of PVDF sensing elements) work in parallel to define the grasp condition and provide the feedback to finger drives. Every channel of the processor receives sampled data from an analogue tree followed by a single-root digital subsystem. The reduction ratio (i.e. the number of high-level parameters computed every thousand input data) is around 0.2–0.5. The main CPU must evaluate only a small set of composed data to define the final result concerning the grasp. Compared with other solutions, our approach reduces global processing time by 30–40%. Hardware costs are higher but system flexibility is greatly increased.

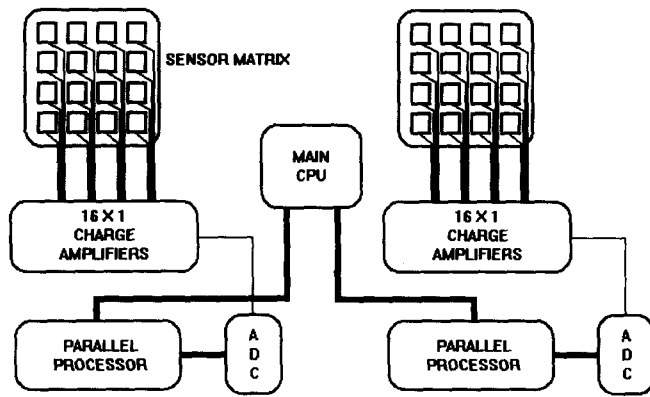


Figure 11 Multiprocessor tactile system

CONCLUDING REMARKS

Design and implementation of advanced architectures for massive computation cannot avoid the problem of the definition of a flexible and friendly environment for user programming. This paper presented a multiprocessor architecture for measurement applications and digital signal processing: particular importance in our system design was given to the identification of a user-oriented interface.

Three levels of the software environment were introduced and described in detail, since the hierarchical interface constitutes one of the most advanced approaches in electronic measurement integrated systems. In particular, the graphic programming approach is attractive since it allows easy interaction even to people not accustomed to advanced programming techniques and to multiprocessor architectures.

Further research activities will be performed to improve the user interface and the programming tools available for our multiprocessor. At this time, we are working in different directions. To simplify program correctness checking, we are designing an advanced interactive tool for multiprocessor debugging by using a symbolic representation of data and procedures. In parallel, we are extending all tools to the FORTRAN language. Moreover, we are studying the design of a debugger for the approach based upon graphic programming.

We are also expanding the libraries of mathematical procedures to provide a large predefined set of complex operations for digital signal processing. As far as algorithm optimization is concerned, we are considering the possibility of implementing a user-guided tool for automatic identification of concurrent operations at the highest programming levels. Such a tool should generate the optimum sequences of concurrent operations which perform the nominal algorithm defined by the user; optimization should consider not only spatial interactions of data and control paths, but also the time overlapping and shifting of operations.

To increase the capabilities of our architecture, we are also redesigning the controller unit to introduce a micro-programmed structure with dynamic microprogramming configurability. This technique will allow the use of the same basic multiprocessor architecture for different applications by defining specialized microprogrammed instruc-

tions to support dedicated operations. Further ideas and suggestions will be acquired by experimenting extensively the practical use of this architecture in real-time applications, such as processing of sensor data in advanced robotics and network control in power systems.

REFERENCES

- 1 Oppenheim, A V and Schaffer, R W *Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ (1975)
- 2 Gandelli, A and Piuri, V 'A flexible multiprocessor architecture for high-level computation in electronic measurements', *Proc. IMACS '91*, Dublin, May 1991
- 3 Hockney, R W and Jesshope, C R *Parallel Computers*, Adam Hilger, Bristol (1981)
- 4 Hwang, K and Briggs, F A *Computer Architecture and Parallel Processing*, McGraw Hill, New York (1985)
- 5 Gandelli, A and Piuri, V 'EMMA—Electronic-Measurement Multiprocessor Assembler: definition and translator characteristics' *Int. Rep. No. 025-90*, Department of Electronics, Politecnico di Milano, 1990
- 6 Aho, A V and Ullman, J D *Principles of Compiler Design*, Addison-Wesley, Reading, MA (1977)
- 7 Wolfe, M *Optimizing Supercompilers for Supercomputers*, The MIT Press, Cambridge, MA (1989)
- 8 Zima, H and Chapman, B *Supercompilers for Parallel and Vector Computers*, ACM Press, Addison-Wesley, Reading, MA (1990)
- 9 Peterson, J L *Petri Net Theory and the Modelling of Systems*, Prentice-Hall, Englewood Cliffs, NJ (1981)
- 10 Nahar, S, Sahani, S and Shragowitz, E 'Simulated annealing and combinatorial optimisation' *Proc. 23rd Design Automation Conference* (1986)
- 11 Kernighan, B W and Ritchie, D M *The C Programming Language*, Prentice Hall, Englewood Cliffs, NJ (1981)
- 12 Preece, J, Rogers, Y, Sharp, H, Benyon, D, Holland, S and Carey, T *Human-Computer Interaction*, Addison-Wesley, Reading, MA (1994)
- 13 Bishops, M T, Gilker, C and Mendis, S R 'PC-controlled hand-held meter measures power system harmonics' *IEEE Comput. Appl. Power* Vol 4 No 4 (October 1991)
- 14 Gilker, C and Flinn, D G 'A measurement technique for power system harmonics' *IEEE Comput. Appl. Power* Vol 3 No 4 (October 1990)
- 15 Kennedy, R A and Rickey, D N 'Monitoring and control of industrial power systems' *IEEE Comput. Appl. Power* Vol 2 No 4 (October 1989)
- 16 Van der Auweraer, H and Snoeys, R 'FFT implementation alternatives in advanced measurement systems' *IEEE Micro* Vol 7 No 1 (February 1987)
- 17 Appiani, E, Contorno, B, Luperini, V and Roncarolo, L 'EMMA2, a high-performance hierarchical multiprocessor' *IEEE Micro* Vol 9 No 1 (February 1989)



Alessandro Gandelli received a Dr. Eng. degree in electrical engineering in 1985 and a PhD in electrical engineering in 1988, at Politecnico di Milano, Italy. Presently he is Assistant Professor in Electronic Measurements at the Department of Electrical Engineering, Politecnico di Milano. His research interests include hybrid microelectronics for advanced sensor technology, signal processing, A/D conversion theory, and their applications in instrumentation accuracy evaluation and sensor data processing.



Vincenzo Piuri obtained a Dr. Eng. degree in electronic engineering in 1984 and a PhD in computer engineering in 1989, at Politecnico di Milano, Italy. He was Assistant Professor in Electrical Engineering and in Computer Programming at Politecnico di Milano; he is presently Associate Professor in Operating Systems at Politecnico di Milano. His research interests include distributed and parallel computing systems, computer arithmetic, neural networks, fault tolerance, CAD for VLSI and digital signal processing.