A *Move* Processor for Bio-Inspired Systems

Gianluca Tempesti Pierre-André Mudry
Ralph Hoffmann
School of Computer and Communication Sciences
Ecole Polytechnique Fédérale de Lausanne (EPFL)
EPFL-IC-ISIM-LSL, INN Ecublens, Station 14, CH-1015 Lausanne, Switzerland
Email: Gianluca.Tempesti@epfl.ch

Abstract

The structure and operation of multi-cellular organisms relies, among other things, on the specialization of the cells' physical structure to a finite set of specific operations. If we wish to make the analogy between a biological cell and a digital processor, we should note that nature's approach to parallel processing is subtly different from conventional von Neumann architectures or even from conventional parallel processing approaches, where specialization is obtained by adapting software to a fixed hardware structure.

In this article we will present the outline of a novel processor architecture based on the Move *or TTA (Transport-Triggered Architecture) approach. The features of such architectures allow them to implement systems that more closely resemble, within the limitations imposed by the capabilities of conventional silicon, the general* modus operandi *of multi-cellular organisms.*

1. Introduction

One of the main motivations for the development of hardware-based bio-inspired systems is the astounding level of complexity achieved by biological organisms, a complexity far beyond that of even the latest silicon-based circuit. The promise of next-generation technologies [4][6][8] lies in their ability to work at the same molecular level, with comparable component densities, as biological systems.

Among the many questions open for these technologies is how to exploit this immense wealth of hardware. The study of how biology, and notably multi-cellular organisms, have successfully solved this issue is a possible avenue for finding approaches that could potentially be applied to these circuits.

Of particular interest in this context is the biological process of *ontogenesis*, whereby molecules self-assemble into cells and cells self-assemble into complete organisms, according to a (very compact) set of instructions contained in the genome. A possible analogy between biological systems and electronics is to compare a cell to a digital processor, implying a correspondence between an organism and a massively parallel multi-processor system. This analogy holds in several respects, but it should be noted that nature's approach to parallel processing is subtly different from conventional von Neumann architectures or even from conventional parallel processing approaches, where specialization is obtained by adapting software to a fixed hardware structure.

Current technology is just barely able to provide an acceptable solution to this requirement: by using reconfigurable logic circuits (FPGAs), we have shown [10] that it is possible overcome the rigidity of silicon and adapt (at a price) the *functional* structure of our systems to a given application. However, the realization of application-specific processor architectures that can efficiently exploit the adaptive features of this approach remains an open problem.

Finding an efficient solution to this issue is a fundamental step in the development of a complete environment for the design of our bio-inspired system. This environment, at the core of our new project, will allow us to integrate in our approach the other axes of bio-inspiration (learning and evolution) in a context which bears a relatively close resemblance to the biological context in which they occur.

This article describes the first results of a new project that, building on the bases provided by the Embryonics [10] and POEtic [25] projects, will define a processor architecture specifically conceived for the realization of this kind of bio-inspired systems. In this paper, we will try to identify some of the requirements of an ontogenetic processor architecture and present the outline of a novel architecture that represents an effort towards the designs of systems that more closely resemble, within the limitations imposed by the capabilities of conventional silicon, the general *modus operandi* of multi-cellular organisms.

2. Background

Many different approaches can be used to draw inspiration from nature in the design of electronic systems. Even within the much more restricted area of ontogenetic hardware (that is, hardware inspired by the development of multi-cellular organisms), several valid approaches have been studied (for a partial review of such systems, see [21]).

Within the Embryonics project [10], aimed at transposing into silicon some of the mechanisms and properties of involved in the development of complex organisms, we have been studying the application of biological ontogenesis to the design of digital hardware for several years. Among what we feel are our main contributions to the field is a self-contained representation of a mapping between the world of multi-cellular organisms in biology and the world of digital hardware systems (Fig. 1), based on 4 levels of complexity, ranging from the population of organisms to the molecule.

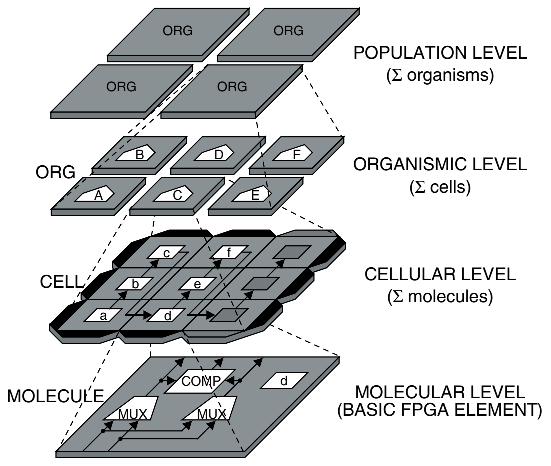

Figure 1. The four hierarchical levels of complexity of the Embryonics project

Within this mapping, we define an artificial organism as a parallel array of cells, where each cell is a simple processor that contains the description of the operation of every other cell in the organism in the form of a program (the *genome*). The redundancy inherent in this approach is compensated by the added capabilities of the system, such as growth [11] and self-repair [22].

The operation of multi-cellular organisms relies, among other things, on the specialization of the cells to a finite set of specific operations, implying that the cells' *physical structure* is adapted to its function (e.g., a skin cell is physically different from a liver cell). Structural differences notwithstanding, the same program (genome) controls the operation of all cells. To maintain the analogy with digital processors, we must achieve a similar degree of adaptation.

A first answer to this issue was to redefine our cells as *reconfigurable* processing elements, realized by programmable logic circuits and structurally adapted to the application to be implemented. For a given application, all cells are structurally identical and contain the same program (and can thus be seen as *stem cells* [13]), but different parts of the program and of the structure are activated depending on the cell's position, implementing specialization.

In 2001, we launched, with the universities of York, Barcelona (UPC), Lausanne, and Glasgow, a project called "Reconfigurable POEtic Tissue" [25] funded by the European Community. This project aims at defining a novel programmable circuit specifically designed for the implementation of systems inspired by all three axes of bio-inspiration in digital hardware [17] (Phylogenesis or evolution, Ontogenesis or growth, Epigenesis or learning) and thus at providing an efficient molecular level for our systems.

Among the contributions of the POEtic project, two are particularly relevant for the subject of this article. The first is a definition of the general structure of a processor for bio-inspired systems (Fig. 2). Such a processor can be seen as a three-layer structure, where each layer is dedicated to the implementation of one of the axes of bio-inspiration. The bottom layer, or *genotype layer*, stores the genetic information of the cell (the genome). In this layer, the genetic operations associated with evolutionary approaches can be easily implemented, with the aid of an on-chip microcontroller. The middle layer, or *mapping layer*, implements developmental algorithms to realize processes analogous to ontogenetic growth. The top layer, or *phenotype layer*, is used for the actual execution of the application. In the case of a POE system, the applications to be executed are based essentially on neural networks, but in theory the architecture is versatile enough to be used for applications that are not directly bio-inspired (in fact, the structure can be adapted to implement any combination of the POE axes).

The second contribution is more directly technological: by implementing in the circuit a dynamic routing network [23], it becomes unnecessary to explicitly define the connections between cells. Communication channels are set up dynamically at runtime using an address-based mechanism: a channel can be created (or destroyed) during the operation of the circuit by setting an address register to some value (stored in memory or computed by the cell) and launching the routing process. This (relatively) simple mechanism has major consequences for the implementation of ontogenetic processes, since it allows cells to be created and connected to the rest of the network (or destroyed and removed from the network) at any time during the circuit's operation and anywhere within the circuit's surface. The impact of these features for growth and self-repair should be obvious and, in defining our ontogenetic architectures, we shall assume the availability of a dynamic routing mechanism.

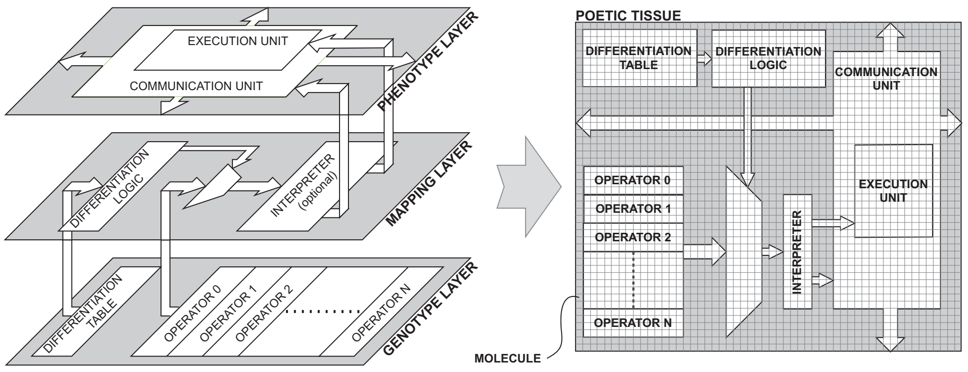

Figure 2. POEtic systems rely on a three-layer structure, which is then mapped onto the reconfigurable logic of the POEtic tissue.

3. Ontogenetic Processor Architectures

Exploiting the results of the projects mentioned above, we have begun to address some of the issues related to the implementation of the *cellular* level of our systems by defining some of the practical requirements of ontogenetic applications.

We found that, in fact, it is not simple to identify applications that can exploit the features of an ontogenetic approach on conventional silicon. A developmental process is likely to become a necessity for the next generation of molecular-scale circuits (and indeed many of the issues we address concern this kind of circuits), but today's technology remains at a level of complexity that can be handled by more conventional design approaches. However, there exist some families of applications where ontogenesis can be useful today (see [21] for a more complete review).

A first set of applications can exploit structural adaptation to respond to environmental stimuli that cannot be foreseen at design time. Typically, these applications, which *self-organize around external stimuli*, correspond to other kinds of bio-inspired approaches, such as neural networks [26] or robotics [1], but the approach can be extended to applications where the circuit's function is determined by the user at runtime (e.g., custom graphic or sound filters [18]).

A noteworthy special case in this context are systems that exploit developmental processes for their capability to represent structural information in a compact form. This compactness is a major advantage when applying evolutionary approaches to hardware design. In this case, the fitness of the individuals can be seen as the external input around which the system is structured and the final individual cannot, by definition, be determined at design time.

A second, more general set of applications (not sufficiently exploited in the context of bio-inspired systems) could use development for the creation of massively parallel arrays of reconfigurable processors. The (relative) decline of massively multi-processor systems is usually explained by the difficulty of exploiting the parallelism inherent in many algorithms. In turn, it could be argued that at least part of this complexity lies in the *implementation* of these algorithms, usually written in a high-level language with a general-purpose instruction set.

A well-known technique to simplify the realization of algorithms on a massively parallel system is the use of *application-specific processors*: if the processing elements in the system are designed to execute a single application (or set of applications), the instruction set of the processor can be targeted to the required operations, leading to programs that are much simpler than those written for general-purpose processors. This approach can simplify the task of programming parallel systems by moving some of the software's complexity to the hardware.

The kind of ontogenetic systems we have been working on are ideally suited for this kind of approach: not only the processing elements are fully configurable (and can thus be made application-specific), but our developmental mechanisms allow the inter-processor communication network to adapt itself at runtime, letting the system *self-organize around the data flow*.

As an important special case of this kind of systems, we shall mention *fault-tolerant* processor arrays. The possibility of operating in the presence of hardware faults is not only a key feature of molecular-level computing [7], but also increasingly important for silicon-based circuits (error rates increase with the shrink in transistor size). In the Embryonics project we have been concentrating on a specific approach to fault tolerance: *on-line self-test and self-repair*. In this approach, the system must be able not only to detect that a fault has occurred in the hardware substrate, but also to self-repair (through reconfiguration) and to resume operation without losing its current state of operation.

This kind of fault tolerance is normally considered prohibitively expensive for commercial purposes because of its inherent overhead. However, ontogenetic systems are ideally suited for this kind of approach, as many of the mechanisms involved in the reconfiguration of the system following the detection of a fault are very similar to those required for the growth of a system. This property is an immediate consequence of the biological inspiration of our systems: in nature, self-repair (e.g., cicatrization) relies on the creation of new cells to replace those damaged by an illness or a wound, and the cellular division involved in this process is very similar to that used during the growth of the organism.

4. A MOVE Architecture for Bio-Inspired Systems

The requirements of our bio-inspired approach imply then an architecture that is substantially different from conventional general-purpose processor architectures: it must be possible to *adapt the structure of the processors to the application* to exploit the programmability of application-specific systems and it must be possible to *adapt the topology of the system to the application* to take advantage of the features of the ontogenetic approach.

To achieve this kind of adaptation within an array of processors, we exploited a relatively little-known approach, known as the *Move* or TTA (Transport-Triggered Architecture) paradigm [2][3][20], originally developed for the design of application-specific *dataflow* processors (i.e., processors where the instructions define the flow of data, rather than the operation to be executed).

4.1. The TTA Paradigm

In many respects, the overall structure of a TTA-based system is fairly conventional (which is an advantage as far as system design is concerned): data and instructions can be fetched to the processor from main memory using standard mechanisms (caches, memory management units, etc.) and are decoded within the processor much more simply than in conventional processors. The basic differences lay in the architecture of the processor itself, and hence in the instruction set.

Rather than being structured, as is usual, around a more or less serial pipeline, a *Move* processor (Fig. 3) relies on a set of *functional units* (FUs) connected together by a *transport layer*. All computation is carried out by the functional units (examples of such units can be adders, multipliers, register files, etc.) and the role of the instructions is simply to move data to and from the FUs in the order required to implement the desired operations. Since all the functional units are uniformly accessed through input and output registers, only one instruction is needed: `move`.

TTA `move` instructions trigger operations which, in the simplest case, correspond to normal RISC instructions. So, for example, to add two numbers the processor would use a functional unit that implements the `add` operation, move one operand into the first input register of the unit, move the other operand into the second input, and move the result from the output register to the unit that needs it.

The *Move* approach, in and of itself, does not imply high performance: a simple addition, in our example, requires three move instructions. Its strength lies in its *modularity*: the architecture handles the functional units as ”black boxes”, without any inherent knowledge of their functionality. This property implies that the internal architecture of the processor can be described as a *memory map* which associates the different possible operations with the address of the corresponding functional units.

This seemingly anodyne aspect of the architecture hides in reality its most powerful feature, allowing the structure of the processors to be adapted to the application by specializing the instruction set (i.e., the functional units) to the application without changing the overall structure of the processor (decode unit, busses, etc.) and the syntax of the assembly language (based on the single instruction `move`) .

In essence, not only this kind of processor can be structurally adapted to the task it has to execute, but this adaptation can be implemented with relatively little effort. Perhaps even more importantly, it becomes possible to integrate the structure of the processor into an automatic design flow: as the elements of the processor surrounding the functional units remain unchanged, these units can, in principle, be selected for a specific application (for example, from a dedicated library) and a custom processor can be automatically constructed to meet the application’s demands. An important part of the work presented in this article was aimed at designing an implementation of the TTA architecture capable of supporting this kind of automated assembly.

Moreover, the communication units CU, used to set up connections between processors in an array, can be handled exactly in the same way as a functional unit (Fig. 3): the address of the target cell can be moved to a dedicated input register in a communication unit (CU), and the data to a second input register. The unit can then autonomously set up the communication channel and transmit (or receive) the data. This key feature of the TTA approach implies that the connection network can be arbitrarily complex, as it is handled by the CUs without directly affecting the structure of the processor itself, and opens the way to the use of complex routing algorithms (e.g., dynamic routing networks [23]) that allow us to adapt the structure of the array to the application. As we will see, the prototype described in this article does not exploit this feature, but work is ongoing to implement more complex connection schemes such as those needed, for example, by artificial neural networks.

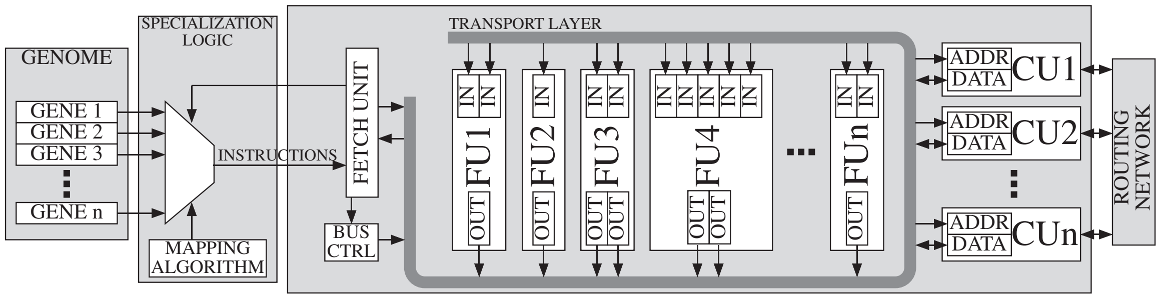

Figure 3. A MOVE processor consists of a set of functional and communication units tied together by one or more data busses.

4.2. Biological Inspiration in a MOVE Processor

In the past years, we have consistently upheld the view that the computing structure that most closely resembles a biological cell is an application-specific processor. Several considerations contributed to this argument, including the possible analogy between the genome and a computer program (where even the more complex programming abstractions such as conditional jumps seem to have a parallel in the biological world) and the versatility inherent in programmable structures (which finds a parallel in the cellular differentiation mechanism).

These, and many other, loose similarities encourage us in our approach. Nevertheless, it is fairly obvious that a conventional von Neumann processor architecture does not fully meet the requirements of a bio-inspired approach. In selecting and adapting the TTA paradigm for the implementation of our systems, we believe that we have introduced some very important features that will become extremely useful for the implementation of systems inspired by all three axes of bio-inspiration.

Along the phylogenetic (evolutionary) axis, the features of *Move* process open several extremely interesting avenues of exploration. First, and more general, the possibility of defining functional and communication units tailored to the application can greatly reduce code size. For example, we have recently realized a multi-processor implementation of the JPEG algorithm, for which we defined a functional unit that directly executes a discrete cosine transform, greatly reducing the number of instructions required for JPEG encoding. This reduction is by itself an advantage for evolutionary processes that want to evolve code for this kind of processors. Moreover, the partitioning of this code into threads of execution (loosely equivalent to genes) to be handled by the processors within the array naturally defines a meaningful *block* size for evolutionary approaches and is a natural target for *morphogenetic* approaches [14][16] that couple evolution and growth.

In addition, the features of TTA architectures introduce novel, unexplored areas where evolutionary algorithms can become extremely useful tools. For example, we are investigating the use of such algorithms early in the processors' design flow, notably to explore the space of possible functional units for a given application. In fact, it is not simple, even for an experienced engineer, to identify sets of instructions that could be efficiently implemented by a dedicated functional unit. Early experiments using genetic algorithms to search for useful units are providing encouraging results.

On the ontogenetic (developmental) axis, the advantages of the *Move* architecture are more subtle, but nevertheless extremely interesting. For example, the small size of the code is an invaluable asset in the implementation of growth for complex systems, as it lowers the cost associated with the presence of the code for the entire system (the artificial genome) in each processor. Also, the possibility to customize the communication units simplifies the implementation of complex communication systems able to adapt the system to the environment, such as, for example, gradient-based systems that exploit on-line evolution [14][16]. In general, growth, seen in the context of the ontogenetic applications we have identified, requires the ability to implement cellular differentiation rapidly and efficiently, and the features of *Move* processors are ideally suited to this task.

From the fault-tolerance point of view, while the analogy with nature needs to be left partially aside, the structure of a *Move* processor is well-suited to implement self-repairing systems from several points of view. First of all, as we mentioned, the compactness of the code is an advantage when the artificial genome needs to be duplicated inside each cell (a great advantage for reconfiguration ,as we have shown in the past) and leaves open the possibility to sore two copies of this information in each cell (providing the same kind of redundancy present in natural DNA). Then, the reduction in the size of the register file compared to a conventional architecture (a consequence of the ability to move data directly from the output of a functional unit to the input of another) simplifies the recovery of the processor's state when on-line self-repair is required. Finally, the structure of the processor itself lends itself well to the implementation of self-testing strategies, a necessary preamble for self-repair.

The advantages of the TTA approach for the epigenetic (learning) axis are much more immediate and stem from the observation that communication are handled just like functional units and can be adapted to the application in the same way. What this implies, in turn, is that it is easy to change the connectivity within the array of processors, the most important feature in a hardware neural network. Without changing the programming of the processor, the connection units can be adapted to implement different connection patterns, ranging from simple local communication to mechanisms that exploit the features of the underlying hardware to implement, for example, run-time reconfigurable connections [15]. The same program can then exploit a wide variety of transfer functions and communication patterns. Coupled with a library of functional and communication units, this approach can then allow the extremely rapid prototyping of novel networks and learning algorithms.

From a structural point of view, independently from the axes, some interesting analogies can also be made. For example, an often criticized aspect of the processor/cell analogy is the crucial difference between a (complex) instruction in a computer program and a (simple) codon in the genome. A conventional instruction, in fact, uses 32 or 64 bits to code not only the place where the data have to be retrieved and stored, but also the operation to perform. The role of a codon is more simply (very loosely speaking) to activate a protein that in turn starts a chain of reactions involving the structural elements of the cell.

In this sense, the TTA approach (being a data-flow architecture) edges closer to biology be defining an instruction as a set of relatively few bits that activate different parts of the processor depending on the operation to be executed. The approach has the added benefit of considerably reducing the complexity of the decoding logic (see, for example, the fetch unit of our prototype in Fig. 4) and opens the way to interesting possibilities. For example, since the transport layer of the processor can be extended to handle several instructions in parallel, it becomes possible to realize processors with multiple decode units, allowing the artificial genome to be accessed in parallel in different places, as is the case for the natural genome.

In the next section we will analyze some of the technical details of our processors through a description of the prototype system we developed to test the feasibility of many of the basic concepts we have described. While no specific bio-inspired mechanism is present in the prototype at this stage, the implementation has been developed so as to easily integrate such mechanisms (e.g., growth and learning processes) in the next phase of our project.

5. A Prototype System

For the implementation of ontogenetic systems, one of the key features of a TTA processor is therefore the possibility to easily parametrize its structure. The fetch and decode subsystems, the transport layer (i.e., the busses that implement the datapath) and the functional units can each be modified almost independently to fit the application.

To test the flexibility of this approach, we realized a prototype to experiment with a possible implementation for each of these subsystems. Our implementation choices represent a fixed compromise between performance and size, but we designed the system so that the specific parameters used can be very easily adapted to shift the balance.

5.1. Fetch and Decode

The processor fetch and decode cycle is relatively standard: the code to be executed is stored locally in a small memory and at every clock cycle the instruction pointed by the program counter (PC) is loaded and decoded. According to the TTA approach, there exists only a single instruction (`move`) with two formats:

Address → Address :

$$0 \mid \underbrace{DDDDDDDD}_{\text{8-bit dest.}} \mid \underbrace{SSSSSSSS}_{\text{8-bit source}} 0 \qquad (1)$$

Immediate → Address :

$$1 \mid \underbrace{DDDDDDDD}_{\text{8-bit dest.}} \mid \quad \underbrace{IIIIIIII}_{\text{9-bit imm. value}} \qquad (2)$$

As a test of the parametrization capabilities of the approach, we adopted a VLIW (Very Long Instruction Word) encoding for our instructions, allowing our processor to execute two `move` instructions in parallel. Every 36-bit *instruction word* can then contain up to two of the above instructions. This parameter represents a compromise between program size and instruction level parallelism (ILP).

After a short decoding phase where eventual immediate values are extracted from the instruction, the *Fetch Unit* (Fig. 4) sends the source and destination addresses of the registers to the functional units through the transport layer. By permanently scanning the busses, the functional units can then know if they are involved the operation either as a source or a destination for the data transfer.

As each address in the instruction uniquely identifies an I/O register of a functional unit, the format of the instructions imposes some upper limits on the size of the processor (in this case, an instruction can address up to 256 source or destination registers). However, the size of the instructions, and hence the size of the processor, can be altered easily since the decode logic is extremely simple.

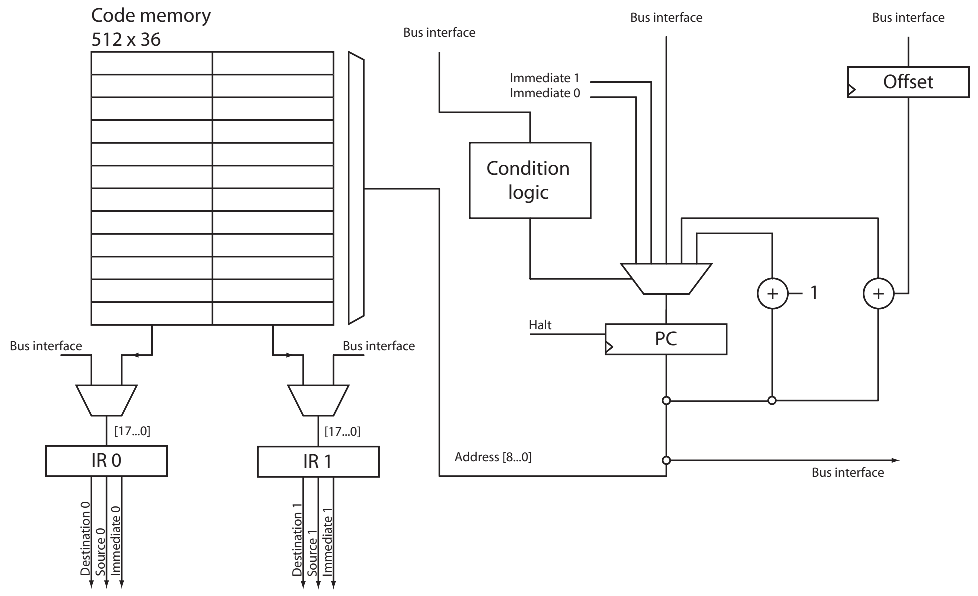

Figure 4. Layout of the Fetch Unit

5.2. The Transport Layer

In our implementation, we opted for a shared-bus topology for the transport layer: for each of the two instructions encoded in a word, three separate busses (one each for the 8-bit source and destination addresses and one for the 32-bit datum being moved) connect all the functional units of the processor.

This choice is more a matter of convenience that anything else: as the majority of common instructions require two operands, it is usually simple to find two `move` instructions that can be executed in parallel. An analysis of tradeoff between complexity and performance is one of the next steps of the project and one where some of the hints provided by biological systems could become useful.

5.3. The Functional Units

In the *Move* paradigm, the functional units define the instruction set of the processor by implementing the operations required by the application and by acting as sources or destinations of data displacements. This approach also implies that the instruction set can be easily modified by adding or removing functional units.

To implement this functionality, we have developed a common bus interface, used by every functional unit to connect to the transport layer. This interface lets heterogeneous components be accessed uniformly and allows the processor to be assembled using a library of pre-defined functional units (written in VHDL). We have then developed a small set of basic FUs, separated in three main classes:

1. *Computational Units*
   This class contains the classical arithmetic and logic operations found in a conventional processor. To this class also belong most of the application-dependent functional units that can be used to customize the processor. In our prototype, we included the basic operations: add, sub, multiply, shift, and some logical operations.

2. *Operational Units*
   This class contains the units required to control the processor, such as a *register file* containing a parameterizable number of general-purpose registers (eight 32-bit registers in our prototype), a *condition unit*, used for branching, offering several comparison schemes (the result of the comparison can then be moved to the fetch unit to serve as a condition for a *jump*), and a *data memory*, which corresponds to the data cache and to the data memory management unit and offers various addressing modes such as stack or auto-incremented addressing (a 512x32-bit memory in the prototype).

3. *I/O Units*
   This class is used to implement the network that connects the processors to each other. In our prototype, the I/O units are 32-bit registers used to implement a simple shared-bus topology that connects all processors in the system. However, as in the TTA approach outside communication is handled as a standard data displacement (a very useful feature of this kind of processors), these units can become arbitrarily complex.

5.4. Development Tools

To test the software-side implementation issues of the TTA paradigm, we wrote an assembler and a minimal simulation environment. These tools are *qualitatively* interesting, as we included many of the key elements required to efficiently exploit the features of *Move* architectures.

With an instruction set reduced to its simplest expression with only two variants of a single instruction, an assembler for a *Move* processor does not have to handle complex instructions encodings. However, it should be able to handle the increased load caused by the deliberate shift of complexity from the hardware to the software layer. In fact, the need to use only the `move` instruction makes programming a TTA processor considerably more difficult than a conventional one, since the level of abstraction usually provided by standard assembly languages is missing.

To overcome this problem, we have designed an assembler that offers an extendable set of *macro-instructions* used to define a ”meta-language” can be considered as an extension of the basic Move language that permits to express programming concepts more intuitively. In practice, we introduce a new level of abstraction whereby sets of `move` instructions are grouped to form the instructions of the new meta-language (e.g., the `add` macro-instruction corresponds to a set of three `move` instructions).

Using this approach, we defined a meta-language that contains all of the conventional RISC instructions (including abstractions such as conditional jumps, load and store instructions, or function calls). The assembler can use these macro-instructions in place of the primitive `move` instructions and thus allows out TTA processor to be programmed not unlike a conventional RISC processor.

The assembler itself has been coded using the JavaCC parser generator [19] with the JJTree extension for syntactic tree marching using the Visitor design pattern described by Gamma et al. in [5] and Palsberg in [12].

5.5. The Memory Map

As we mentioned, the *Move* paradigm implies that every FU corresponds to an address range in a memory map. To design application-dependent implementations, the address map of the functional units is defined in a file which is accessed at assembly time. This file makes the relation between the physical address space and a set of symbolic address names used in assembly code. As a consequence, the physical units (i.e., the VHDL code) are separate from their software representation, implying that the assembled code is compatible across implementations that share a common subset of FUs.

Defining the instruction set in a file also simplifies the specialization of TTA processors: if an algorithm would benefit from the use of a specific hardware function (e.g., a FFT), a custom FU can be designed and added to the memory map, where the assembler could directly exploit it.

5.6. Implementation

To verify the implementation of our prototype, we instantiated into a Xilinx Virtex II-3000 FPGA a matrix of twelve processors running at approximately 50 MHz (Fig. 5). Each processor (code-named *Ulysse*) is independent from the others and runs its own program, uploaded dynamically from the host PC. As we mentioned, the interconnection network is very basic (a shared-bus architecture where all processors and the host PC, who initiates all transfers, are connected by a single bus) but sufficient for the purpose.

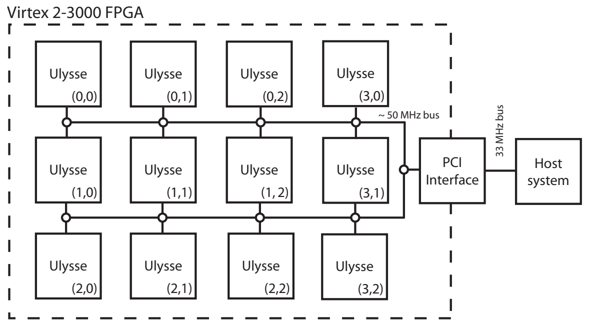

Figure 5. Overview of the prototype system.

To test the functionality of the processors, we built a demonstration application that uses the twelve processors to compute at the same time the factorial of a number input by the user and a graphical *plasma* effect. The latter is computed by combining various trigonometric functions taking the time and position of each considered picture element as parameters. The result is then displayed on the host screen in a Java GUI where the user can input values for the factorial program and visualize the real-time plasma effect.

As we mentioned, the goal of this prototype was to validate some of the key features of the processor, in view of their exploitation to implement bio-inspired systems. In particular, we tested the possibility to adapt the structure of the processors to the target application. The results are encouraging, as we were able to define a methodology to automatically add arbitrarily complex functional units to obtain a drastic reduction in code size, an interesting property that will provide a valuable field of application for evolutionary mechanisms. Moreover, among the tools that were not described in this article, we successfully implemented a monitoring system (currently used for debugging purposes) that could become the basis for the realization of the ontogenetic mechanisms involved in the growth of complex systems and in the reconfiguration needed to achieve fault tolerance.

6. Conclusions and Future Work

The implementation of bio-inspired systems in silicon in our approach amounts to the creation of massively parallel arrays of application-specific processors with properties, such as growth and self-repair, typical of biological entities.

Two practical considerations stand in the way of such an implementation. The first is technological: current reconfigurable circuit densities do not allow the realization of massively parallel systems. However, improvements in silicon technology and, eventually, the development of molecular-level circuits should not only allow such systems to be built, but even *require* some of their features.

The second consideration concerns the implementation of ontogenetic systems: there exists today no universal architecture for application-specific processors that can be used to implement effectively our approach. The processor architecture presented in this article, coupled with some of the bio-inspired concepts we have developed in earlier projects (a genome in every cell, growth algorithms that define system topology, dynamic routing networks, etc.), responds to many of the necessary criteria for the realization of ontogenetic systems and represents a step forward in the direction of systems that more closely resemble the organization and operation of multi-cellular systems in nature.

Future work within the project calls for two main axes of research. On one axis, we will pursue the development of the processor by investigating in detail the different implementation parameters for the processor and by setting up a complete design environment to simplify its use. In this context, we have recently completed an implementation of the JPEG algorithm on an array of processors, validating our approach on a real-world application. On another axis, we will integrate to the architecture the most interesting features of bio-inspired systems, introducing high-level processes such as growth, learning, and fault-tolerance.

Finally, by allowing the creation of application-specific functional units handled as "black boxes", a *Move* architecture can potentially be the object of an automated design flow, in which the functional units could be selected from pre-defined libraries and the architecture used to provide a framework in which the units are inserted. This approach, in addition to simplifying the design of bio-inspired processors also provides interesting opportunities for an efficient application of evolutionary mechanisms.

The features of *Move* processors, in fact, suggest several opportunities for the use of evolutionary techniques in the design of complex systems. While the evolution of an entire system consisting of the hardware (the array of processors) and of the software to implement the desired application remains beyond the capabilities of evolutionary algorithms, the TTA approach allows the problem to be easily *partitioned* into tractable pieces, opening several possibilities:

- While the evolution of circuits using FPGAs has spawned considerable literature (see [24] or [9]), no reasonable answer has been found for the *scalability* of such evolution. By partitioning the computational part of the processors into small, often combinational functional units, the *Move* processor represents an ideal platform for the application of evolution to digital circuits. In fact, while evolving an entire processor is not feasible, the evolution of single functional units could well be achievable given current computational power.
- An interesting design issue that has not, to the best of our knowledge, been explored in this precise context through evolutionary approaches is *mapping applications to connection networks*, that is, finding an efficient network to connect processors for a given application. A problem known to be NP-complete, it could become a useful target for evolution, particularly if the search space can be reduced. Since they handle communication units in the same way as functional ones, our processors can exploit libraries of predefined components that can simplify the definition of new communication networks and at the same time reduce the search space by limiting possible networks to hardware-friendly implementations.
- As we already mentioned, the TTA approach considerably reduces code size and thus increases the chance that a given problem could be evolvable. While this consideration is immediately applicable to assembly-level code, there exists an interesting opportunity to apply evolutionary algorithm at a higher level in the compiler hierarchy. More in detail, *Move* processors require that the code be transformed from its initial format (a high-level language) into an intermediate *data-flow notation*. This notation is in many respects closer to assembly code than to the initial high-level language and already benefits from the code shrink associated with the TTA approach while retaining some of the structural features of high-level languages.
- A very promising aspect of the TTA design flow where evolution could play an interesting flow is the selection of the appropriate functional units for a given application. This mapping, a key *hardware/software co-design* issue, is generally quite complex and because of that has been exploited only in very limited scope where evolution is concerned [18]. The "plug-and-play" use of functional units in *Move* processors considerably simplifies this *differentiation process*, since it reduces the problem to finding a mapping between the code and the units provided in the libraries. Evolutionary approaches can then be integrated directly within the design flow of our systems, possibly exploiting once again the intermediate data-flow representation.

We have begun investigating some of this areas. The preliminary results are very encouraging and will be the subject of future publications.

References

[1] J. Bongard and R. Pfeifer. Repeated structure and dissociation of genotypic and phenotypic complexity in artificial ontogeny. In *Proc, of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 829–836. Morgan Kaufmann, 2001.

[2] H. Corporaal. *Microprocessor Architectures – from VLIW to TTA*. John Wiley & Sons, 1998.

[3] H. Corporaal and H. Mulder. MOVE: A framework for high-performance processor design. In *Proceedings of the 1991 International Conference on Supercomputing*, pages 692–701, 1991.

[4] K. E. Drexler. *Nanosystems: Molecular Machinery, Manufacturing and Computation*. John Wiley, New York, NY, 1992.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[6] W. Goddard III, D. Brenner, S. Lyshevski, and G. Lafrate, editors. *Handbook of Nanoscience, Engineering, and Technology*. CRC Press, Boca Raton, FL, 2002.

[7] J. Han and P. Jonker. A system architecture solution for unreliable nanoelectronic devices. *IEEE Transactions on Nanotechnology*, 1(4):201–208, 2002.

[8] C. Lent and P. Tougaw. A device architecture for computing with quantum dots. *Proceedings of the IEEE*, 85:4:541–557, 1997.

[9] D. Levi and G. S.A. Geneticfpga: Evolving stable circuits on mainstream fpgas. In *Proc. 1st NASA/DOD Workshop on Evolvable Hardware*, pages 12–17. IEEE Computer Society Press, Los Alamitos, CA, 1999.

[10] D. Mange, M. Sipper, A. Stauffer, and G. Tempesti. Towards robust integrated circuits: The embryonics approach. *Proceedings of the IEEE*, 88(4):516–541, 2000.

[11] D. Mange, A. Stauffer, E. Petraglio, and G. Tempesti. Embryonic machines that divide and differentiate. In *Proc. 1st Int. Workshop on Biologically Inspired Approaches to Advanced Information Technology (BioADIT04)*, 2004.

[12] J. Palsberg and C. B. Jay. The essence of the visitor pattern. In *Proc. 22nd IEEE Int. Computer Software and Applications Conf., COMPSAC*, pages 9–15, 19–21 1998.

[13] H. Pearson. The regeneration gap. *Nature*, (414):388, 2001.

[14] D. Roggen, D. Floreano, and C. Mattiussi. A morphogenetic evolutionary system: Phylogenesis of the poetic circuit. In *Proc. 5th Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES2003)*, pages 153–164. Springer-Verlag, Heidelberg, DE, 2003.

[15] D. Roggen, S. Hofmann, Y. Thoma, and D. Floreano. Hardware spiking network with run-time reconfigurable connectivity in an autonomous robot. In *Proc. 2003 NASA/DoD Conf. on Evolvable Hardware*, pages 189–198. IEEE Computer Society Press, Los Alamitos, CA, 2003.

[16] D. Roggen, Y. Thoma, and E. Sanchez. An evolving and developing cellular electronic circuit. In *Proc. Artificial Life IX*, pages 33–38, 2004.

[17] E. Sanchez, D. Mange, M. Sipper, M. Tomassini, A. Prez-Uribe, and A. Stauffer. Phylogeny, ontogeny, and epigenesis: Three sources of biological inspiration for softening hardware. In *Proc. 1nd Intl. Conf. on Evolvable Systems: From Biology to Hardware (ICES96)*, volume 1259 of *LNCS*, pages 35–54. Springer Verlag, 1997.

[18] L. Sekanina. Virtual reconfigurable circuits for real-world applications of evolvable hardware. In *Proc. 5th Intl. Conf. on Evolvable Systems: From Biology to Hardware (ICES03)*, volume 2606 of *LNCS*, pages 186–197. Springer Verlag, 2003.

[19] G. Succhi and R. W. Wong. The application of JavaCC to develop a C/C++ preprocessor. *ACM SIGAPP Applied Computing Review*, 7(3), 1999.

[20] D. Tabak and G. J. Lipovski. MOVE architecture in digital controllers. *IEEE Transactions on Computers*, C-29(2):180–190, Feb. 1980.

[21] G. Tempesti, D. Mange, E. Petraglio, A. Stauffer, and Y. Thoma. Developmental processes in silicon: An engineering perspective. In *Proc. 2003 NASA/DoD Conference on Evolvable Hardware (EH-2003)*, pages 255–264. IEEE Computer Society Press, Los Alamitos, CA, 2003.

[22] G. Tempesti, D. Mange, and A. Stauffer. A robust multiplexer-based fpga inspired by biological systems. *Journal of Systems Architecture*, 43(10):719–733, 1997.

[23] Y. Thoma, E. Sanchez, J.-M. Moreno Arostegui, and G. Tempesti. A dynamic routing algorithm for a bio-inspired reconfigurable circuit. In *Proc. 13th Int. Conf. on Field-Programmable Logic and Applications (FPL03)*, volume 2778 of *LNCS*, pages 681–690. Springer Verlag, 2003.

[24] A. Thompson. An evolved circuit, intrinsic in silicon, entwined with physics. In T. Higuchi, M. Iwata, and W. Liu, editors, *Proceedings of The First International Conference on Evolvable Systems: From Biology to Hardware (ICES96)*, volume 1259 of *Lecture Notes in Computer Science*, pages 390–405. Springer-Verlag, Heidelberg, 1997.

[25] A. Tyrrell, E. Sanchez, D. Floreano, G. Tempesti, D. Mange, J.-M. Moreno, J. Rosenberg, and A. Villa. Poetic tissue: An integrated architecture for bio-inspired hardware. In *Proc. 5th Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES2003)*, volume 2606 of *LNCS*, pages 129–140. Springer Verlag, 2003.

[26] J. Vaario, A. Onitsuka, and K. Shimohara. Formation of neural structures. In *Proc, 4th European Conference on Articial Life (ECAL97)*, pages 214–223. MIT Press, 1997.