U.P.B. Sci. Bull., Series C, Vol. 72, Iss. 3, 2010 ISSN 1454-234x

MANY-PROCESSORS & KLEENE'S MODEL

Mihaela MALIȚA[1], Gheorghe M. ŞTEFAN[2]

Conform [1], [2] numim **multi-processors** *sistemele cu mai mult de un processor, în timp ce termenul de* **many-processors** *este folosit pentru sistemele cu un număr foarte mare de procesoare. Fundamentarea teoretică pentru cele două tipuri de maşini este diferită şi semnificativă pentru înțelegerea evoluției ştiinței calculului în era paralelismului emergent. În timp ce multi-processor-ul este o construcție fundamentată pe modelul lui Turing, many-processor-ul cere un context conceptual diferit. Propunem drept cadru conceptual pentru many-processor modelul lui Kleene. Exemplificăm abordarea de tip many-processor prin arhitectura cipului* ***BA1024*** *- un SoC complet programabil.*

According to [1], [2] more than one processor means **multi-processors***, while* **many-processors** *refers to big-n processors. The theoretical foundation for the two kinds of parallel machines is different and is meaningful for understanding the evolution of computer science in the emerging parallel computing era. While a multi-processor is a construct starting from Turing's model of computation, a many-processor is better explained in a different conceptual environment. We propose Kleene's model as a theoretical framework for describing the many-processor paradigm. In order to exemplify the many-processor approach the architecture of the* ***BA1024*** *chip, a fully programmable SoC, is presented.*

Keywords: computer architecture, parallel computation, many-processor, partial recursive functions, integral parallel architecture.

1. Introduction

Roughly speaking a multi-processor is a multi-threaded machine and a many-processor is a data-parallel machine. On the other hand, using Flynn's taxonomy, a multi-processor looks more as a MIMD machine, while a many-processor looks like a SIMD machine [3]. Building a multi-processor by *few* mono-processors is an incremental process, which can be easy controlled in the same theoretical framework, while putting *one thousand* cells on a single chip results in a strange "beast". The theoretical framework for such a machine cannot be the same as for a multi-core processor. We provide a way to understand a computation with *n* actors, that is a **many-processor** architecture. The distinction between *multi* and *many* is exemplified using *Intel Core 2*, and *Connex Core*.

[1] Prof., Saint Anselm College, Manchester, NH, mmalita@anselm.edu
[2] Prof., BrightScale, Inc., Sunnyvale, CA University and University POLITEHNICA of Bucharest, gstefan@arh.pub.ro

Intel Core 2 is a typical multi-processor where each CPU implements various forms of *transparent parallelism*, such as: super-scalar, pipeline and speculative executions. The two cores support multi-threaded executions. The multi-processor implemented as a multi-core machine controls the parallel execution at the program level. For this reason the computational model used to support mono-processors works very well for the multi-processor paradigm.

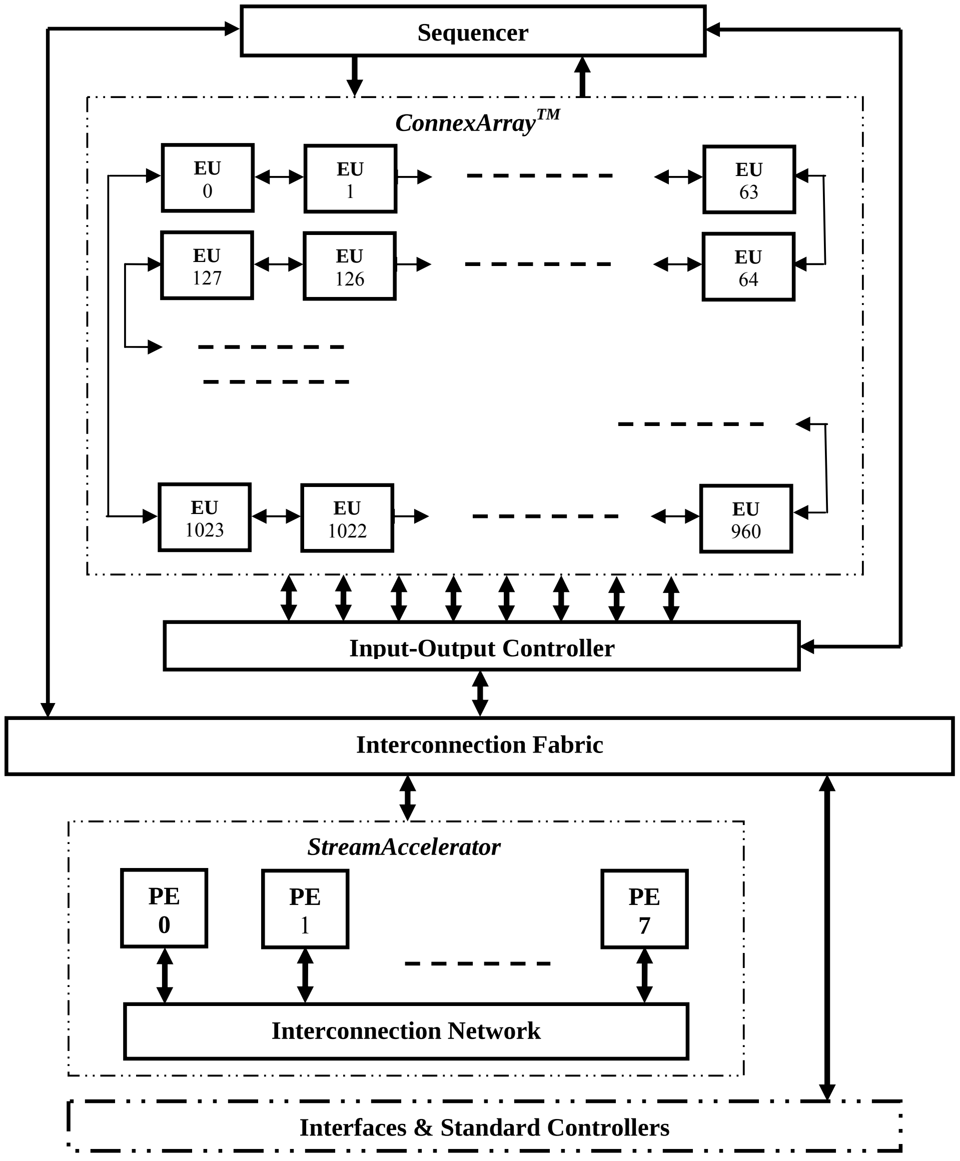


Fig. 1. **The *BA1024* chip**

Connex Core is a typical many-processor. Fig.1 presents the block diagram of the ***BA1024*** chip, a many-processor engine. Data intensive computations are done by the linear array called *ConnexArray*^{TM} (CA). One thousand *execution units* (EU), execute, according to their internal states, the instruction issued in each clock cycle by the *Sequencer*. Eight *processing elements* (PE) are devoted to accelerate pure sequential computations. The array, called *StreamAccelerator* (SA) is dynamically reconfigured by the interconnection network to solve problems like *arithmetic coding*. The *Interfaces & Standard Controllers* block contains general purpose interfaces (DDR, PCI, ...), specific interfaces (audio and video), and general purpose controllers (MIPS machines).

The difference between "multi" and "many" is rather *qualitative* than *quantitative*. The multi-threaded execution takes few of almost *independent* processes, while the data-parallel or time-parallel computation refers to ***n*** *interdependent* ones. There are well established techniques to deal with the multi-threaded approach, while we are in the infancy of the many-core approach - the programming techniques are far to be established. One main reason for this weakness is the theoretical framework which hosts the research in this domain.

Another difference, evident in this early stage of the split between "multi" and "many" refers to ***intensity vs. complexity***. Multi-processors deal with complex computations, while many-processors are comfortable with the intense ones. Thus, there are a lot of reasons to have a specific theoretical background for the many-processor paradigm, with an explicit reference to *n*.

2. Two different computational models

The architecture of the standard mono-processor computer derives from Turing's model [15]. The usual representation of Turing Machine (TM) is reformulated (see Fig. 2) in terms of real circuits [13], where the infinite tape is an infinite RAM, the access head is an up-down counter, and the control section is a finite automaton (FA). The Universal TM (UTM) is a special TM able to modify the content stored in a part of the tape, called *data* section, according to the interpretation of the binary content stored on the same tape, called *program* section. Optimizing the structure of the FA and the Up-Down Counter for a UTM we obtain the structure of a processor, used to process data stored in a finite RAM, according to the program stored in the same memory. Thus, the von Neumann style is supported directly by Turing's model. The model is expanded, using multi-threaded programming, to the multi-processor paradigm.

Turing's model can be used indirectly to ground the many-processor paradigm, but, the "many" aspect is not derived directly from this model, rather it results as an artificial construct based on mono-machines. We need to start directly with a model which "naturally" fits with a machine of thousands cells.

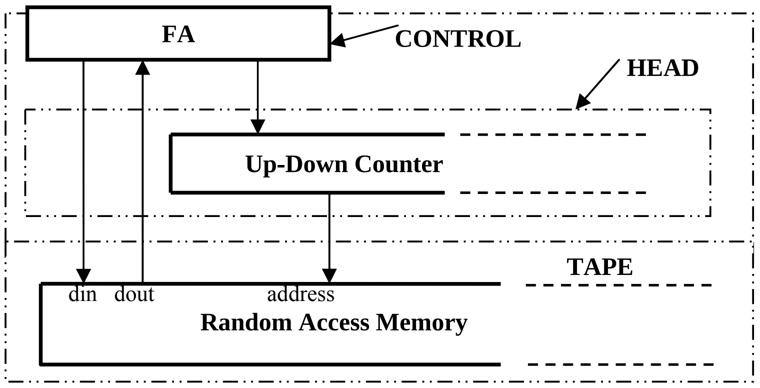


Fig. 2. **Turing Machine with circuit components**. The infinite tape is an infinite memory, addressed by an infinite up-down counter, which performs in each cycle a *read-modify-write* operation under the control of a finite automaton

2.1. Partial recursive functions

In the same year Turing published his paper, Kleene published the partial recursive functions model [4]. It consists in defining computation involving three basic functions (zero, increment, projection), and three rules for using them (the composition rule, the primitive recursive rule, and the minimalization rule).

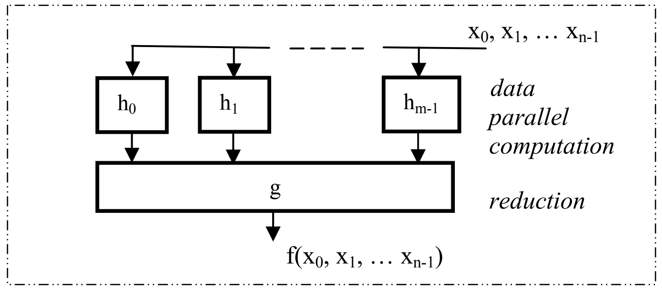

$x_0, x_1, \dots x_{n-1}$
h_0
h_1
h_{m-1}
data parallel computation
g
reduction
$f(x_0, x_1, \dots x_{n-1})$

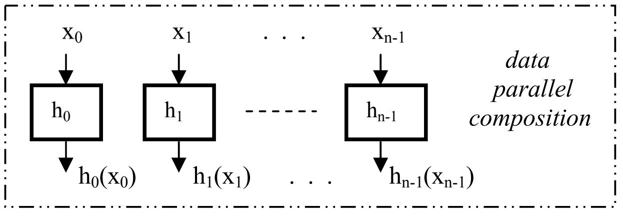
Fig. 3. **The structure associated with the composition rule**. The composition of function *g* with the functions $h_0, \dots, h_{m-1}$ is performed by a two level system. On the first level **many** functions are computed in parallel, while on the second a *reduction* function is performed

There are small, simple and fast circuits for the basic functions. An *increment* circuit has an optimal solution as circuit (polynomial size and poly-log time). For the projection function the *multiplexor* circuit is an optimal solution. The composition rule is actually the main rule. We show that all the other rules can be performed based on it. The physical structure associated with the general form of the composition is shown in Fig. 3. The first level performs *synchronic* *parallelism*, while both levels are involved in *diachronic parallelism*. The composition *contains all the features involved in performing the other two rules*.

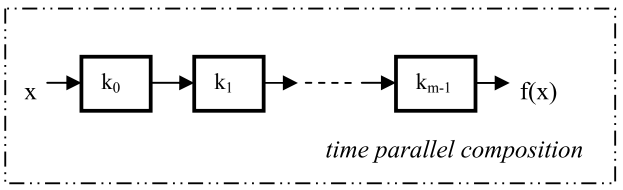
3. From partial recursive functions to many-processors

The general form of composition has *particular*, simpler forms able to express the other two rules. In Fig. 4, three particular forms are emphasized.

a.



b.



c.

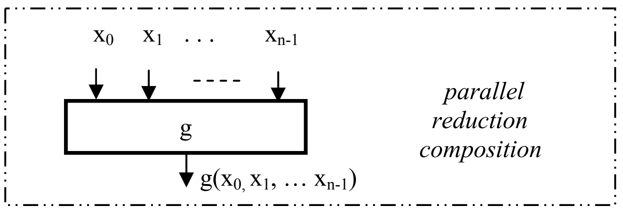


Fig. 4. **The three basic forms of composition**

Data parallel composition (see Fig. 4a) computes in parallel n functions, each applied to a component of the input vector $x_0, \dots x_{n-1}$, and generates the output vector $h_0(x_0), \dots h_{n-1}(x_{n-1})$. In this case g is the identity function.

Serial composition (see Fig. 4b) is defined for multiple applications of the composition with $n=1$. Results a pipe of m functions, $k_0, \dots k_{m-1}$, which computes $f(x) = g(k_{m-1}(k_{m-2}(\dots k_0(x)) \dots)$. If in each cycle a new value from the input stream, $<x_0, \dots x_{s-1}>$, is inserted, after a **latency** of m cycles results $<f(x_0), \dots f(x_{s-1})>$. The circuit computes in $s+m$ cycles m values for f. If $s >> m$, where s is the length of the input stream and m the number of functions composed in order to compute $f(x)$, then this kind of parallel computation is very efficient.

Reduction composition (see Fig. 4c) is when $h_i(x_i) = x_i$ for any i. The input vector $x_0, \dots x_{n-1}$ is reduced to a scalar. This function is usually performed in poly-log time and the structure has the linear size (the structure is a binary tree).

3.1. Implementing primitive recursion

The primitive recursion rule computes *f(x,y)* using *f(x,y) = g(x,y,f(x,y-1))*, where *f(x,0) = h(x)*. The circuit which computes *f* is presented in Fig. 5. It has an infinite pipe of circuits and a reduction network used to select the result.

The function performed by *H* is *H(x,y) = {x, y, f(x,0), (y == 0)}*. It sends to the first input of the reduction network, r_0 *= {f(x,0), (y == 0)}*, which is a pair *{scalar, predicate}*, and to the next level in pipe *{x,y, f(x,0)}*. The function performed by G_i is G_i*(x,y,f(x,i-1)) = {x,y,f(x,i),(y==i)}*. It sends to the corresponding input r_i *= {f(x,i), (y == i)}* and to the next stage *{x, y, f(x,i)}*.

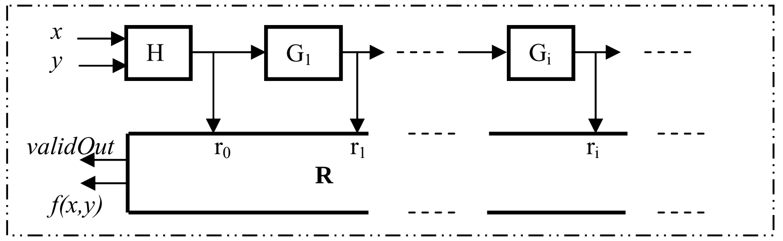


Fig. 5. **The primitive recursive circuit**. An infinite pipe of machines, *H*, G_1, G_2, ... ,G_i, ... and an infinite reduction circuit, *R*, as forms of composition, are used to build the circuit for applying the primitive recursive rule

The function *R* is defined on vectors of pairs *{scalar, predicate}*. *R* returns a scalar accompanied by a validating predicate, i.e., the output value, *f(x,y)*, is valid only if the predicate *validOut = 1*.

3.2. Implementing the minimalization rule

The minimalization computes *f(x)* to the value of the minimal *y* for which *g(x,y)=0*. The associated circuit is composed by two forms of composition (Fig.4a and Fig. 4c). The reduction selects the result from the output of a data parallel circuit (see Fig.6). The function performed by G_i returns *{scalar, predicate}*: G_i(x)= {i, (g(x,i) == 0)} = r_i. *R* selects from

{r_{0}, ... r_{i},...}={{0, p(0)}, {1, p(1)}, ... {i, p(i)}, ...}

the scalar of the first pair with *p(i)=1*, **if any** (*f(x)* is partial) and sends it to the output accompanied by the predicate used to validate the result.

The data parallel composition performs a **speculative** computation, computing *g* for *x* and "all" the values of *y* starting with *0*. The reduction function selects the **first** result. Consequently, for partial functions a special feature must be provided: the **function first**, *FRT(<B>) = <0, ... 0, 1, 0, ... 0>*, where *<B>* is a *n*-bit binary stream. The function *FRT* indicates the position of the **first** 1, **if any**, in the input binary stream *<B>*.

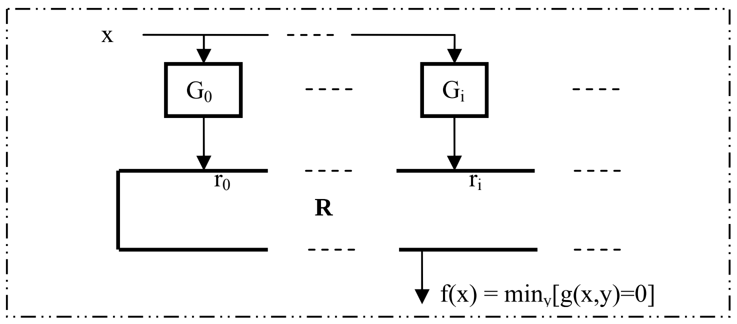


Fig. 6. **Minimalization circuit**. Each circuit G_i computes the function *g(x,i)*, and the reduction circuit selects the minimal *i* for which *g(x,i)=0*

4. The many-processor architecture

The model just introduced can be translated in a few ways into an actual universal machine. For many-processor architecture two data structures must be added to allow the description of the basic features: **vectors**, $\{X\}=\{x_0, x_1, \dots x_{n-1}\}$, and **streams**, $<X>=< x_0, x_1, \dots x_{n-1}>$, where x_i are scalars or Booleans. Data parallel composition means to receive input vectors, and to generate output vectors. Serial composition means inserting input streams and extracting output streams. Reduction composition receives vectors and outputs scalars or Booleans.

Data parallel composition can be involved efficiently in vector operations, and speculative computation. Serial composition is imposed by the pure sequential algorithms. Sometimes, serial composition asks for speculative computation. Reduction composition is involved with both, data parallel and serial composition. Therefore, it seems to be useful to define a computing system having the possibility to combine in a very flexible way all kinds of compositions.

4.1. Data parallel many-processor architecture

There are application domains characterized by very intensive data parallel computation with very small weight of the inherent serial computations. In this case a minimal system is implemented by three parallel resources: (1) the data parallel array, (2) the loop closed over the data parallel array through a two-directional *FRT* network, and (3) the reduction three. To each h_i an EU is associated (containing ALU, registers, data memory). The reduction composition is a tree structure which performs simple functions (extract and add scalars or Booleans). Each vector is distributed along the EUs. Each EU_i contains the component x_i in its data memory. Thus, all the components i of the vectors are processed in EU_i or in a small neighborhood, usually in EU_{i-1}, EU_{i+1}.

In order to link vector operations, in each h_i module is added a local scalar memory. The execution model is:

- in each clock cycle an **instruction sequencer** (IS) broadcasts one instruction to be executed by each EU (see Fig. 7)
- each EU executes the received instruction according to its internal state (stated by the selected Boolean), for example:
```
where (bool_vect_q == 1)
     vector_n = f(vector_m, vector_p);
  elsewere
     vector_n = g(vector_m, vector_p);
```
- the instruction operates on data stored in each EU and, sometimes, on some data stored in a small neighborhood (usually in EU_{i-1} and EU_{i+1})
- the sequence of instructions evolves according to the IS internal state and according to the scalars or Booleans provided by the reduction tree.

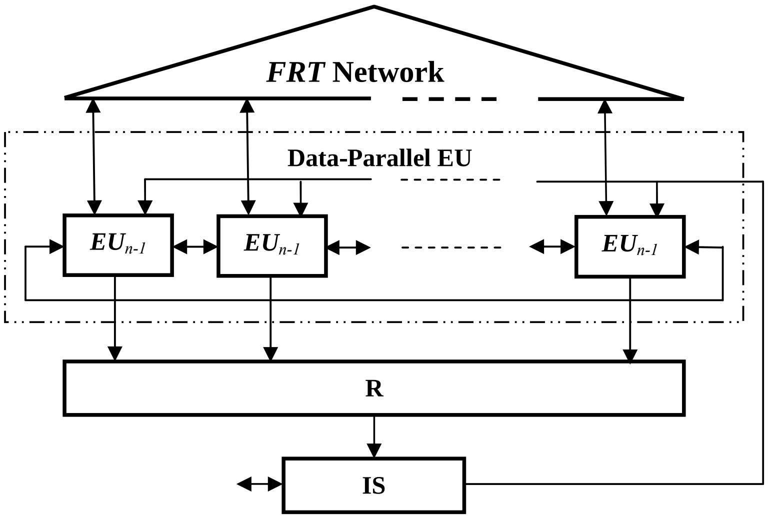


Fig. 7. **Data parallel many-processor architecture**. EUs are interconnected in a two-direction ring. The reduction tree R closes the loop back to the instruction sequencer (IS). The *FRT* network closes a loop over EUs

The minimal structure of a data parallel architecture is in Fig. 7. The partial recursiveness asks for the loop closed over the entire n-EU array through the *FRT* network [7] [8]. It classifies the EUs in: (1) the *first EU* (FEU) with the selected Boolean on 1, (2) the EUs positioned before FEU, and (3) the EUs following FEU. The main function is:

$$FEU(B) = PRX_{\mathbf{OR}}(B)\ \mathbf{XOR}\ (\ PRX_{\mathbf{OR}}(B) >> 1\ )$$

where: $B = \{b_0, \ldots b_{n-1}\}$, with $b_i \in \{0,1\}$, and $PRX_{\mathbf{OR}}(B)$ is the *OR prefix* function applied on B. Besides the function $FEU(B)$, the function $PRX_{\mathbf{OR}}(B)$ is used.

For input-output functions ***IOPlane***, where or from where a full vector is transferred in one cycle, is provided. Transparently to the main computation, this "plane" takes care of the communication with the external memory.

4.2. Integral many-processor parallel architecture

There are application domains with balanced data intensive and intensive sequential operations. More, the two types of operations are highly interleaved. Then, both, data parallel composition and serial composition must be supported by the same hardware [4] [14] [5]. The resulting structure is similar with the previous, with the difference that the EUs are substituted by PUs containing an additional *program memory*. An EU executes only the instruction received from IS, while PU executes also its own program stored in the local memory.

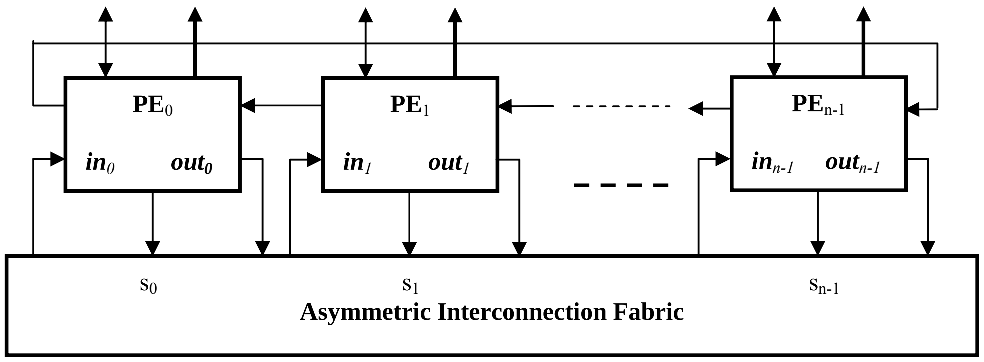


Fig. 8. **Data-Parallel PU**. The left connection of each PU is selected, by the 3-bit code s_i, from one of the previous 8 PUs

Another important difference is made by the way PUs are interconnected. If conditioned pipeline executions are performed, then the interconnection neighborhood must be expanded to allow speculative evaluations. If no more than m conditions are involved at any stage in the pipeline execution, then each PU_i must be able to select data from the previous 2^m PUs. The *Data-Parallel EU* (see Fig. 8) is augmented with an *Asymmetric Interconnection Fabric* (AIF) which allows each PU to select as left input the data from the previous 8 PUs ($m=3$). The AIF's outputs are selected according to the following expression:

$$in_{i+1} = S(s_i, out_{i-0}, out_{i-1}, \ldots) = out_{i-s_i}$$

The selection code, s_i represents the condition computed by PU_i. The condition selects the result of the speculation performed by the previous PUs.

Programming the serial composition part is done by defining the concept of **vector of functions**, $\boldsymbol{F} = [f_0, \ldots f_{p-1}]$, where f_i represents the local program executed by PU_i. If no conditional operation is performed, then f_i receives its input data from PU_{i-1}. If m-condition execution is performed, then PU_i selects (reduces) its input from the output of PU_{i-2^m}, … PU_{i-1} ad-hoc involved in a speculative data parallel array of 2^m PUs.

4.3. Segregated many-processor parallel architecture

If the data intensive computations and the inherent sequential intensive computations are grouped in clearly distinct stages of the application, then they deserve specialized hardware units. Because the number of EUs and of PUs can be stated independently, advanced optimizations are allowed. The structure consists in two subsystems: (1) a Data Parallel Many-Processor Architecture (Fig.7), connected with (2) a Data-Parallel PU (Fig.8).

5. Case study: the *BA1024* SoC

Using a segregated many-processor architecture, this section exemplifies the many-processor approach. The figures provided in this section allow us to understand the big differences between multi-processors and many-processors.

The ***BA1024*** is a SoC designed by *BrightScale, Inc*. to implement a platform for the HDTV market. The chip is implemented in *130nm* standard process and works at 200MHz. It contains the audio and video interfaces for two HD channels, 4 MIPS processors, the DDR interface (*3.2GB/sec*), an 128-bit interconnection fabric, and the intensive parallel machine (Fig.1) containing:

- **Instruction Sequencer** (IS): a 32-bit controller with stack architecture
- **Input-Output Controller** (IOC): another 32-bit controller with a stack architecture which communicates with the previous by interrupts
- ***ConnexArray***^{TM} (CA) a *linear* data-parallel array of 1024 16-bit EUs, where each EU contains:
    - a 16-bit integer ALU
    - a Boolean machine working on 8 Boolean variables
    - a local data memory for 256 16-bit words

  and receives in each clock cycle an instruction, issued by the IS
- a **global loop**, closed over CA, used mainly to identify the first EU with the selected predicate on 1
- a **reduction** tree to extract data and some critical parameters from CA
- ***IOPlan*** a *two-dimension* array which transfers data between CA and the rest of the chip under the control of IOC; the process is transparent

- ***Stream Accelerator*** (SA), a serial composition engine of 8 16-bit PUs, each having its own program memory with *m=4*.

One of the main design decisions was to keep the CA's interconnection network as simple as possible, while ***IOPlan*** was designed to perform in fly complex rearrangement of data. Another important design choice was to perform in each EU only simple functions (no multiplications, no floats). The level of simplicity was established (see also [10]) estimating the frequency of complex operations. Only the frequent operations must be performed by hardware in order to use the area very efficiently.

- General performances of the core of ***BA1024***:
  - *200 GOPS* (OP means 16-bit simple operations, no multiply or FP)
  - *3.2GB/sec* external bandwidth, *400GB/sec* internal bandwidth
  - *>60GOPS/W*
  - *>2GOPS/mm^2*
- Some video specific performances of ***BA1024***:
  - DCT: *0.15* clockCycle/pixel (on 8 × 8 arrays)
  - SAD: *0.0025* clockCycle/pixel

The previous figures resulted by running on **BA1024** programs in **CPL**, a language developed by BrightScale for the architecture of ***ConnexArray***^{TM}.

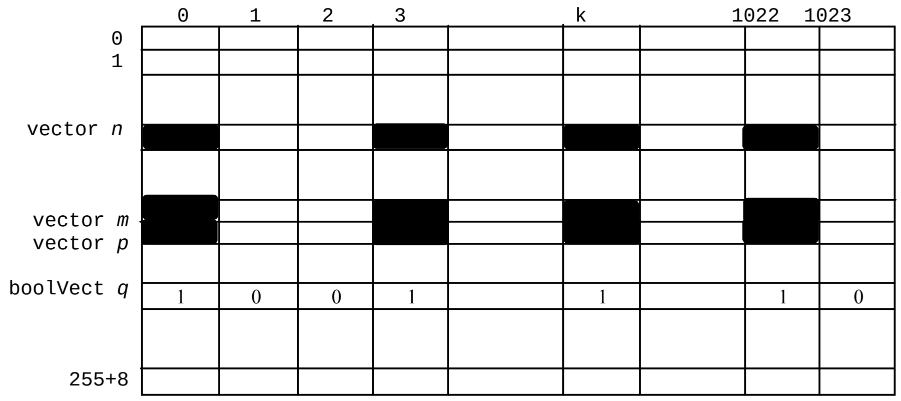


Fig. 9. **User's image of CA's content**. DPE contains 256 scalar vectors and 8 Boolean vectors. Only the components of the three scalar vectors *n*, *m*, and *p*, which are selected by the Boolean vector *q* are involved in the execution

Because the HDTV domain requests data intensive computations, CA is used in its full power. SA is needed to accelerate the pure sequential part of the codec (mainly for the H.264 standard). The architecture seen by the user of the CA is presented in Fig.9. It performs conditioned operations on 256 scalar vectors and 8 Boolean vectors.

6. Concluding Remarks

Partial recursive functions are computed using only various forms of composition. Elementary composition ($f(x) = g(h_0(x), h_1(x))$) can be used to build any composition, and composing particular forms of composition the primitive recursive circuit and the minimalization circuit can be implemented. More, the elementary multiplexor ($p = s' \& i_0 \mid s \& i_1$ which is a sort of Boolean composition) is the only brick used to build the circuit associated with the basic functions (increment, selection). Thus, basic functions are built using an elementary Boolean composition (the elementary multiplexor) and the circuits associated to the rules are built using elementary scalar function compositions.

The basic structures for building a many-processor are:

Data-Parallel EU: is a linear array of Execution Units with the simplest interconnection network. It performs only *data-parallel composition* (see Fig. 4a). Conditioned vector operations are mainly executed. The conditions are applied according to a Boolean vector, computed on the same structure. The construct

if - then - else

of the Turing approach is substituted by the

where - then - elsewhere

construct derived from Kleene's model.

Reduction network: is usually a tree network which receives a vector of scalars or Booleans and outputs a scalar or a Boolean. The most frequent functions are: selecting one scalar, adding scalars, OR-ing Boolean vectors.

FRT Network: is a function mainly imposed by partial recursiveness. It is defined on Boolean vectors and returns a Boolean vector with maximum one component having the value 1. It is a global function, because each output component depends on many input components (one of them depends on all components of the input vector).

Data-Parallel PU: is a "fat" linear array of Processing Units with an *Asymmetric Interconnection Fabric*. The interconnection network has a linear size, because it connects each PU with only a constant (usually small) number of previous PUs. The additional connectivity accelerates the pipeline execution when conditioned operations are performed.

Instruction Sequencer: is a conventional controller used mainly to issue in each cycle an operation to be conditionally executed in each EU. Sometimes it receives a scalar or a Boolean through the reduction network, and uses it to make decisions or to make sequential computations. If a Data-Parallel PU system is involved it is used to select the function vector.

Multi-processing for complex computation and many-processing for intense computation: The use of the multi-threading programming on multi-processors represents an incremental optimization of the solution offered by running similar programs on mono-processors.

The many-processor approach seems to be successful only if the complexity of the problem to be solved does not follow the intensity of the computation it involves. While the multi-processor solution remains to optimize complex applications, the many-processor solution applies where the intensity prevails complexity.

Complex versus intensive by numbers: There is a significant difference between the performance measures of standard mono- or multi-processors and those of machines making use of many-processors, of which the Brightscale BA1024 is an example. Typical values for today's architectures involved in complex computations are

- *4 GIPS + 4 GFLOPS*
- *(0.08 GIPS + 0.08 GFLOPS)/Watt*
- *(0.02 GIPS + 0.02 GFLOPS)/mm*^{2}

where an instruction is a 32-bit operation. For the intensive many-processor architecture of BA1024 chip, however, the measures are:

- *200 GOPS* **or** *2 GFLOPS + 100 GOPS*
- *60 GOPS/Watt}* **or** *(0.6 GFLOPS + 30 GOPS)/Watt*
- *2 GOPS/ mm*^{2}.

The two classes of architectures are perfectly differentiated by the **two orders of magnitude** separating the performance/power evaluation. For sure the two kinds of architectures are fundamentally different and they ask for different theoretical justifications. Turing's model manages complexity, while Kleene's model supports intensity.

Segregating the complex multi-processor architectures from the intensive many-processor architectures: The segregation between the two types of architectures is the best solution for optimizing, both *price (area) versus performance* and *power versus performance*.

High Performance = mono/multi-processor + many-processor.

Almost all demanding applications would benefit from the use of this new kind of segregated computing architecture. We claim that in the case of such architecture, **the complex part must be strongly segregated from the intensive part** in order to reach the target performance at a competitive price and with a minimum amount of dissipated energy. Maximizing the intensive part area is squeezed and power is saved.

Acknowledgments The authors of this paper got a lot of support from the main technical contributors to the development of the ***ConnexArray***^{TM} technology, the ***BA1024*** chip, the associated language, and its first application: Emanuele Altieri, Frank Ho, Bogdan Mîţu, Marius Stoian, Dominique Thiebaut, Dan Tomescu.

R E F E R E N C E S

[1]. *K. Asanovic, et al.*: The Landscape of Parallel Computing Research: A View from Berkeley, Technical Report No. UCB/EECS-2006-183.

[2]. *Shekar Y. Borkar, et. all.*: Platform 2015: Intel Processor and Platform Evolution for the Next decade, Intel Corporation, 2005

[3]. *M.J. Flynn*: "Very High-Speed Computing Systems", in *Proceeding of the IEEE*, 54(12), December 1966, p. 1901-1909

[4]. *S.C. Kleene*: "General Recursive Functions of Natural Numbers", in Math. Ann., 1936

[5]. *M. Maliţa, Ghe. Ştefan, D. Thiebaut*: "Not Multi- but Many-Core: Designing Integral Parallel Architectures for Embedded Computation" in International Workshop on Advanced Low Power Systems held in conjunction with 21st International Conference on Supercomputing June 17, 2007 Seattle, WA, USA

[6]. *B. Mitu*: private communication, 2005

[7]. *Ghe. Ştefan*: "The Connex Memory: A Physical Support for Tree / List Processing" in The Romanian Journal of Information Science and Technology, **Vol. 1**, Number 1, 1998, p. 85-104

[8]. *Ghe. Ştefan, D. Thiebaut*: "Memory Engine for the Inspection and Manipulation of Data", United States Patent 6,760,821, July 6, 2004; Filed: August 10, 2001

[9]. *Ghe. Ştefan*: "The CA1024: A Massively Parallel Processor for Cost-Effective HDTV", in SPRING PROCESSOR FORUM: Power-Efficient Design, May 15-17, 2006, Doubletree Hotel, San Jose, CA. and in *SPRING PROCESSOR FORUM JAPAN*, June 8-9, 2006, Tokyo

[10]. *Ghe. Ştefan, M. Maliţa*: "Granularity and Complexity in Parallel Systems", in Proceedings of the 15 IASTED International Conf, 2004, Marina Del Rey, CA, ISBN 0-88986-391-1, p. 442- 447

[11]. *Ghe. Ştefan, A. Sheel, B. Mitu, T. Thomson, D. Tomescu*: "The CA1024: A Fully Programmable System-On-Chip for Cost-Effective HDTV Media Processing", in Hot Chips: A Symposium on High Performance Chips, Memorial Auditorium, Stanford University, August 20 to 22, 2006

[12] *Ghe. Ştefan*: "The CA1024: SoC with Integral Parallel Architecture for HDTV Processing", in 4th International System-on-Chip (SoC) Conference and Exhibit, November 1 and 2, 2006 - Radisson Hotel Newport Beach, CA

[13] *Ghe. Ştefan*: "A Universal Turing Machine with Zero Internal States", in Romanian Journal of Information Science and Technology, **Vol. 9**, no. 3, 2006, p. 227-243

[14] *D. Thiebaut, M. Maliţa*: "Pipelining the Connex Array," *BARC07*, Boston, Jan., 2007

[15]. *A.M. Turing*: "On computable Numbers with an Application to the Eintscheidungsproblem", in Proc. London Mathematical Society, 42 (1936), 43 (1937).