Architecture of a FPGA-based Coprocessor: the PAR-1.

Javier Morán Carrera, Eduardo Juárez Martínez,
Sadot Alexandres Fernández, Juan M. Meneses Chaus.
Electronic Engineering Department, ETSI Telecomunicacion.
Universidad Politécnica de Madrid.
e-mail: moran@die.upm.es

Abstract

The implementation of a FPGA-based coprocessor and its programming methodology are shown. The effects of different sequencing models, and regular and irregular circuits on the hardware and in the programming methodology are discussed. Two examples are described: a sorting network and the kernel of a speech recognition algorithm. The results are still preliminary but they suggest some architectural improvements for general FPGA-based computing machines.

1 Introduction.

The advent of the Field Programmable Gate Arrays (FPGAs) [1] has enabled the realization of two important concepts: fast-prototyping systems, and custom computing systems. These two applications use FPGAs to support the implementation of logic circuits which are not known at the time of the system design. In fast-prototyping systems, the goal is to prove that a complex logic system works functionally as expected (that is, they are "fast functional simulators" [2]). With custom computers, the goal is to accelerate a program through the execution of critical operations on a coprocessor which provides reconfigurable hardware that evaluates directly the desired operation ([3],[4],[5]). In this article, we will focus on this second application.

The custom computers (FPGA-based custom computing machines or FCCMs) are built with basically three elements: FPGAs, memories and Field-Programmable Interconnection Circuits (FPICs). For some special applications, considerable speedups (>100) relative to conventional processors, have been reported ([3],[6]). The most important drawback of this approach is the effort required to program efficient algorithms on FCCMs, which is greater compared with the traditional software developing techniques, even with the support of logic synthesis tools.

In this paper, we present the implementation of a FCCM prototype, describe the experiences with it, and propose some architectural improvements. The paper is organized as follows: description of our prototype, programming methodology, applications, preliminary conclusions, and architectural proposals.

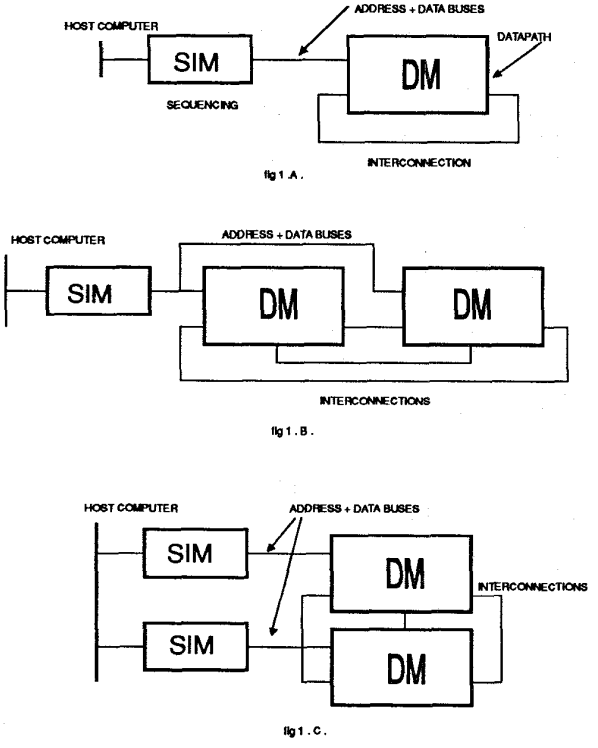

Figure 1. Different configurations of Datapath Modules and Sequencing Modules.

0-8186-7086-X/95 $04.00 © 1995 IEEE

2 Description of the PAR-1 Prototype.

In order to prove the concept and experience the real-life problems with the application of a FCCM, we decided to develop a prototype. The system is named PAR-1, the spanish acronym of Reconfigurable ALU Processor.

The current system is built around two kinds of modules: datapath modules (DM), and sequencing and interface modules (SIM). A sequencing module can command several datapath modules, which can be arranged with different topologies, as can be seen in figure 1. However, in the implemented prototype, the architecture follows the first drawing of the figure.

2.1 Datapath Modules (DM).

Each DM is composed by 4 FPGAs XC4008-6 with 512Kbytes of memory and 1 FPIC IQ160. Each FPGA is connected independently to 1 bank of 128 KBytes that can be addressed by the local FPGA or by a SIM. That means that there are two addressing modes: in the first case (master mode), the FPGA addresses directly its memory, with no interference with any other device; in the second case (slave mode), the memory (the data) is

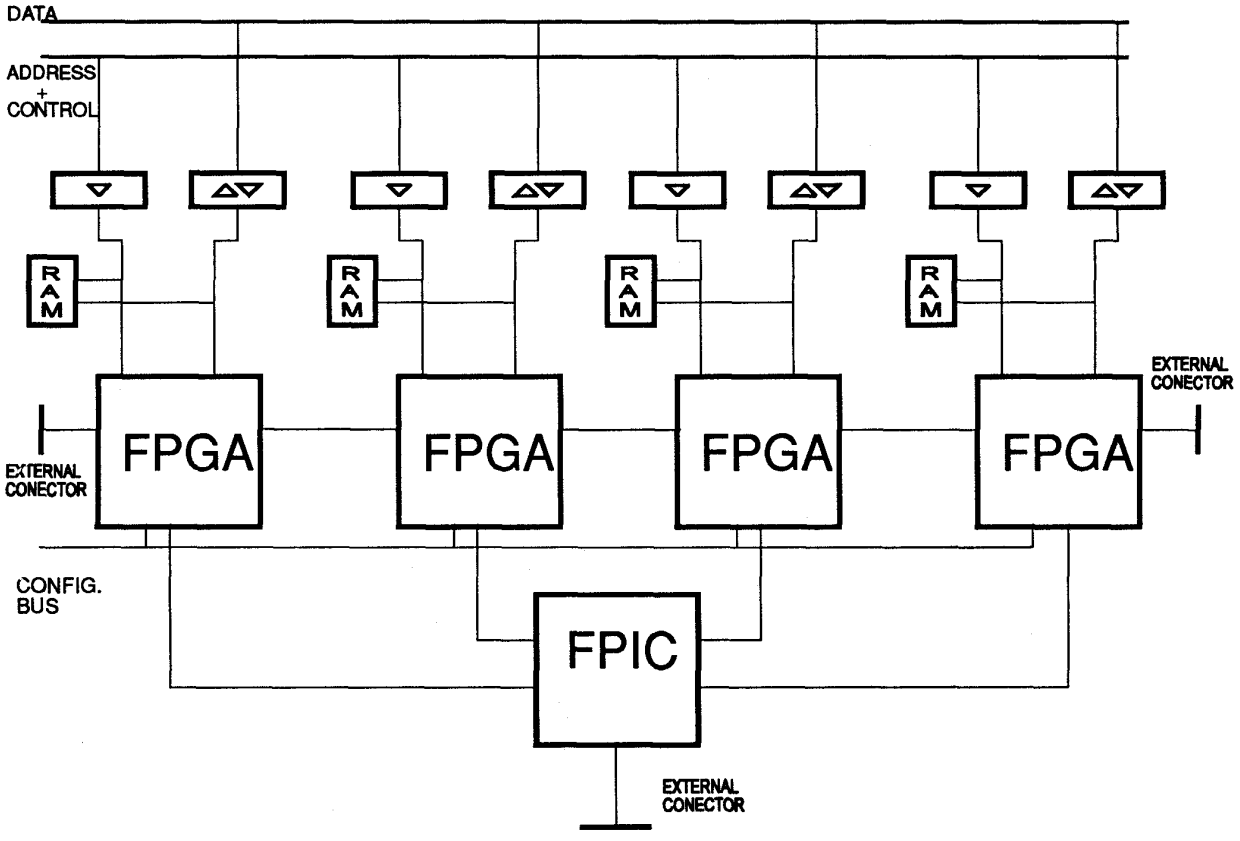

Figure 2. Scheme of the Datapath Module.

addressed (sequenced) by a SIM. These two addressing modes enable two different approaches to access the data: in the master mode, the DM memory can be seen as a 4-way (8 bits per way) distributed memory where each FPGA can perform its operations independently of the others; in the slave mode the DM memory is addressed only by one source (a SIM) and it can be considered as a single 128K x 32 memory. The two addressing modes provide more flexibility to implement algorithms on the PAR1.

The DM memory can be accessed by the SIM without programming any FPGA as a "gateway". This improves the access times for input and output (an important bottleneck in other systems that require the reprogramming of the FPGAs to provide a path to the RAMs) but requires a large number of buffers. Indeed, the memory organization can be considered the most distinctive feature of this architecture, compared with for instance the Anyboard I [2]. Its architecture is similar to the SPLASH2 machine [7], but smaller, and with a divided sequencing unit.

The FPGAs, memories, and the FPIC are connected in a linear array, as shown in figure 2. The reason for this topology is that it can support wide bus oriented implementations and because most systolic algorithms can be scheduled to fit on a linear array. The FPIC provides a path to connect the FPGAs in a more flexible way.

The DM has an external interface of 2 buses of 33 lines (connected to the FPGAs in the extremes of the array) and 1 bus of 28 lines connected to the FPIC. Additionally, there is an access bus for the memories (17 address lines, 8 data lines, 8 control lines) that also allows direct access to the FPGAs, and clock buses for synchronization of the devices. The FPIC can be removed, so it is possible to get 4 additional 32 bit buses.

All the FPGAs are configured in parallel, while the bitstream of each FPGA is loaded bit-serially. The FPGAs internal state can also be readback in parallel. The time required to program the module is currently around 590 milliseconds. It will be advantageous to accelerate the configuration times to reduce the overhead of configuration (or "instruction") loading. Each module includes a JTAG boundary-scan bus, that can be used for testing, reading and writing of FPGA configurations, and as a serial channel for user logic. The FPIC is also programmed using a JTAG bus.

2.2 Sequencing and Interface Modules (SIM).

The SIM is composed of 2 FPGAs, one devoted to the sequencing of the memory (a XC4008) and the other for interface and main control purposes (host interface, configuration loading and readback, etc...) (a XC4005). The interface FPGA is not programmable by the host processor. It has a 256 KBytes memory to store the configuration to be loaded on the DM and the sequencing FPGA, and their readback bitstreams. A scheme of the SIM is shown in figure 3.

As illustrated in the figure 3, there are buffers to isolate the activity of the buses, and the host can access directly the memories in the DMs. Notice that the sequencing FPGA has no "instruction memory". Currently, the "sequencing program" is stored or hardware-coded entirely in the FPGA.

In the current prototype, the host computer is a PC-type computer. A kernel provides the interface services with the application running on the host. The functions of this kernel include read and write operations to the configuration-readback memory and the DM-memory, configuration loading and readback, command operations to the sequencer, and clock generation (start - stop - step). The memory of the DM is directly mapped in the addressing space of the host computer.

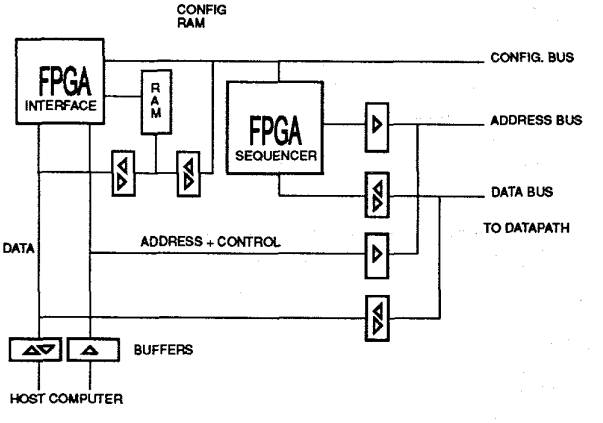

Figure 3. Scheme of the Sequencing and Interface Module.

3 Programming the PAR-1.

The programming methodology of the PAR-1 resembles those used in previous systems (see e.g. [7]). We have developed a Verilog model of the PAR1. With which, we simulate the application before actually programing it. Even with the readback support, the in-system debugging can be quite complex, and the simulation can prevent some simple problems.

We also distinguish between two kinds of circuits: a) regular circuits (usually composed by "small" cells working bit-serially, i.e. systolic designs) and b) irregular circuits (usually with "large" cells such as parallel adders). With regular circuits, it is important to control the logic mapping, partitioning, and placement-routing, in order to get good results: that currently means hand-work because we do not have tools to manage such circuits. However, because of the regularity of these circuits, the operations are not as complex as in

the case of irregular ones. With irregular circuits, we rely on the automatic tools to generate the results.

The programming flow of PAR-1 follows these steps:

1.- Algorithm extraction and architectural synthesis. First a decision must be made about what portion of the main algorithm is going to be implemented on the machine. That includes an analysis of the communication costs between the host processor and the coprocessor. Later, using estimates of the area consumed by every operator, the number of operators, its implementation (serial vs. parallel), and its scheduling is decided. These steps are done entirely by hand, they are quite heuristic, and require a deep knowledge of hardware techniques.

2.- Actual hardware specification. This phase considers separately the sequencing (control) part and the actual computation part of the algorithm. Each one has its own design methodology. The sequencing can be implemented as a microprogram of a microsequencer, or as a state machine. The first approach is more flexible and easily modifiable but it can be less efficient for some simple control and sequencing schemes.

The specification (implementation) of the computational part of the algorithm depends on its structure. In irregular designs, this is done mainly with the Verilog hardware description language (HDL) (we can also use VHDL, but we prefer Verilog because of our environment), and using a synthesis tool in a later step. For regular designs, it is much more efficient to capture a schematic (i.e. complete synthesis control) and develop *hard-macros* or *relationally placed macros* [15] which guarantee a good structure of the circuit.

Depending on the memory model employed, the placement of the sequencing and computational parts is different. When a centralized memory approach is used, the sequencing part is placed in a SIM and the computational part in the DM. When a distributed memory approach is used, the sequencing circuits are integrated together with the computational circuits in the DM.

3. Functional Simulation. The whole system is validated using the Verilog model of the PAR1. The tools produce Verilog models of each FPGA, that are inserted in the code of the general model of the PAR-1 for simulation. This procedure allows identification of problems which appear after a few clock cycles.

4.- Synthesis. If the hardware was specified with a hardware description language, a synthesis step must be performed. Also, some architectural optimization (with the XBLOX tool of Xilinx) is done.

5.- System partition. The actual hardware is partitioned to fit on the architecture of the system. The partitioning step must take into account the FPGA-memory interface mode (i.e. centralized addressing or distributed addressing). At present, this step is done by hand and during step 2 (i.e. before synthesis). An automatic partitioner is being developed.

6.- Placement and routing of each FPGA. In the case of regular designs (using lots of small *hard-macros* or *relationally placed macros*), it is convenient to fix the placement using constraints (additionally to the IO constraints). This improves greatly the quality of the final result. It should be very convenient to develop a tool to perform this task automatically, generating internal placement constraints to the PPR Xilinx tool.

7.- Timing simulation and timing analysis. A simulation with actual delays of the system (i.e. with all the FPGAs) can be performed. The timing simulation can be unnecessary if it is considered that some validation has been done during the functional simulation. If the circuit has been designed using a full synchronous approach, the timing simulation can be replaced with a timing analysis that will give the maximum work frequency of the system. Then, the system should behave like the functional simulation. We perform this timing analysis with the Veritime tool. This is similar to the XDelay tool provided by Xilinx for the timing analysis of a FPGA, but extended to the system level.

8.- Bitstreams generation, debugging, application integration and execution. The bitstreams generated by the Xilinx tools are reformated and combined to download all the FPGAs in parallel. Before integrating the calls to the PAR1 into the application code, the "hardwired program" is tested using a monitor which helps debugging in an isolated environment. The kernel in the host can include special functions to debug the application together with the PAR1 system.

The use of a Verilog model of the PAR-1 prototype has several advantages. We are no longer restricted to implementing another FCCM to prove a concept. Theoretically, we can simulate the behavior of a system with tenths of FPGAs, with other memory architectures or the support of some specific devices without any physical implementation. The main limitation of the simulation is that it is practical only with a small number of cycles and low complexities.

The tools employed by us include FPGA Compiler of Synopsys Inc. (synthesis), Composer (schematic capture) , Verilog (simulator) and Veritime (timing analysis) of Cadence Inc., and the tools of Xilinx Inc.

4 Applications.

We show two applications which have been developed with the previous methodology, and simulated. They have not yet been completely tested on the prototype. These aplications illustrate the design methodology, and some tradeoffs and problems in FCCMs.

The first application is a regular design. It is a sorter circuit which takes an array of 32 bits integers and outputs an (ascending or descending) ordered array. Sorting has attracted a great deal of attention and several algorithms for VLSI implementation have been proposed. Both serial and parallel hardware approaches have been analyzed, with several area-time trade-offs. (See e.g. [8]). It is a common task suitable for execution by a special coprocessor. Its main application is in database processing. A related operation is merging of sorted arrays.

The second application is an "irregular" design, part of a speech recognizer algorithm which uses what are called "Hidden Markov Models" or HMMs. A chain of Hidden Markov Models is defined by a set of states and transitions among states. Each state has a probability value which is computed from those of the transitions. The purpouse of these models is to correlate a sequence of symbols with a spoken word.

We use the Viterbi Algorithm to evaluate the HMMs, We will not discuss here how it works (see [10]). The actual algorithm is more complex because the data to be computed is indexed by arrays, requiring some address calculation. We have chosen this application because of our familiarity with this algorithm. We have implemented it in transputer arrays and in standard cells ASICs, and it will let us compare the speeds of the different approaches. However, we think that this is not an application where the use of a FCCM can provide an advantageous edge, because it is a task to be performed by a dedicated hardware (a stand-alone special purpouse DSP). It is described here because it helps to understand the problems that must be faced. In both applications, we will concentrate on the datapath aspects. The sequencing aspects will be considered later.

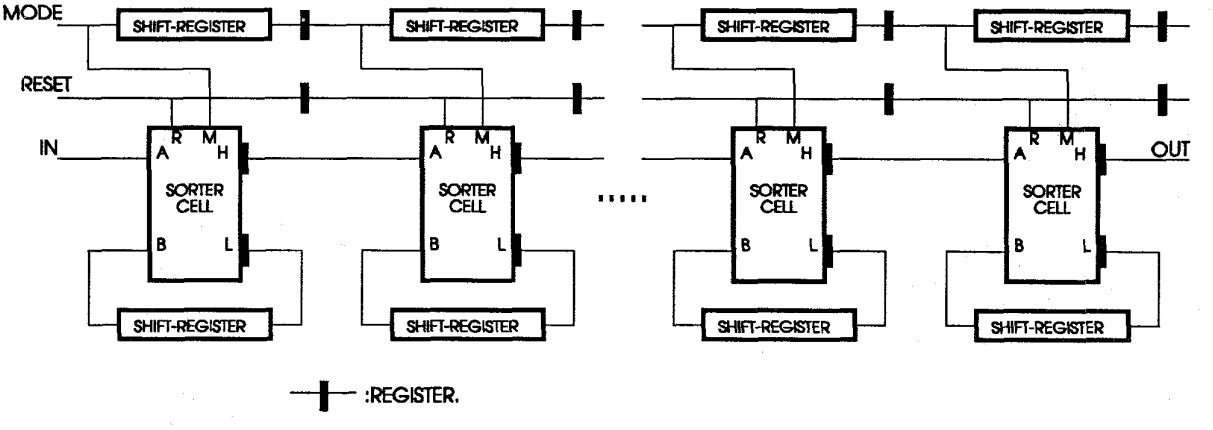

Figure 4: A Sorter Array.

4.1 Sorter Application.

The sorter is based on the design presented in [9] and a more extensive explanation can be found there. It is a linear array of cells as shown in figure 4. Each cell has a bit-serial comparator, two 1-bit multiplexers, and two 32 bit shift registers, plus some flip-flops. There are two control signals in each cell, a "reset" which indicates that a new word is coming and initializes the comparator, and a mode signal which controls the way the data is ordered (ascending or descending). The keys to order are assumed to be unsigned integers.

Every cell of the array receives a data word which is introduced serially most significant bit (MSB) first. The incoming data is compared bit by bit with the data in the shift-register. Depending on the sorting mode (ascending or descending mode), if the incoming word is bigger (in descending mode) or lower (in ascending mode) than the word stored in the shift-register, it is passed to the output. Otherwise, the data previously stored in the shift-register is passed to the output, and the incoming data goes to the shift-register. Figure 5 shows the internal structure of the cell. The serial comparator works according to the state machine of the figure.

To initialize the sorting network, we must first shift out all spurious data stored in the internal shift-registers. This is performed by changing the sorting mode and introducing a "separator": "+∞ " (i.e. the highest possible data) or a " -∞" (i.e. the lowest possible). To sort an array in ascending (descending) mode, first a "+ ∞ " (a " -∞") is introduced in descending (ascending) mode. Then after this word, the data is introduced normally according to the selected mode. After every word is introduced, the comparator must be reset to accept a new word for a new comparison. Note that the mode bit is shifted at the same time as the data, with a dedicated shift-register.

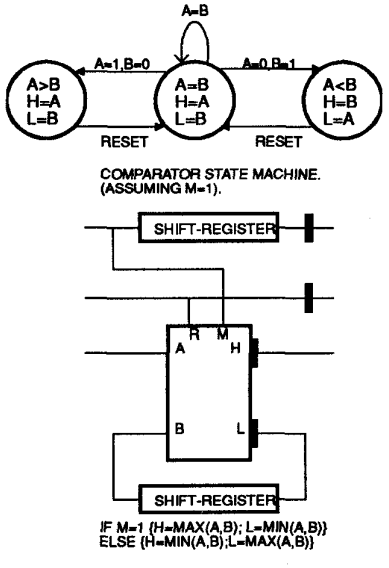

Figure 5: Details of the sorter cell.

The sorting of two sequences can be pipelined, without wasting cycles. After the first sorting, a "+∞" (or a "-∞" in accordance with the previous ordering mode) must be introduced as separator. Then the data of the second array is input and inverted (that is we change the relative magnitude of the keys), and the sorting mode changed. When the data of the second array leaves the array, it must be inverted again to restore the original value. These details are covered in [9].

Assuming M the number of cells, N the length of the vector to sort, and B the length in bits of the words, the circuit has a delay of 2(M+1)B cycles to sort and output all the valid data, and approximately (N+1)B cycles to sort a complete vector. This circuit can not sort sequences with length longer than the array (that is M≥N). Otherwise, it will retain in the shift-registers of the cells array the biggest or smallest data in the vector as selected in the sorting mode, and flush out all the other.

On the XC4000 FPGAs, the shift-registers are implemented using the CLBs RAM plus a counter. Less CLBs could be used if the counters of the shift-registers are shared by several cells. Note that if another kind of FPGA without any internal RAM capability is used, this circuit would require too many flip-flops to be efficient. The internal RAM also reduces the number of transactions to the external memory, needed to order all the data. Usually keys are sorted with pointers to the actual data. This circuit can also be used in this case, just feeding first the key, and then the pointer. The data will be sorted using the key, and if two keys are the same, using the value of the pointer.

In XC4000 FPGAs, a sorter cell for 32 bits long data and local counter, requires 9 CLBs each where 2 CLBs are used for the shift-registers. If the data is longer, more CLBs would be necessary to implement the shift-registers. The circuit can be theoretically clocked at 40 MHz, but we will make the experiments at 20 MHz. On the XC4008, 32 cells can easily be allocated in one FPGA. A DM of the PAR-1 can accomodate 128 cells.

We have compared this architecture against the `qsort` function of C, running on a SUN 10/30 workstation (128 words long arrays, 32 bits per word). The speedup is at first, disappointing: around 5.3 (the computation time depends on the data distribution). For longer arrays, the speedup is higher. Notice that the computing time using the hardware is independent of the data distribution, while the software can take advantage of "almost-sorted" sequences .

There are several trade-offs between frequency of operation, shifter implementation (i.e. with local counters or not), and the width of the comparator (i.e. 8 bits wide instead of pure bit-serial) that have not been entirely explored, and could lead to better implementations. For instance, some preliminary evaluations indicate that a sorter for 128 bits long keys,

using a 8 bits wide comparator in the sorter cell, can be implemented with 25 CLBs, with a better area-time factor than the full serial cell described above. We think that speedup of 50 could be achieved with more parallel algorithms, but more FPGAs would be required to sort vectors of the same length. Also, other hardware oriented sorting algorithms could be used, which could lead to improved results.

4.2 Hidden Markov models application.

The calculation of a state probability in a chain of Hidden Markov Models can be performed quite easily using the Viterbi Algorithm. The Verilog code of figure 6, illustrates how it can be done, in a completely synthethizable description. The variables alfa_in and alfa_out are the values of the state probabilities before and after calculation. All other variables are the transitions probabilities. This description does not consider any hardware limitations. The HMM module receives its data from the memory. Accessing the data is not easy because some indexing operations must be performed first. However, we will not address this sequencing problem (although a very important one) but concentrate only on the datapath aspects: the HMM module. The module requires 4 input data and generates 1 output data. (The reset and clk inputs are control signals, provided by the sequencer). Three addtions and one comparison have to be performed.

```
module HMM (alfa_in, alfa_out,
         self_prob, trans_prob, obs_prob,
         reset, clk);
input [31:0] alfa_in, self_prob,
trans_prob, obs_prob;
output [31:0] alfa_out;
input reset, clk;

reg [31:0] alfa_out, c1, c2;
always @(posedge clk)
  if (reset) c1=0;
  else
   begin
    c2=alfa_in + self_prob;
    if (c1>c2) alfa_out=c1 + obs_prob;
          else alfa_out=c2 + obs_prob;
    c1=alfa_in + trans_prob;
   end
endmodule
```

Figure 6: Verilog code to evaluate a HMM state.

The main problem in the implementation of this code on the PAR-1 is that the datapath modules have only a single memory port. This limitation means that the data accesses have to be done one at a time. That is, a minimum of 5 memory cycles must be performed, one for each parameter. The memory bottleneck has other effects: a) because data arrives in different clock cycles, more registers are required to store the data from the memory in the FPGA, b) the reduction of some hardware requirements, because it allows to reuse operators. For instance, the code in figure 6 has 3 add operations and 2 registers (c2 does not require a register). With scheduling, this becomes one adder and 4 registers. Figure 7 shows the new scheduled code with 6 states, a single bidirectional memory port.

```
module HMM (memory, reset, clk);
inout [31:0] memory;
input reset, clk;
reg [31:0] c1, c2, R1, R2;
reg [2:0] state;
reg c1_or_c2;

assign memory=(state==5)?R2:32'bz;

always @(posedge clk)
  if (reset)
    begin c1<=0; state<=0; end
  else
    case (state)
    0:begin R1<=memory; state<=1; end
    1:begin R2<=memory; state<=2; end
    2:begin
      c2<=R1+R2; R2<=memory; state<=3;
      end
    3:begin
      R1<=R1+R2; R2<=memory;
      c1_or_c2<=(c1>c2); state<=4;
      end
    4:begin
      if (c1_or_c2) R2<=c1+R2;
         else       R2<=c2+R2;
      c1<=R1; state<=4;
      end
    5:state<=0;
  endcase
endmodule

/*Scheduling:
In state 0 alfa_in is input.
In state 1 self_prob is input.
In state 2 trans_prob is input.
In state 3 obs_prob is input.
In state 4 a sum is performed.
In state 4 alfta_out is output.
*/
```

Figure 7: Scheduled Verilog code.

The new scheduled code is a state-machine in which each state can be considered to be equivalent to the instructions of a processor. In each state, a transaction from (to) the memory is performed, plus some other operations. Registers R1 and R2 are reused

to hold several variables. Also the adder can be reused, but at the cost of several multiplexers. (The code in figure 7 does not include the extra support required for its integration on the datapath module of PAR1). This is a typical scheduling problem in conventional processors. Note that we have not considered any limitation due to the FPGA core. The computational limit is imposed by the memory interface.

This application is synthesized later with the FPGA Compiler of Synopsys. Even if this application does not take advantage of the FCCM capabilities due to the memory bottleneck, our preliminary estimates indicate that it performs almost as fast as other implementations on ASIC or arrays of transputers. A possible solution to the memory bottleneck could be to use the internal FPGA RAM as a cache unit which could load pages of data. However, for this application, much more FPGAs or denser ones, would be required.

4.3 Data sequencing.

The data sequencing in the sorter application is rather simple, because it only indexes two vectors, one for reading the data and the other to write the sorted data (basically two counters). Notice that the memory is not a bottleneck in this application because the processing of the data is serial. If the data were processed in parallel, this could be a problem because the FPGAs can not work faster than the memory.

The data sequencing in the HMM application is more complex. The sequencing specification must follow the organization showed in figure 7. The address generation and control states, and the datapath states must be correlated. If wait states (e.g. for address calculation in the datapath) must be introduced, they will affect to both sequencer and datapath.

5 Some preliminary conclusions of the PAR-1 prototype.

These are some of our first conclusions with the PAR-1 architecture, that, we think, could be applied for other FCCMs:

- Organization of the memory. In applications like the HMM one, the real bottleneck is not the computing core (i.e. the FPGAs) but the number of ports of the memory. For these applications, the FPGAs will be underutilized because the bottleneck limits the achievable speedup. The implementation of a full multiported memory with enough capacity (>1 MByte) is very expensive but an interleaved memory (much like that of the vectors processors) could solve the problem. (A FCCM machine with a dual port memory-FPGA interface has been presented in [11], but we think, with too few memory). The same memory organization issues of the traditional computer systems must be faced by the FCCMs: central organizations much like vector processors or distributed organizations like those found in multiprocessors. We think that the first model suits better the capabilities of FCCMs. (S. Guccione and M. Gonzalez have proposed such machine in [12]).Also memory hierarchy concepts (cache) may have to be considered. One of the main differences between a fast-prototyping system and a FCCM, is the way in which the memory is organized. In a fast-prototyping system, no such real memory exists; the data is input to the FPGAs core much like a pattern generator, (i.e. with limited control in the sequencing), and the results of the computation are stored on a "logic analyzer".

- Data sequencing and control. Data sequencing is achieved in a similar way in many applications. A custom device could perform it much faster and more flexibly than a FPGA-based sequencer. We are actually designing such a device: it is similar to a VLIW processor with a conventional multiported register set, several ALUs, an internal instruction memory, and an interface to an external data memory. This custom device could perform other operations like memory management, parallelism support, or communications. Note that this approach changes the way an application must be designed; the address sequencing would not be "hardware", but a "sequencing program" that later, will be executed. This will make the design process easier, faster, and more parametrizable. (Some architectures use conventional processors to generate the addresses, but the lack of a specific port to address the memory, makes them slow). The FPGAs and FPICs (i.e. the datapath core) should be devoted only to the datapath. The memory address calculations, and all the memory related operations (i.e. interface operations, communications or even caching) must be performed by a much more specific hardware.

- FPGAs for computing. Some proposals have been made to fit current FPGAs for computing. Fast reprogramming or operation switching are capabilities which have already been requested by several authors. This rapid switching should not be

at the expense of the CLBs internal RAM capability, because it can save considerable amounts of registers in bit-serial and other applications. In [13], a FPGA architecture for reconfigurable datapaths has been proposed. Also parallel-serial converting capabilities to overcome the pin limitations of current FPGAs has been proposed in [14]. We find a lack of high-fanout signals. These signals can be very useful for bit-serial processing, where several common signals must be distributed to address CLBs RAM. We think that more high fanout networks (at the expense of high clock frequencies) would improve the capabilities of the FPGAs for computing applications.

6 Further work.

Our work is now being focused on the following topics:

- Design of a custom address sequencer. We are currently developing an integrated circuit to perform this task. This circuit would replace the sequencing FPGA in the SIMs of the PAR-1. The development of this circuit requires a software which helps to generate the data sequencing programs.

- Evaluation. The modeling of boards and systems with a hardware description language allows evaluation of the benefits and drawbacks of different architectural proposals.

- Automatic hardware partitioner. We need a partitioner to help in the design process. It can also give more insight into architectural questions relating to FCCMs.

- Applications. We have already analyzed some techniques to improve the arithmetic capabilities of FPGAs, in particular the use of serial signed-digit on-line arithmetic [15]. We are also working on neural network implementations on FPGAs, and other symbolic applications like the sorter described. To validate these applications, it is necessary to contrast the results obtained in a conventional computer, with those obtained using a FCCM implementation or model. This task has not been realized yet.

7 Conclusions

We have shown the architecture of a FPGA-based coprocessor, the PAR-1. Two applications, a sorting network and the core of a speech recognition algorithm have been described. Certain limitations have been identified, and some potential solutions suggested. We believe that future developments in FCCM hardware design, should be addressed to the memory organization and the sequencing units, instead of the core of FPGAs or FPICs. The adoption of a programming model for FCCMs (vector-based models or data parallel model) would be required to follow this research.

Acknowledgements.

We acknowledge the work of the students Cristobal Bonillo and Julio Arrebola, who helped in the design of the PAR-1 board. This work has been partially supported by TIC92-0083 (CYCIT- Ministry of Education and Science). To Santiago Morán, *in memorian*.

References

[1] S. D. Brown, R. J. Francis, J. Rose, Z. G. Vranesic, *Field-Programmable Gate Arrays*, Kluwer Academic Publishers, 1992.

[2] D. E. Van Den Bout, J. N. Morris, D. Thomae, S. Labrozzi, S. Wingo, D. Hallman. *AnyBoard: an FPGA-Based Reconfigurable System*. IEEE Design & Test of Computers, September 1992.

[3] M. Gokhale, W. Holmes A. Kopser et al., *Building and Using a highly parallel programmable logic array*. IEEE Computer. January 1991.

[4] P. Bertin, D. Roncin, J. Vuillemin. *Introduction to Programmable Active Memories*. Research Report 3. June 1989. DEC Paris Research Laboratory.

[5] P. Athanas, H. Silverman, *Processor reconfiguration through instruction-set metamorphosis: Architecture and compiler*. IEEE Computer , March 1993.

[6] P. Bertin, D. Roncin, J. Vuillemin. *Programmable Active Memories: A Performance Assesment*. Research Report 24. March 1993. DEC Paris Research Laboratory.

[7] J. M. Arnold, D. A. Buell., *VHDL Programming on Splash2. More FPGAs*, Abingdon EE&CS Books. 1994.

[8] C. D. Thompson. *The VLSI Complexity of Sorting*. IEEE Trans. on Computers, Vol. C-32 N. 12, December 1983..

[9] M. Afghahi. *A 512 16-b Bit-Serial Sorter Chip*. IEEE Journal of Solid-State Circuits, Vol. 26 N. 10, October 1991.

[10] S. Alexandres, J. Morán, J. Carazo, A. Santos. *Parallel architecture for real-time speech recognition in spanish.* International Conference on Acoustics, Speech and Signal Processing. ICASSP 1990.

[11] C. Iseli, E. Sanchez. Spyder*: A Reconfigurable VLIW processor using FPGAs.* Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines. April 5-7 1993. Napa,California

[12] S. A. Guccione, M. J. Gonzalez. *A Data-Parallel Programming Model for Reconfigurable Architectures.* Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines. April 5-7 1993. Napa,California

[13] Q. Wang, P. Glenn Gulak. *An Array Architecture for Reconfigurable Datapaths. More FPGAs*, Abingdon EE&CS Books. 1994.

[14] J. Babb, R. Tessier, A. Agarwal, *Virtual Wires: Overcoming Pin Limitations in FPGA-based Logic Emulators.* Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines. April 5-7 1993. Napa,California

[15] J. Morán, I. Rios, J. Meneses. *Signed Digit Arithmetic on FPGAs. More FPGAs*, Abingdon EE&CS Books. 1994.

[16] *The Xilinx Data Book.* Xilinx, Inc 1994.