VHDL Modeling of Fast Dynamic Reconfiguration on Novel Multicontext RAM-based Field Programmable Devices

Julio Faura*, Juan Manuel Moreno**, Jordi Madrenas**, Josep Maria Insenser*

* Semiconductores Investigación y Diseño, S.A. (SIDSA), c/ Isaac Newton, 28760 Tres Cantos, Madrid, Spain. Phone: (34) 1 803 5052, email: faura@sidsa.es
** Universidad Politècnica de Catalunya, c/ Gran Capitá s/n, Barcelona 08034, Spain, Barcelona, Spain Phone: (34) 3 401 67 42, email: moreno@eel.upc.es

ABSTRACT

We describe in this paper how VHDL greatly helps in modeling the dynamic reconfiguration of a novel Field Programmable Device (FPD) and the applications especially suitable for it. This dynamic reconfiguration methodology is based on multicontext operation, that is, having several copies of the configuration memory controlling the programmable features of the device. Configuration dynamic management is carried out by an on-chip microprocessor. We model all these interface operations - programmable hardware, configuration memory and internal microprocessor - with behavioral VHDL benches acting over the configuration signals on achitecturally described VHDL blocks. We also describe how a novel analysis and synthesis tool could be written to map general purpose applications on dynamically reconfigurable arrays like the one we present.

1. INTRODUCTION

Dynamic reconfiguration of Field Programmable Gate Arrays (FPGAs) has recently attracted the interest of the scientific community and has become an important topic for microelectronic research, especially for the sake of reusability. Beyond the *reconfigurable* FPGAs [1], those which can be reconfigured any number of times to carry out different functions depending on the required application, lie the *dynamically reconfigurable* ones which can change their functionality *on the fly*, that is, without having to stop the whole circuit while it is being reconfigured. Among these, some can be *partially* reconfigured so only the part of the array being reconfigured has to stop while being reprogrammed, and some can instantaneously change their configuration because they store two or more configuration contexts that can be swapped away in just a clock cycle. A good example of the partially reconfigurable known FPGAs is the ATMEL architecture [2], while no multicontext commercial devices has yet been introduced, being the only precedent the DPGA architecture developed at MIT [3].

Within this framework we introduce the FIPSOC (FIeld Programmable System On a Chip), a novel multicontext dynamically reconfigurable Field Programmable Device which includes an array of LUT-based digital configurable cells, a number of programmable analog blocks, and a microcontroller which manages the chip configuration and also runs general purpose user programs. The array of configurable logic blocks is in fact a FPGA with a built-in microprocessor interface: The on-chip microprocessor can access the configuration data, the Look Up Tables (LUTs) contents and the actual values of the array elements outputs because they are mapped onto the microprocessor address space. Furthermore, it can even write data onto the flip-flops (FFs). A more thorough description of this chip can be found in [4] and [5].

The key point to dynamic reconfiguration of the FPGA included in the chip is the fact that two configuration contexts have been implemented and can be read and written by the microprocessor. This means that once one of the contexts has been set as the *active* one, the other can be reprogrammed by the microprocessor without having to stop the chip operation. After loading the new configuration and intializing the sequential elements (FFs) of the non-active context, the *hardware swap* can take place in the time of a microprocessor write cycle by setting the non-active context to be active and vice-versa. Furthermore, a set of cells can be selected for reconfiguration, so *partial* reprogramming is also possible.

The benefits from this approach rely on the fact that typically not every part of an integrated circuit is working at a given time: If we can monitor which parts of the circuit are being used at each moment, we could use a dynamically reconfigurable FPGA to implement each active part at the required time, thus dramatically reducing the necessary area. It turns out that FIPSOC is a perfect platform for such a task: we have several configuration contexts so dynamic reconfiguration while in operation (of the whole array or just part of it) is granted. On the other hand, the on-chip microprocessor can control the context swap operation according to the required application at each time. Finally, non-critical tasks can also be trasferred to software, especially due to the

enhanced communication between hardware and software which makes the FIPSOC a powerful platform for HW/SW co-design (the microprocessor can probe at any time the outputs of the configurable elements as memory locations).

We have modeled the FPGA element using VHDL because this language is especially suitable to evaluate applications using the multicontext dynamic reconfiguration and because an activity analysis of the circuit (that is, which parts of the circuit are active at each time) could be done by tracing which VHDL *process* is active at each time (if the application had been specified in VHDL prior to technology mapping). Configuration management units are described in VHDL to control the configuration signals of the programmable blocks, so different scheduling algorithms can be tested before implementing them as microprocessor programs.

This paper is focussed on how this reconfiguration has been modelled with VHDL and how we use it to test applications suitable to be mapped on these dynamically reconfigurable blocks. We first review the FIPSOC chip architecture, especially these issues related to reconfiguration. Then we describe how the architecture has been modeled using VHDL, and how the dynamic reconfiguration management units are specified. Finally, we propose a new high level VHDL synthesis tool for dynamically reconfigurable targets, which would be especially suitable for this chip and that would constitute a very exciting research field.

2. FIPSOC Architecture Overview

2.1 Functional overview

As it has been said, a thorough description of the FIPSOC chip (fig. 1) can be found elsewhere [4], [5]. Just for reference let us mention that the chip is a mixed signal field programmable device with an on-chip microcontroller. It includes a Field Programmable Gate Array (FPGA), a set of fixed-functionality yet configurable analog cells, and a microprocessor core with RAM memory and some peripherals. The programmable digital and analog blocks are well defined and are separated from one another due to noise inmunity considerations. Nevertheless, the different interfaces between these blocks themselves and to the microprocessor provide a very powerful interaction between software, digital hardware and analog hardware.

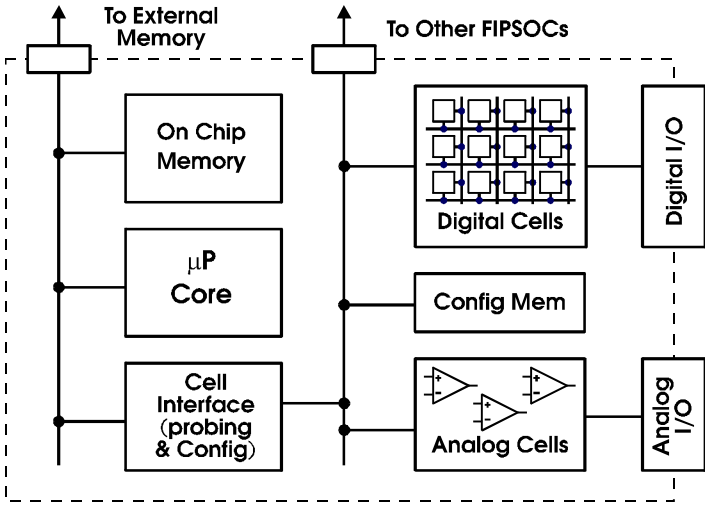

Figure 1: FIPSOC block diagram

The FPGA element (depicted in fig. 2) has been called Digital Macro Cell (DMC), and it is a 4-bit RAM LUT-based cell with four sequential elements (FFs) which can be configured in a number of ways: it can perform any four 4-input boolean functions, two 5-input functions or any 6-input boolean equation; it can be configured as an up/down 4-bit cascadable counter with load and enable, a 4-bit cascadable shift register with load and enable, a 4-bit cascadable adder, etc. The FFs in the sequential part of the DMC can be individually used and configured as mux-type or enable with synchronous or asynchronous set or reset, etc. A number of configuration bits are provided to select all these functions, and as it has been said, they are duplicated. The memory bits controlling the routing resources are also duplicated.

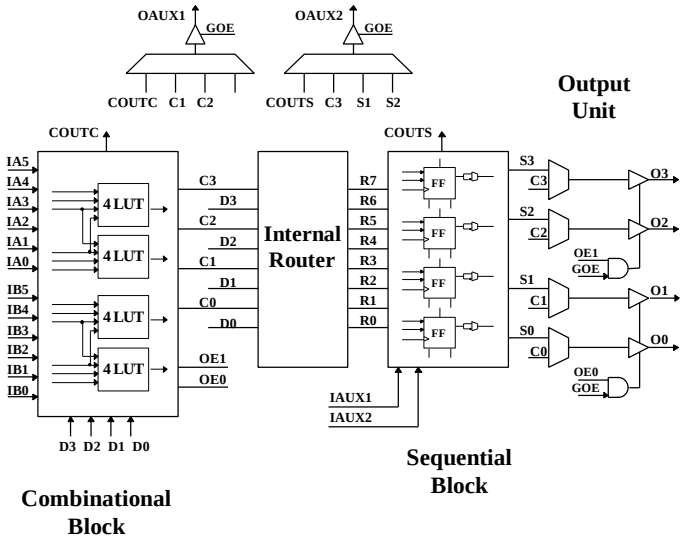

Figure 2: DMC block diagram

The interface between the microprocessor and the programmable hardware is very powerful: The outputs of the combinational and sequential parts of the DMC are mapped on the microprocessor memory space. This is equivalent to have a large number of parallel I/O ports distributed along the FPGA. Further reconfiguration decissions may be taken as a function of the real time actual values of these signals. This is why we believe that this chip is perfectly suitable for hardware-software codesign techniques.

2.2. Multicontext configuration memory implementation

As it has been already mentioned, the chip configuration is managed by the internal microprocessor, which can in turn be used as a dynamic reconfiguration engine. To do so, the configuration memory is organized in words which could be mapped onto the microprocessor address space. Especially due to noise inmunity considerations, the actual configuration bit is separated from the mapped memory through a NMOS switch, as depicted in figure 3. This switch can be used to load the information coming from the memory bus onto the configuration cell. The microprocessor can only read and write the *mapped* memory, and it could only transfer the information in one way.

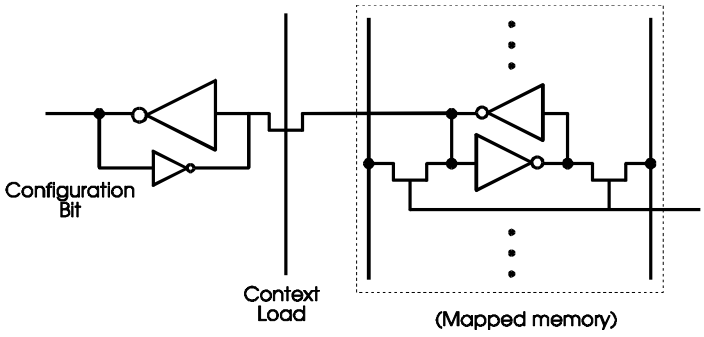

Figure 3: One *mapped* context and a *buffered* one

This technique entails several benefits:The mapped memory can be used to run programs and to store general purpose RAM data, and the configuration bit is isolated from the mapped memory cell (thus being safely isolated). The microprocessor can then use to mapped memory to load the configuration bits, then release it for a different use. This implementation is said to have one *mapped* context (the one mapped on the microprocessor

memory space) and one *buffered* context (the *actual* configuration memory which directly drives the configuration signals).

There is also the possibility of having more than one mapped contexts to be transferred to the buffered context, like depicted in Figure 4. However, as the number of mapped contexts increase, the efficiency of the proposed solution could decrease due to the bigger decoders needed to drive so much memory. The size of the DMC would of course increase, but more user memory would be available to the user.

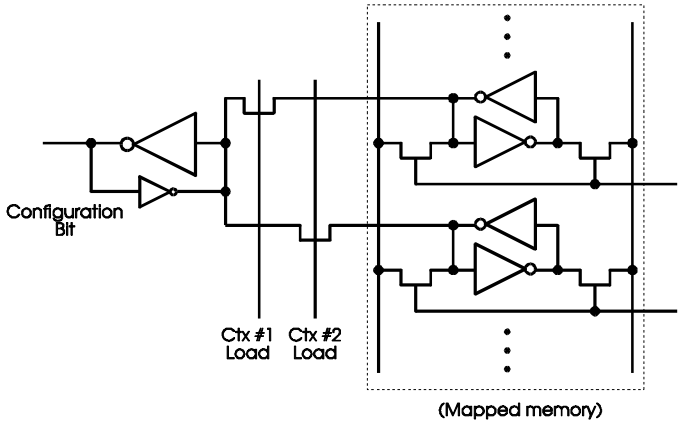

Figure 4: Two *mapped* contexts and a *buffered* one

A final possibility to increase the number of contexts would be to stack them, as depicted in Figure 5.

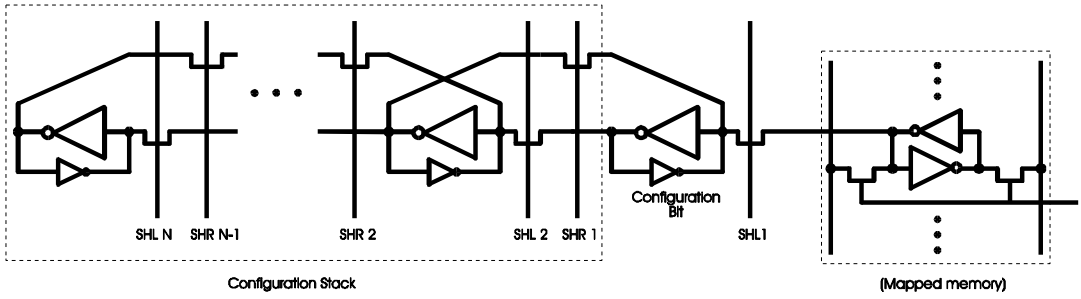

Figure 5: A configuration stack structure

This technique consists of providing a greater number of contexts, maybe four or eight, but only having access to one of them. Data can be transferred both ways to effectively *push* and *pop* configurations onto the *configuration stack*. This way, a given application would have different smaller subsystems which could be treated as *hardware subroutines* [6]. These subroutines could be hierarchical and call more sub-subroutines, and so on. Everytime a subroutine is released, the actual DMC space upon which it was working would recover the task it was doing at a higher hierarchical level. Of course, the efficiency of this technique would greatly depend on the application and the detailed scheduling of hardware tasks. A further detailed study of this solution would be necessary to assure its feasibility, although its implementation in VHDL can be accomplished from a theoretical point of view.

2.3. Feasibility and economics

To study the feasibility of the proposed architectures, two full custom implementations have been done, one as depicted in Fig. 3.A and another one like in Fig. 3.B. The DMCs included in the first silicon

implementation have two mapped contexts and a buffered one like depicted in Fig. 3.B. Table 1 shows an area breakdown of the DMC implemented in silicon:

Function	Area percentage
Routing switches (NMOS multiplexers)	34
Buffered context	9
Mapped context	14
Extra mapped context	9
Four 4-LUTs	20
FFs and their control	4
Output buffers, microprocessor interface logic, etc	3
Internal interconnections and stray area	7

Table 1: Area breakdown of a multicontext DMC

As it can be seen from the table, an extra context is not prohibitive. The area difference between the mapped and the extra mapped context comes from the decoder and its interface to the microprocessor bus - the same decoder is used for both contexts, while it is counted only once in the breakdown. The buffered context is bigger than the mapped one because it uses a larger gate width to be 5V compatible (this is interesting since the routing resources extensively use NMOS multiplexers), and because of noise inmunity considerations. This buffered context should not be completely regarded as an "extra" option to configuration: It provides two polarity signals to the NMOS multiplexers, which would be complicated to route from the mapped memory blocks (which is evenly organized). It is also important to note that the mapped context, after the configuration load, can be freely used for general purpose use. The memory is distributed along the DMCs, but the microprocessor can use it normally to run programs and to store user data. For a 12x12 DMC array, around 16 Kbytes of RAM memory is provided to the user. The FFs are also internally duplicated to support the dynamic reconfiguration feature. The area percentage shown in the table accounts for two-context FFs.

Finally, the area required by the LUTs includes the memory cells (which store the configuration), their connections and their interface. As far as this amount of area is somewhat large, LUTs are in fact internally multiplexed rather than duplicated for dynamic reconfiguration purposes. This means that LUTs are twice smaller when used in dynamic reconfiguration mode, but they are more flexible because they share no inputs - unlike in the *static* modes in which they share two out of their four inputs.

Multicontext dynamic reconfiguration takes place then like follows: The microprocessor loads the initial configuration on the mapped memory, then transferring it to the *actual* configuration memory with a single memory write cycle. When a new configuration is required, it is loaded on the mapped memory again. It is important to note that the chip can work normally while this configuration is being loaded, because the last configuration still applies. Then, a single write command can load the new configuration, minimizing the device stop time. If two mapped contexts are available, two configurations can be loaded and interchanged without having to rewrite the mapped memory every time prior to reconfiguration. Hoever, this is true if only two configurations are needed.

3. VHDL Model of the Programmable Digital Hardware

The VHDL model of the DMC is structural on the top and behavioural in the bottom. This means that the architecture of the DMC can be tracked down thru several meaningful architectural levels. For example, combinational and sequential parts are well defined and separated; the LUTs are used as *components* of the combinational parts but only a behavioral description with no further hierarchy has been created for the LUT. The same applies for the sequential part, being composed of *components* down the hierarchy until a D-FF is reached, whose behavior is described.

All thru the hierarchy, a configuration record is used. This is a complex record made out of simpler records. For example, the top configuration record has a slice dedicated to the combinational part. This slice is itself a record which has a string to configure one of the LUTs. That string is passed down to the LUT behavioral description, which actually uses it. The top configuration record type and all the sub-record types are defined in a global package used by all levels of the hierarchy. They are defined as actual PORTs in the entities instead of

being GENERIC constants because they can dynamically change as long as our FPGA is dynamically reconfigurable.

DMC arrays are iterated in bidimensional GENERATE loops. The array interface is behaviorally described and configured for each array size and organisation. A software model of the microprocessor would be connected to this block, exactly as the configuration management engines, to be described in next section, are. Interface issues such as actual signals values look-up and configuration signals driving are done thru this block to the rest of the array.

4. VHDL Modeling of the Reconfiguration Process

This approach has proved to be quite convinient for multicontext dynamic reconfiguration experiments: The configuration context management is a separate VHDL engine which deals with the configuration signals which are then connected to the instances of the DMC VHDL description. Each application then encompasses a set of DMC instances interfaced thru a configured interface block, a routing resource block (which dynamically changes the connectivity between the routing channels which are its inputs and outputs depending on a configuration signal) and a configuration management engine which dynamically generates the configuration signals for the DMCs and the routing block. Different applications may only vary in their configuration management engines, using the same background DMC array and routing block.

Three different types of dynamic reconfiguration management engines have been implemented for this experiments, corresponding to the three multicontext configuration bit implementations described before: A single context one with a context load feature which allows an inmediate reconfiguration with a single microprocessor command while in operation, a two context engine which allows context swaps between them both without having to rewrite the non-active context each time, and a generic context FIFO of a generic length suitable for multi-level hardware procedures.

The last one would be the most interesting from the VHDL tools developer point of view, though it could be found impractical from a design engineer point of view. The main advantage of this structure is its support for multi-level hardware procedures [6]: Treating hardware blocks as we treat software procedures, and considering the DMC array as the memory space, one could dynamically allocate a hardware task to a given position on the DMC array. This hardware task may use the dynamic reconfiguration as well because it may have different tasks to be carried out at different moments so it would be no need to have stopped silicon areas due to sequential processing. Then, this hardware procedure would spawn hardware subroutines which could also spawn new ones. When a subroutine is allocated, the context FIFO is shifted. When the subroutine finishes, the FIFO is shifted back so the previous configuration is restored. This way, a very large and hierarchically deep and complex virtual circuit could be implemented to carry out the function of a circuit orders of magnitude larger in area, although the configuration cells in this case would also impact the circuit area as well.

It is worth noting that in all three cases the FFs are duplicated and the microprocessor can write the active or the non-active one every time (in the context FIFO case there are as many non-active FFs as contexts). This is very useful for two purposes: first, it lets the user to initialize the FFs prior to context load; second, it enhances communication between contexts and software. When the hardware swap takes place, the FF can be selected to swap its value with the non-active one or to remain the same. Following the analogy with procedures, a FF which mantains its data after reconfiguration would be likened to a global variable, while a FF which stores its data with the context (thus being able to be restored back) would be more like a procedure parameter or local variable. In the case of the context FIFO, FFs are freezed when a new subroutine is allocated over them, and their values are restored back after the subroutine has ended. The outputs of a subroutine are communicated thru FFs whose values are maintained upon context swap.

These engines have been implemented in VHDL and have been used for several applications, obtaining significant area reductions (i.e., number of used DMCs) compared to non-dynamically reconfigurable FPGA solutions. As a drawback, the more complex reonfiguration the more intensive processing is required from the microprocessor, which becomes then apparently slower when running user programs.

Depending on the application, a different multicontext architecture proves to be the optimum, though the first two architectures (the ones effectively implemented in full custom) are often the most economic. Optimality of each architecture for each type of applications has been found after area measurements of the DMC cell (silicon area of the full custom implementation of each DMC type) and the number of DMC instances in each VHDL description of each application. According to the economic study outlined before, an extra mapped context, using the same microprocessor interface, takes around 10% of the DMC area, while it provides extra memory for the microprocessor and extra functionality due to the reconfiguration capability. In most of the cases this slight area

overhead is widely justified because the DMC count reduces to 60% or even less, and at the same time more free memory is available from those DMCs which can not be reconfigured because their function has to be kept working.

5. Future Work: Automatic Synthesis for Dynamically Reconfigurable Arrays

Finally, it is worth talking about the future work after having proved that multicontext dynamic reconfiguration is useful and can be implemented: A software tool could be created to analyze a VHDL description (the application itself) composed of processes. It would determine which process is active at each moment and would also distribute between hardware and software the different tasks. It would generate then a microprocessor program which would dynamically reconfigure the FIPSOC FPGA areas according to the needs at each moment, keeping the rest of the time running the software tasks and monitoring the hardware/software interaction (writing data onto the FFs and reading from the DMC outputs). Such a tool could succesfully distribute large applications in several non-simultaneous hardware contexts and software, and everything could be mapped on a single FIPSOC chip, saving silicon area.

A very interesting related tool designed for partially reconfigurable arrays has already been reported [7]. The improvements to dynamic reconfiguration introduced by FIPSOC, together with its unique ability to handle hardware/software interaction, makes this device suitable for a wide range of new synthesis methodologies for reconfigurable targets and hardware/software co-design.

6. Conclusions

VHDL, used here not for design nor synthesis but for modeling purposes, has been used to describe and study the dynamic reconfiguration operation of a novel multicontext dynamically reconfigurable field programmable device. Different reconfiguration strategies have been tested using VHDL, a language whose flexibility has been a key point for this task. It has been found that the lenguage may play a fundamental role in the design methodology to be followed for this new device and the ones of its kind. The feasibility of our approach has been successfully tested using VHDL models of the programmable hardware and applications mapped over them.

7. Acknowledgements

The authors would like to thank the European Community for the economic support. This work has been carried out within the framework of ESPRIT Project #21625 (FIPSOC).

8. References

[1] Doug Conner, "Reconfigurable Logic: Hardware Speed with Software Flexibility", EDN Europe, July 1996 issue, pag 15.

[2] Configurable Logic Design & Application Book, © ATMEL corporation, 1994.

[3] E. Tau, D. Chen, I. Eslick, J. Brown and A. DeHon, "A First Generation DPGA Implementation", FPD'95 -- Third Canadian Workshop of Field-Programmable Devices, 1995 Montreal, Canada.

[4] Julio Faura, Paul Naish, Paul Paddan, Bernd Krah, Phuoc van Duong, Juan Manuel Moreno, Antonio Torralba, José Laúna, and Josep Maria Insenser, "FIPSOC: A New Concept to Mixed Signal Integration", The Silicon Design Show 1996, Birmingham, UK.

[5] Julio Faura, Chris Horton, Phuoc van Duong, Jordi Madrenas, Miguel Angel Aguirre, and Josep Maria Insenser, "A Novel Mixed Signal Programmable Device with On-Chip Microprocessor", to be published in the Custom Integrated Circuits Conference (CICC'97), May 1997, Santa Clara (CA), USA.

[6] Neil Hastie and Richard Cliff, "The implementation of hardware subroutines on field programmable gate arrays", IEEE 1990 CICC.

[7] Patrick Lysaght and Jon Stockwood, "A Simulation Tool for Dynamically Reconfigurable Field Programmable Gate Arrays", IEEE transactions on VLSI systems, vol. 4, nr. 3, September 1996.