05-Lallet-v4n1-AF 19.08.09 19:11 Page 36

Efficient and Flexible Dynamic Reconfiguration for Multi Context Architectures

Julien Lallet, Sébastien Pillement, Olivier Sentieys

INRIA Rennes - Bretagne Atlantique, Université de Rennes, F-22300 Lannion
e-mail: lallet@irisa.fr

ABSTRACT

Dynamic reconfiguration is possible on both fine-grain and coarse-grain architectures. One of the used methodology used consists in the use of multi-context architectures. Unfortunately, the multiple contexts bring power and area overhead. This paper introduces the Dynamic Unifier and reConfigurable blocK (DUCK) concept, a new structure to perform efficiently dynamic reconfiguration on both custom designed fine-grain and coarse grain architectures. The DUCK allows to separate the configuration path and the configuration registers which facilitates simultaneous configuration and computing steps. The reconfiguration process is presented in detail, and synthesis results are given for different structures. Our solution is finally validated with the implementation of a WCDMA (*Wideband Code Division Multiple Access*) receiver on a multi-context embedded FPGA and on the dynamically reconfigurable processor DART. This implementation demonstrates the interest and the efficiency of the use of dynamic reconfiguration and the proposed flexible structure.

Index Terms: Dynamically reconfigurable architectures, Multi-context.

1. INTRODUCTION

Systems on Chip (SOC) are generally based on three main kinds of architecture. First, Application-Specific Integrated Circuits (ASIC) allow to efficiently compute an algorithm due to dedicated hardware but are unfortunately inflexible. Secondly, Generic Purpose Processors (GPP) are the most flexible architectures, but compute in an inefficient way. Finally, static reconfigurable architectures such as Field-Programmable Gate Array (FPGA) are considered as a good compromise between processors and ASIC. Meanwhile, mixed architectures have been developed in order to improve the efficiency and the performance of GPP by the use of static reconfigurable co-processors. These static reconfigurable co-processors embedded into an SoC, namely embedded-FPGA (e-FPGA), have allowed GPP to follow application developments. Dynamic reconfiguration allows partial configurations at run-time, and thus improves performances. Some specific processors and FPGAs take advantage of dynamic reconfiguration in their architecture by the use of the multi-context process. This architectures are either Dynamically Reconfigurable Processors (DRP) or Dynamically Reconfigurable FPGAs. This is achieved by the local storage of any possible context. When a new configuration is required, the system switches between one or the other context. The major drawback of this solution is the silicon area and power inefficiency caused by local memories needed to store all the contexts. Our contribution to multi-context DRA is the definition of a flexible and optimized structure that supports dynamic, partial and run-time reconfiguration dedicated to both finegrain and coarse-grain DRA structures. This is performed by only two configuration memories, one current configuration memory and one parallel configuration memory. The parallel configuration memory is used for loading or saving contexts for preemption to or from the configuration memory in one clock cycle. New contexts are stored in this parallel configuration memory thanks to a splittable scanchain. Compared to previous multi-context Dynamically Reconfigurable Architectures (DRA), configurations exploit efficiently the available silicon resources and enables the implementation of any kind of computing granularity.

The paper is structured as follows. Section 2 describes related works on optimization of multi-context DRA. Section 3 presents our contribution on dynamic reconfiguration processes of DRA. In Section 4, we present the experimental method and discuss results on a WCDMA receiver implementation on an e-FPGA and on the dynamically reconfigurable processor DART. Finally, Section 5 sums up this paper.

05-Lallet-v4n1-AF 19.08.09 19:11 Page 37

2. RELATED WORKS

For a decade, many dynamically reconfigurable architectures have been developed but only a few can be considered as multi-context architectures, as they locally store one or more parallel contexts. NEC-DRP [7] is a massively parallel processor architecture. Reconfiguration processes are managed by a central manager which can select one futur context from 16 multi-contextmemories. For the architecture XPP [13] (*eXtreme Processing Platform*), dynamic reconfiguration is managed by a hierarchical manager composed of a tree of sub-managers. A cache memory placed in parallel of the configuration memory enables multi-context reconfiguration. The Adres architecture [14] (*Architecture for Dynamically Reconfigurable Embedded Systems*) stores its configuration either in a RAM configuration memory or in the hierarchical memories for bigger configuration spaces. The main constraint of these architectures comes on one hand from the few flexibility offered by these architectures in terms of computing granularity, and, on the other hand, from the fact that this architectures do not to achieve an efficient reconfiguration from their parallel memory.

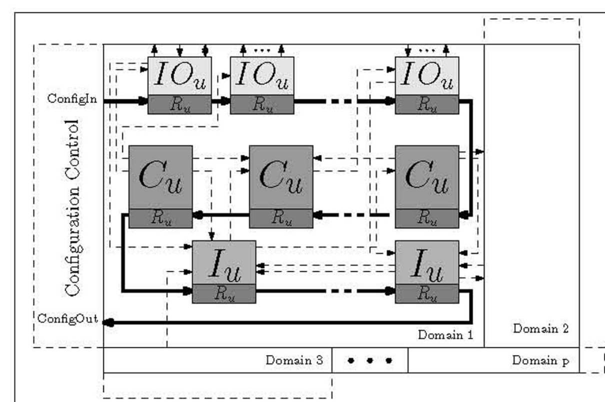
The only fine-grain architectures which implement dynamically reconfigurable computing are multi-context FPGAs. Commercial FPGAs (e.g. Xilinx Virtex family) allow dynamic reconfiguration, but the reconfigured resources have to be stopped before a new configuration can be propagated [2]. Different approaches have been proposed in the literature to reduce the excessive silicon area used by multicontext FPGAs. First, some works focus on the reduction of the configuration words. In [6], the method consists in the limitation of the connection map inside a switch box. In [1], the authors reduce the context memory by using redundancy and regularity in the configuration data. The first method has the disadvantage to reduce routability. The second is efficient only in good conditions of redundancy and regularity, which is not the case for all applications. The second approach [3] is a technological solution which consists in the use of DRAM memories instead of SRAM usually implemented for storing configuration contexts. This allows to save between 10% and 60% transistors, but causes a new problem concerning mixed process of DRAM and logic.

3. EFFICIENT DYNAMIC RECONFIGURATION

In this section, we present the resource that is proposed to make the reconfiguration efficient and flexible, whatever to the granularity of the computing resource.

A. DUCK: Dynamic Unifier and reConfiguration blocK

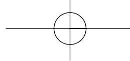
As mentioned in Section 2, multi-context reconfiguration has provided solutions for fast reconfiguration but generates redundant resources (local context memories) which contributes to a power inefficient design even if some solutions have been developed. However, the solution that we propose in this paper needs only one context memory for each resource, independently of the granularity of the computing resource. But, using only one context memory means that it is necessary to develop other architectural concepts in order to maintain the timing constraints and the flexibility required by today's applications. The first concept of our contribution consists in the isolation of the configuration paths and the configuration resources which allows to prepare new contexts during the computation. The Dynamic Unifier and reConfiguration blocK (DUCK) is in charge of the configuration path and has to swap the required contexts to the configuration registers when needed. The second concept consists in the possibility to split the configuration path while maintaining a unique computing path in order to propagate the configuration through several configuration paths at the same time. Each configuration path composes a reconfiguration domain. Figure1 shows an example of the implementation of the DUCK concept. This basic example implements interconnection units (I_u), computing units (C_u), input and ouput units (IO_u) and reconfiguration units (R_u) namely the DUCK resources. For each units (I_u, C_u or IO_u) one DUCK (R_u) is associated and composes the configuration path (black arrow in bold print). The inputs of the two configurations paths are represented by the name *ConfigIn(i)* and the outputs by the name *ConfigOut(i)*. When the system is ready to reconfigure, each DUCK swaps the configuration context from its internal registers to the control registers of each unit. Once the configuration is swapped, it is possible to extract the context through the configuration path.



Configuration ScanPath
Computing DataPath
R_u: Reconfiguration Unit
IO_u: Input/Output Unit
C_u: Computing unit
I_u: Interconnection Unit

Figure 1. Example of a DRA composed of several kind of resources

05-Lallet-v4n1-AF 19.08.09 19:11 Page 38



B. Dynamically Reconfigurable Architectures

According to the resource to reconfigure, the DUCK implement different functions. In our DRA model, the communication resources, called DyRIBox (Dynamically Reconfigurable Interconnection Box), switch signals from input ports to output ports. Each DyRIBox has n_i inputs and m_i outputs on b bits at each i of its four sides (North, South, West, East). The total number of input and output ports is therefore $N = \Sigma_{i=0}^{3} n_i$ and $M = \Sigma_{i=0}^{3} m_i$ respectively. Depending on the value of the configuration register, each input can be connected to one or several outputs. To reduce the complexity and the size of the configuration stream of the DyRIBox, the number of inputs that can be switched to an output is set to P, with $P \leq N$. Therefore, the DyRIBox contains M configuration registers of $[p = \log_2 P]$ bits.

In case of classical dynamic reconfiguration, the reconfiguration time is too long for the given timing constraints. To reduce this time, the reconfiguration process of the DyRIBox and of the computing resources is based on DUCK context registers (Figure2). Each configuration register is connected to one context register contained in the DUCK resource and data could be swapped when needed.

In order to manage the reconfiguration process, all DUCK registers are interconnected through a scanpath bus. Scanpath registers are used in design-for-test (DFT) techniques instead of classical registers in order to extract the register value at any time. The scanpath bus creates a unique big shift register with all the scanpath registers of the architecture. Thus, the extracted data flow is compared with the test vectors during testing to detect errors in the computing path. This method has been cited in [4] for applying preemption in reconfigurable architectures but has not yet been implemented. This was due to the fact that the extraction time was too long for the given timing constraints required by today's applications.

The use of the configuration path in a scanpath manner associated with the DUCK concept allows the system to be reconfigured in one clock cycle. The use of the DUCK registers allows the system to prepare the next configuration while it is computing. The propagation of a new context is done by three different steps. First, the configuration registers are already loaded with the current context (Figure2(a)). The DUCK registers are waiting for the next step. Either the new configuration is already propagated, or is waiting to be configured. The second step (Figure2 (b)) shows how a new configuration is spread to the DUCK registers. As explained before, the DUCK is connected in a scanpath manner which allows to propagate the next context. In case of preemption, the process is still the same for extraction of the previous context. Finally, each DUCK register swaps its data with the configuration register. Every configuration register is directly connected to a DUCK register. It is noteworthy that in case of a new configuration identical to the current one, the configuration swap does not disrupt the interconnection and computing resource behavior. Therefore, reconfiguration is possible even if a computing datapath crosses a reconfiguration area which it does not belong to.

Today's SOCs use very different kinds of computing resources, so that, for every new dynamically reconfigurable architecture or computing resources, it becomes more difficult to extract an homogeneous reconfiguration protocol. The DUCK aims to solve this issue. For example, considering a classical logic cell (gray area on Figure3) several resources are part of it. The reconfiguration path (area Ⓐ) allows to set or reset the output register, to select the sequential or combinatorial output, and to select the carry input. The memory area (Ⓑ) allows to use the logic cell as a RAM memory. The carry resources are needed for arithmetic operations (area Ⓒ). The LUT resources are needed for the implementation of logical functions (area Ⓓ).

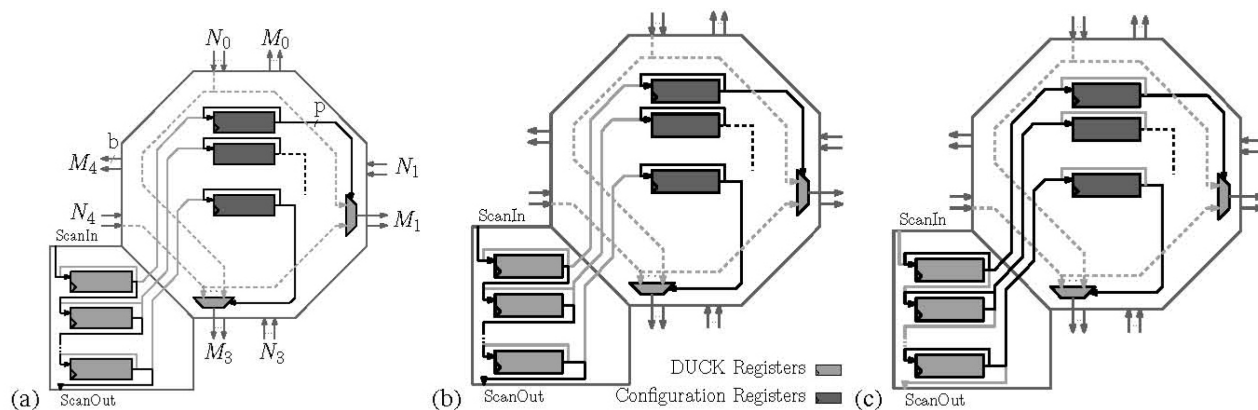
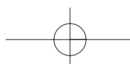


Figure 2. Reconfiguration process inside the DUCK structure



05-Lallet-v4n1-AF 19.08.09 19:11 Page 39

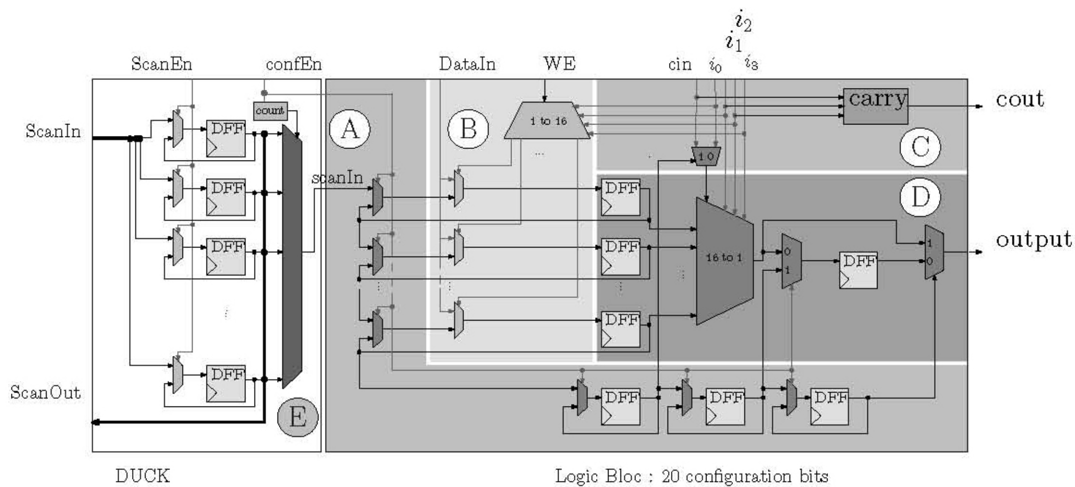


Figure 3. Simple logic cell architecture developed for fast reconfiguration

The configuration path goes through all configuration registers and LUT registers. In this example, one logic cell needs 20 clock cycles to be reconfigured. Thus, for an e- FPGA composed of a $n{\cdot}m$ array of logic cells, $n{\cdot}m{\cdot}20$ clock cycles are needed to reconfigure the whole FPGA. This time is not acceptable for fast reconfiguration. Our solution, the DUCK (area Ⓔ of Figure3) allows to shift the configuration context locally in the same way as for the DyRIBox and to swap the reconfiguration when needed. Therefore, the whole embedded FPGA can be reconfigured in 20 clock cycles. In the DUCK, a counter selects each configuration register one after the other and shifts it to the logic cell configuration path.

C. Results and Exploration

We present here exploration results on the DUCK parameters. First, synthesis results are given to estimate the impact on silicon area of the size of outputs and inputs, and the number of possible connections to one output. The critical path and power consumption are also analyzed. All results are expressed as a function of the computing data bitwidth. Results are obtained with the synthesis tool Design Compiler from Synopsys and for a 130nm CMOS technology.

The influence of DUCK parameters on design area, power and critical path is given in Figure4. The results have been obtained by changing the number of outputs on the DyRIBox. First, the DUCK has clearly no influence on the critical path results because of the physical separation of the configuration path and the configuration registers in the DyRIBox. Secondly, the more connection possibility the DyRIBox has, the less impact the DUCK has in the design area. This is explained by the fact that the silicon area used for the interconnection wires between inputs and outputs grows faster than the silicon area used by the configuration/DUCK memories. Due to custom libraries used for 8-bit words, the power consumption is better controlled from this bitwidth than for 4-bit data.

The interconnection network presented in [9] consists in a set of reconfigurable circuit-switched routers interconnected by links. One router is composed of five 16-bit bidirectional ports connected through a 16x20 fully connected crossbar. We have generated and synthesized a DyRIBox associated to a DUCK with the same functionality. The results are given for the 130 nm CMOS technology from ST Microelectronics. Area, frequency and power after synthesis are given for the two solutions in Table I. These results show that the simplicity of our solution allows to keep as many flexibility as in their solution, whereas our structure has only 4% area overhead and a gain of 25% on the critical path and of 69% in power.

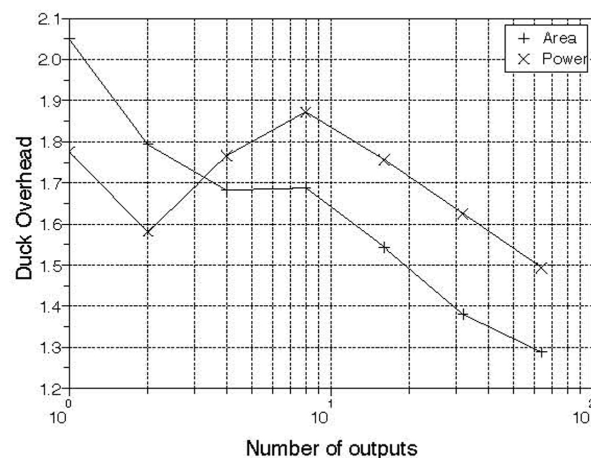


Figure 4. Influence of DUCK on area, power and time

Table I. Synthesis results compared with the 4S projects solution

Interconnection	Area in mm^2	Critical Path in *ns*	Power in *mW*
4S project	0.0506	930	17.32
DyRIBox	0.0526	692	7.22

In conclusion, fast dynamic reconfiguration is made possible by the use of the DUCK concept: the separation of configuration path and the configuration registers. A small overhead of silicon area for each logic cell and interconnect box is involved by our method, but on the other hand, the reconfiguration itself allows to save resources compared to multi-context DRAs. Furthermore, to maintain the timing constraint, it is necessary to propagate each new context as fast as possible, so that the new tasks can swap in the most efficient way. That is realized by the introduction of the split configuration path. Indeed, when several configuration paths are created, it is possible to propagate new contexts in parallel with each configuration path. This method allows to reduce the propagation time with regard to the number of configuration path used. The following case study gives more precise results about saved resources on a telecommunication application.

4. CASE-STUDY

In this section, we present the implementation of a *Wideband Code Division Multiple Access* WCDMA receiver on our embedded FPGA (Figure 6) and on the dynamically reconfigurable processor DART. WCDMA is a high-speed transmission protocol used in third generation mobile communication systems such as UMTS (Universal Mobile Telecommunications System), and is considered as one of the most critical applications of third-generation telecommunication systems. It is based on the CDMA access technique where all data sent within a channel and for a user to have to be coded with a specific code to be distinguished from the data transmitted in other channels [8]. The number of codes is limited and depends on the total capacity of the cell, which is the area covered by a single base station. To be compliant with the UMTS radio interface specification (UTRA – Universal Terrestrial Radio Access), each channel must achieve a data rate of at least 128kbps. The theoretical total number of concurrent channels is 128 channels. As in practice only about 60% of the channels are used for user data, the WCDMA base-station can support 76 users per carrier.

TheWCDMA application executed on our reconfigurable architecture consists in the alternate execution of three main tasks (Figure5): *FIR (Finite Impulse Response) filter*, *Searcher*, and *Rake Receiver*. Within a WCDMA receiver, real and imaginary parts of the signal received on the antenna after demodulation and digital-to-analog conversion, $S_r(n)$, are filtered by an FIR (Finite Impulse Response) *shaping filters*. Since the transmitted signal reflects in obstacles like buildings or trees, the receiver gets several replicas of the same signal with different delays and phases. By combining the different paths, the decision quality is

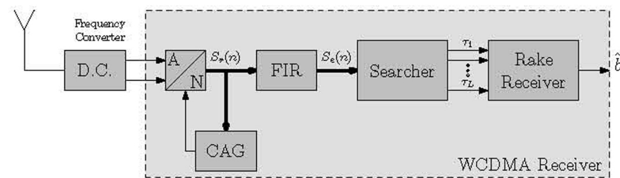


Figure 5. WCDMA receiver synoptic

drastically improved. Consequently, the *Rake Receiver* combines the different paths extracted by the *Searcher* block in order to improve the quality of the symbol decision. Each path is computed by one *finger* which correlates the received signal by a spreading code aligned with the delay of the multipath signal. In our case, a maximum number of fingers are considered. This task is realized at the chip rate of 3.84 MHz. The decision is finally done on the combination of all these spreaded paths.

A. Timing Constraints

WCDMA is the highest speed transmission protocol used in the UMTS system. The bandwidth of the transmitted signal is equal to 5 MHz. The frequency of the code corresponding to the chip rate (F_{chip}) is fixed to 3.84 MHz. One slot is composed of 256 chip data. Registers are used to pipeline data while *FIR*, *Searcher* or *Rake Receiver* are computing in one slot. For better synchronization results, the received chip is 4-time over-sampled. The computing time available for the three functions (*FIR*, *Searcher*, and *Rake Receiver*) is therefore t_{slot} = 66.6_s between the computation of two consecutive slots. The *FIR* and *Searcher* computes on 1024 samples while one *Finger* of the *Rake Receiver* computes on 256 samples. One sample is computed at each clock cycle.

B. e-FPGA implementation

The implementation of the WCDMA receiver on an hardware accelerator composed from standard logic cells as such implemented in a FPGA architecture is presented. The interaction of a DUCK and a logic cells allows the architecture to reconfigure the whole resources in parallel.

Table II presents synthesis results obtained with the VPR [15] and ABC Berkeley [16] frameworks. The most complex function, the searcher, requires 4953 logic cells to be configured in the e-FPGA. It is therefore possible to implement the whole WCDMA decoder into 4953 logic cells using dynamic reconfiguration. To illustrate dynamic reconfiguration, the three functions are executed sequentially in a time slot of 66.6µs i.e. 22.2µs for each function. Therefore, each function is executed during 22.2µs while the next context is propagated. As said previously, each function is completed in 1024 clock cycles, and the clock frequency is therefore

05-Lallet-v4n1-AF 19.08.09 19:11 Page 41

Table II. Necessary logic-cells for WCDMA decoder implementation on a dynmicall reconfigurable architecture

	FIR	Searcher	Rake Receiver	
			a Finger	All
Logic cells	3475	4953	561	4488
Total	12916			

greater than 46.55 MHz. The logic cell critical path has a value of 0.6ns in a 130nm CMOS technology. Considering that the functions have a critical path of 13 logic cells, the computing frequency can be up to 128.2 MHz. For a better power consumption, the frequency can be reduced to a lower value maintaining the timing constraint. For this implementation, the computing frequency is set to 50 MHz ($t_{computing}$ = 20.48µs).

To perform dynamic reconfiguration, 4953 logic cells need to be reconfigured in less than 22.2µs. One logic cell has 20 reconfiguration bits and a DyRIBox 10 bits. A 6-bit width configuration path is used for its good trade-off between performance and silicon area. Therefore, 4953 x 30/6 = 24765 6-bit words are needed for each context.

Thanks to our system architecture, the global configuration is split into 8 reconfiguration domains managed in parallel. Using a 300MHz clock frequency for the reconfiguration process allows to reconfigure in less than 11µs.

Figure 6 shows the implemented architecture with 8 domains of 620 logic cells. Static memory is used to allow data exchange between each functions. Light-gray areas represent the 8 configuration paths composed of 620 logic cells each. Each WCDMA function can be implemented on this architecture. The *FIR* function is depicted as task T1. Its implementation requires all domains and thus designs a unique computing path. The *Searcher* function requires also the 8 domains and thus designs also a unique computing path. The last function, *Rake Receiver*, can be split on 8 computing paths. One computing path for one *finger*. Assuming that a *Finger* implementation requires 561 logic cells, one domain is used for each *finger*. The 59 remaining logic cells are used to realize the decision on symbol.

Figure 7 shows that the process of propagation, computing and reconfiguration is fast enough to maintain the timing constraints thanks to the DUCK resources in the DyRIBox and the logic cell. On one *slot time*, the DUCK resources are able to extract the previous context or propagate the future context. $Pree_{RFS}$ means preemption of the *Rake receiver* or *FIR* or *Searcher* contexts and $Conf_{RFS}$ means configuration of the *Rake receiver* or *FIR* or *Searcher* contexts. The *NOP* operation means that the DUCK resources are waiting for working. *Domain 0* and *Domain 1* are giving an example of a complete WCDMA computing implementation including *Finger* implementation. *Domain 7* gives an example where no *Finger* needs to be implemented. The computing time ($t_{computing}$) represents the available computing time of one function, the propagation time ($t_{propagation}$) represents the available time for the configuration and the preemption processes, and the reconfiguration time (t_r) represents the time needed to reconfiguration the whole domain. The synthesis results the silicon overhead of the added local configuration memories. The overhead silicon area of the DUCK resource is 998µm² for a DyRIBox and 1468µm² for a logic cell. Considering that 4960 of the two resources are implemented, the overall area overhead can be estimated at 12.23mm². It is important to notice that 12926 logic cells should have been used for a static implementation. Our implementation using dynamic reconfiguration consumes 7966 logic cells less than the static implementation. Considering that the silicon area needed for a logic cell is 2160µm² and 6850µm² for a DyRIBox, we can estimate the saved area to 59mm². Thanks to partial reconfiguration offered by today's Xilinx FPGAs, it could be possible to implement a WCDMA decoder on two areas of 4960 logic cells. This solution requires 4960 logic cells more than our dynamic implementation.

Finally, Table III compares the same WCDMA decoder implemented in a Xilinx Virtex FPGA. It can

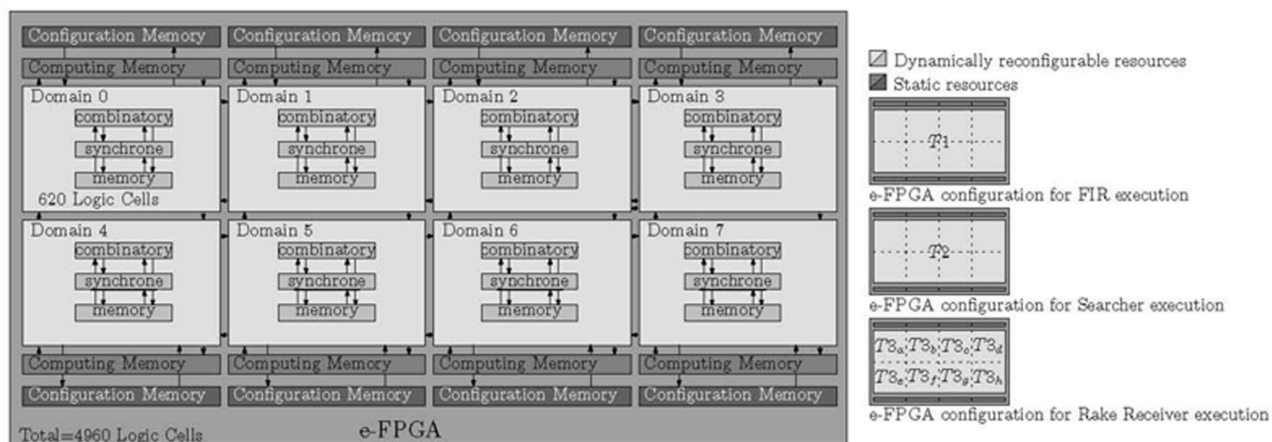
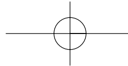


Figure 6. Resource allocation of the implemented embedded FPGA

05-Lallet-v4n1-AF 19.08.09 19:11 Page 42



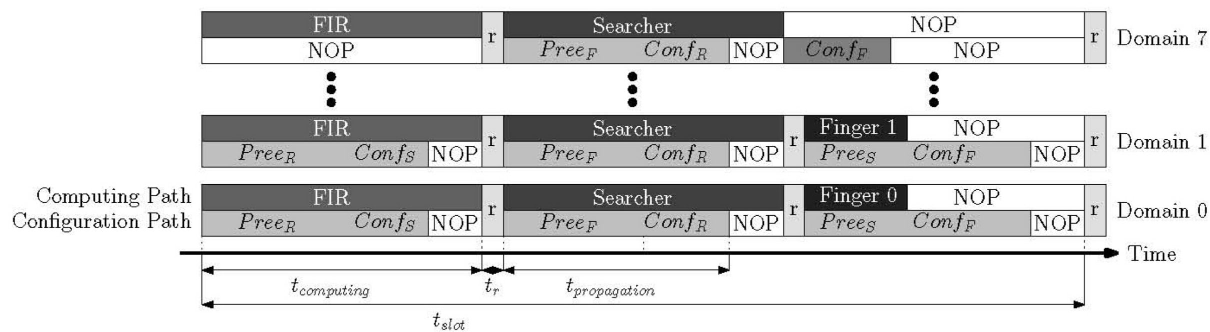


Figure 7. Gantt diagram of computing and reconfiguration process

Table III. Comparison between results on an enbedded FPGA solution and on a Virtex commercial FPGA

System	Logic Cells	Configuration Size (8-bit word)	Reconfiguration Time
e-FPGA	4960	36k	22.2µs
XCV200	5292	164k	2.53ms

be easily concluded that a dynamic reconfiguration is not possible on the Virtex since the reconfiguration of the entire FPGA takes more than 2ms [12] with a configuration frequency of 60MHz and with the SelectMAP interface which enable 8-bit word configuration.

C. DART implementation

In this section, we present the implementation of the WCDMA receiver on a hardware accelerator composed from the computing resources of the DART architecture.

The dynamically reconfigurable processor DART [11] is a coarse-grain reconfigurable architecture developed mainly for 3G mobile telecommunication application domain. DART architecture is build around six computing elements called DPR (*DataPath Reconfigurable* Figure 8). Each DPR is composed of two registers (reg), four Address Generators (AG) to access four local memories (Datamem), and four FUs (*functional units* two adder/subtracters and two multipliers). The DPR is fully configurable thanks to a fully connected multi-bus. The original architecture was fixed and it was not possible to modify the structure of the DPR.

A DPR reconfiguration is executed in either 3 or 9 clock cycles.Thanks to the DUCK concept and the parallelization of the reconfiguration processes, this reconfiguration is reduced to one clock cycle. An example of the interaction between a functional unit of a DPR and its dedicated DUCK is given Figure 9. In this example, each bit register used for the configuration of the functionnality of the FU becomes a parallel configuration register so that each configuration bit can be switched. Each DPR requires $N_{bconfPE}$ = 38 configura-

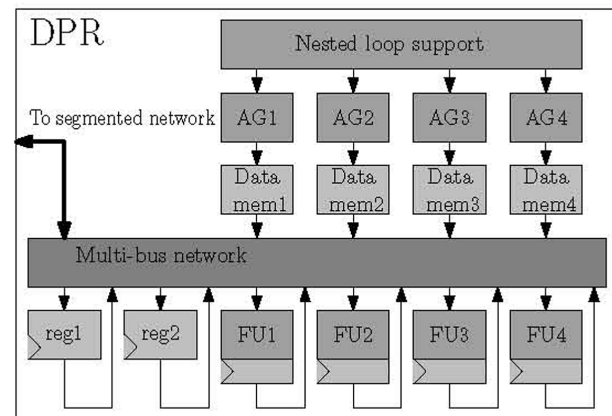


Figure 8. A DPR computing element of the DART reconfigurable architecture.

Table 4. Configuration size for each unit of one cluster of six DPR

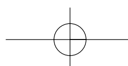
Reconfiguration Target	Size (Bits) /Resource	Size (Bits) /DPR	Size (Bits) /Cluster
AG	1	4	24
Registers	1	6	36
Add/Subb(FU1-3)	3	6	36
Multiplier(FU2-4)	11	22	132
Total		38	228

tion bits (Table IV). Figure 10 shows the implemented architecture with 6 DPR needed for this application. Therefore, 228 bits are needed for each configuration of the whole DART architecture, interconnection DUCK excepted.

Two kinds of interconnection units are used. First, one kind of interconnection (DB_{dpr}) is necessary to connect all the resources inside a DPR. 18 inputs are connectable to 10 outputs. Therefore, the configuration size ($TCDB_{dpr}$) of this interconnection unit requires :

$$TCDB_{dpr} = 10 \times \lceil log_2(18) \rceil = 50 \ bits \quad (1)$$

The second kind of interconnection unit ($DB_{cluster}$) is an 8-bitwidth crossbar type and is necessary for the communication between the 60 registers and functional units of all the DPR inside one cluster



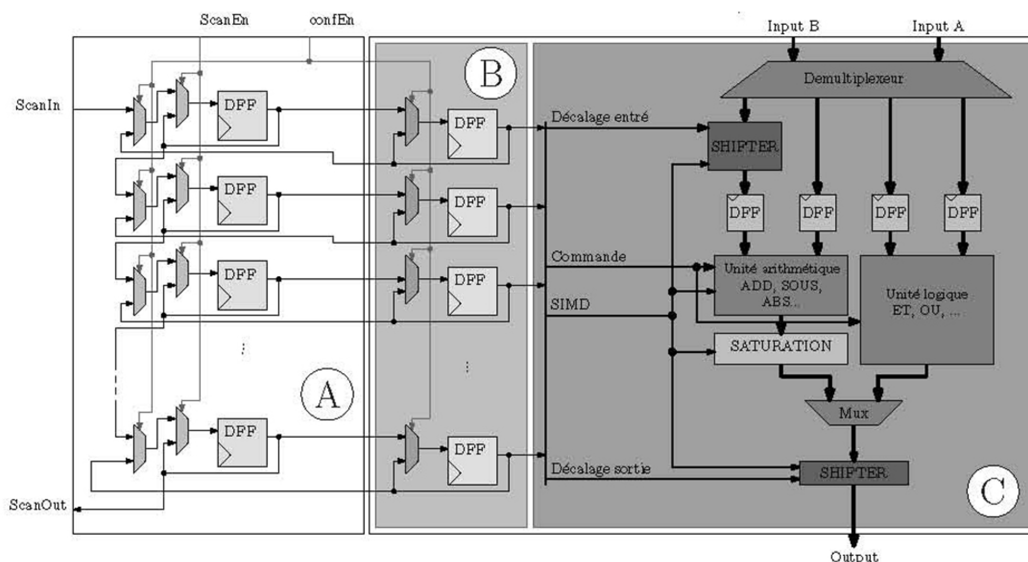


Figure 9. DUCK Generated for a fast reconfiguration on DART architecture

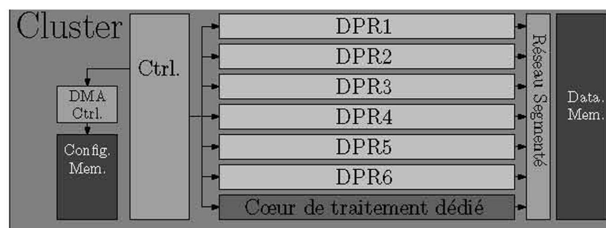


Figure 10. Resource allocation on the implemented DPR of DART

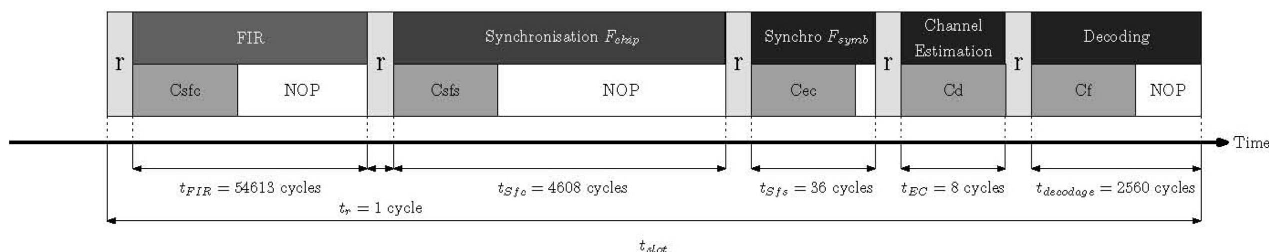


Figure 11. Gant diagram of computing and reconfiguration process for the DART implementation

with a maximum of 30 possible connections. Therefore, the configuration size ($TCDB_{cluster}$) of this interconnection unit requires:

$$TCDB_{cluster} = 8 \times [log_2(30)] = 40\ bits \qquad (2)$$

The WCDMA implementation on DART has already been presented in [10]. The configuration size of all interconnection units ($TCDB$) requires $TCDB = 6 \times 50+40 = 340$ bits. The complete bitstream size for the whole DART architecture (TC_{DART}) requires $TC_{DART} = 228 + 340 = 568$ bits. On DART, the reconfiguration is executed at the same frequency as the processing frequency, which is up to 130 MHz for the reference design in [11].

The number of domains needed (ND) is specified by the available time between two configurations ($Prop_t$). $Prop_t$ is determined by the number of registers to reconfigure and the speed of the configuration memory to read the bitstream which is equal to 300 MHz. A complete new context can be propagated in

$$Prop_t = \frac{580/8}{300E^6} = 240\ ns \qquad (3)$$

The shortest propagation time available is executed during the channel estimation function. This context is only implemented during 8 clock cycles. Considering that for this application, the working frequency is fixed to 93 MHz, the propagation time available is then equal to 86.22 ns. Therefore, the number of reconfiguration domain required to maintain the reconfiguration constraints are equal to:

$$N_D = \left\lceil \frac{241E^{-9}}{86.22E^{-9}} \right\rceil = 3 \qquad (4)$$

05-Lallet-v4n1-AF 19.08.09 19:11 Page 44

Figure 11 shows the timing implementation of the different functions on one reconfiguration domain. Each domain receives the same configuration in parallel.While the cluster computes, the DUCKs are receiving the different configuration simultaneously. Next configurations are prepared during the execution of the functions. *Csfc* stands for configuration of the function *Synchronisation* F_{chip}, Csfs for configuration of the function *Synchronisation* F_{symb}, *Cec* for configuration of the function *Channel Estimation*, *Cd* for configuration of the function *Decoding*, and *Cf* for configuration of the function *FIR*.

5. CONCLUSIONS

In this paper, a new fast dynamically reconfigurable concept for embedded hardware accelerator is proposed. This method allows to use dynamic reconfiguration and to gain in flexibility and in silicon area while maintaining the timing constraints. The reconfiguration time is reduced compared to traditional FPGA or DRP. The proposed concept is based on the isolation of the configuration paths and the configuration resources, which allows to prepare new contexts during the computations. The second concept consists in the possibility to split the configuration path while maintaining a unique computing path in order to propagate the configuration through several configuration paths at the same time. In the near future, we will develop exploration tools in order to estimate the possible configuration paths to automatically get the best trade-off between speed, performance and silicon area.

ACKNOWLEDGMENTS

This work has been performed in the context of the CoMap project and is financed by the French Ministry of Foreign Affairs. The authors would like to thank A.Kupriyanov, D.Kiessler, F.Hanning, J.Teich, B.Pottier and R.Keryell for their fruitfull collaboration.

REFERENCES

[1] M. Hariyama, W. Chong, S. Ogata, and M. Kameyama. Novel Switch Block Architecture Using Non-Volatile Functional Pass-Gate for Multi-Context FPGAs. In *Proceedings of the IEEE Computer Society Annual Symposium On VLSI (ISVLSI)*,2005, pages 46–50.

[2] I. Robertson and J. Irvine. A Design Flow for Partially Reconfigurable Hardware. *ACM Transaction on Embedded Computing Systems*, vol. 3, no. 2, May 2004, pages 257–283.

[3] D. Kawakami, Y. Shibata, and H. Amano. A prototype chip of multicontext FPGA with DRAM for Virtual Hardware. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2001, pages 17–18.

[4] D. Koch, A. Ahmadinia, C. Bobda, H. Kalte, and J. Teich. FPGA Architecture Extensions for Preemptive Multitasking and Hardware Defragmentation. In *Proceedings of the IEEE Conference on Field-Programmable Technology (FPT)*, 2004, pages 433–436.

[5] L. Lagadec and B. Pottier. Object-Oriented Meta Tools for Reconfigurable Architectures. In *Proceedings of the SPIE Conference on Modeling, Signal Processing, and Control* , 2000, pages 69–79.

[6] V. B. Lecuyer, M. A. Aguirre, A. B. Torralba, L. G. Franquelo, and J. Faura. Decoder-Driven Switching Matrices in Multicontext FPGAs: Area Reduction and Their Effect on Routability. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, 1999, pages 463–466.

[7] M. Suzuki, Y. Hasegawa, V. M. Tuan, S. Abe, and H. Amano. A Cost-Effective Context Memory Structure for Dynamically Reconfigurable Processors. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2006, pages 1101–1109.

[8] T. Ojanpera and R. Prasad. *Wideband CDMA For Third Generation Mobile Communication*. Artech House Publishers, Norwood, MA, USA, 1998.

[9] P. T. Wolkotte, G. J. M. Smit, and J. E. Becker. Energy-Efficient NoC for Best-Effort Communication. In *Proceedings of the International Conference on Field-Programmable Logic, Reconfigurable Computing, and Applications (FPL)*, 2005, pages 197–202.

[10] R. David, D. Chillet, S. Pillement, and O. Sentieys. DART: A Dynamically Reconfigurable Architecture Dealing with Future Mobile Telecommunications Constraints. In Proceedings of the IEEE Reconfigurable Architectures Workshop (RAW), 2002, pages 118–123.

[11] S. Pillement, R. David, and O. Sentieys. DART : A Functional-Level Reconfigurable Architecture for High Energy Efficiency. In EURASIP Journal on Embedded Systems, vol. 2008, January 2008,13 pages.

[12] Xilinx. Virtex series configuration architecture. Technical report, 2004.

[13] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach and M.Weinhardt PACT XPP - A Self-Reconfigurable Data Processing Architecture, Journal of Supercomput, 2003, pages 167–184.

[14] B. Mei, A. Lambrechts, D. Verkest, J. Y. Mignolet and R. Lauwereins. Architecture Exploration for a Reconfigurable Architecture Template, in IEEE Journal of Design and Test, March 2005, vol. 22, no. 2, pages 90–101.

[15] V. Betz and J. Rose. VPR: A new packing, placement and routing tool for FPGA research. in *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL), 1997, pages 213–222.*

[16] J. Pistorius, M. Hutton, A. Mishchenko, and R. Brayton. Benchmarking Method and Designs Targeting Logic Synthesis for FPGAs. In *Proceedings of the ACM/SIGDA International Workshop on Logic and Synthesis (IWLS), 2007, pages 230–237.*