# Configuration Sequencing with Self Configurable Binary Multipliers

Mathew Wojko
Dept. of Electrical & Computer Engineering
The University of Newcastle
Callaghan, NSW 2308, Australia
mwojko@ee.newcastle.edu.au

Hossam ElGindy
School of Computer Science and Engineering
University of New South Wales
Sydney, NSW 2052, Australia
hossam@cse.unsw.edu.au

**Abstract.**
*In this paper we present a hardware design technique which utilises run-time reconfiguration for a particular class of applications. For a multiplication circuit implemented within an FPGA, a specific instance of multiplying by a constant provides a significant reduction of required logic when compared to the generic case when multiplying any two arbitrary values. The use of reconfiguration allows the specific constant value to be updated, such that at any time instance the constant multiplication value will be fixed, however over time this constant value can change via reconfiguration. Through investigation and manipulation of the sequence of required multiplication operations for given applications, sequences of multiplication operations can be obtained where one input changes at a rate slower than the other input. That is one input to the multiplier is fixed for a set number of cycles, hence allowing it to be configured in hardware as a constant and reconfigured at the periodicity of its change. Applications such as the IDEA encryption algorithm and every cycle Adaptive FIR filtering are presented which utilise this reconfiguration technique providing reduced logic implementations while not compromising the performance of the design.*

## 1 Introduction

Since the introduction of reconfigurable computing as a viable computing alternative, several reconfigurable computing techniques have been suggested for applications which have been previously performed/executed by existing well known computing paradigms [2, 6, 10, 9, 13]. While reconfigurable computing defines its own computing paradigm, there are varying techniques which in general categorise its use and application. In this paper we provide a case for the use of Run-Time Reconfiguration (RTR) within FPGAs for high throughput data streaming applications requiring binary multiplication. We show how reconfiguration can be exploited without any effect on the operation of the application while at the same time providing reduced logic area implementations over those that do not use reconfiguration.

In many cases, reconfiguration can be viewed as providing an Area Time (AT) design alternative. Instead of implementing application specific circuitry, a configurable resource is provided that can be reconfigured over time to provide the specific functionality when required. This precludes an AT trade-off for application designs utilising reconfiguration over those that don't. Reconfiguration provides an AT advantage when the reduction in area coupled with the time to reconfigure provides better utilisation of the hardware resource.

For reconfigurable designs requiring $r$ reconfiguration cycles every $n$ cycles, the data throughput rate is reduced by the fraction $\frac{n}{n+r}$. While the hardware utilisation may be higher, the maximum potential throughput is reduced. Provided $r \leq n$, reconfiguration delay can be eliminated if two individual configurable resources are provided such that while one resource is being reconfigured, the other is active in operation until such time a new configuration is required. This process can be seen below in Figure 1. This process of switching between reconfigured contexts, termed *reconfigurable context switching* has been used previously in [4] and is a derivative of the DPGA concept and temporal pipelining [3]. The implication of such a proposal is that additional hardware resource is required for the second configuration. The overall hardware cost must not exceed that of any design techniques not using reconfiguration, since for both the maximum throughput rate is now achieved. Additionally, to sustain maximal throughput, applications must be able to effectively use one configurable resource while scheduling and performing the configuration of the second such that it can be seamlessly switched into operation.

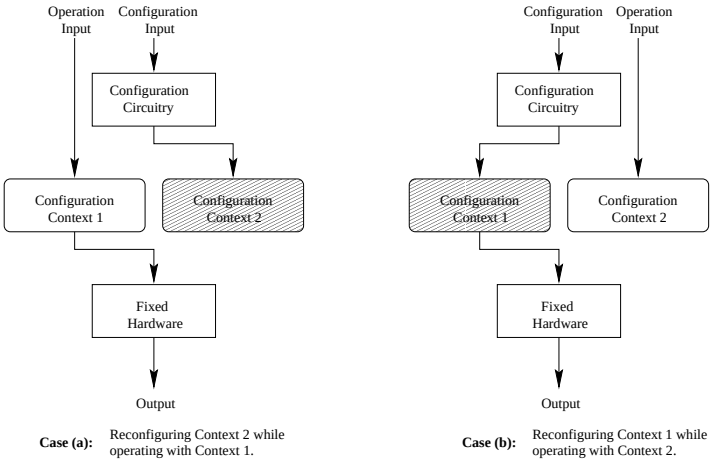
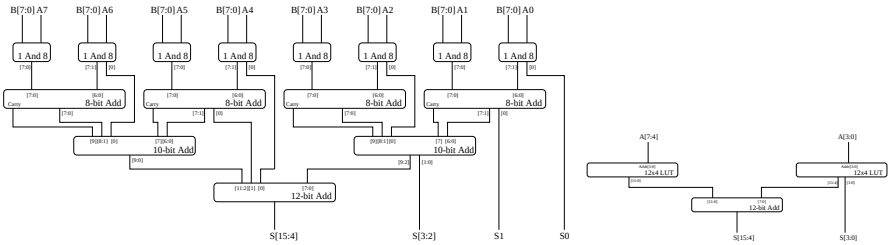



**Fig. 1.** Reconfiguring one context while the other is operating.

If these criteria can be satisfied then applications which can use the reconfigurable context switching technique to its advantage will comparatively provide a resource efficient technique when implemented. For this paper, we present a self configurable binary multiplier design comprising of two configuration resources which allow continuous processing on input data streams at the maximal throughput rate. Based on the number of cycles required to reconfigure, $r$, we then investigate the data flow for adaptive FIR filtering and the IDEA encryption algorithms with the objective of obtaining continuous sequences of constant value multiplications. This then allows such algorithms to utitilise the multiplier design presented. The result being an application design alternative which incorporates reconfiguration to allow the possibility for higher utilisation of the hardware resource over more traditional conventional design techniques.

## 2 Self-Configurable Multiplication Technique

The self-configurable multiplication technique can be applied to any FPGA allowing read/write RAM primitives mapped to the Logic Elements (LEs) of the architecture [14]. The self-configurable technique uses a lookup based distributed arithmetic approach [8] to perform the multiplication of two numbers. By storing one value in hardware Look-Up Tables (LUTs), the Distributed Arithmetic (DA) approach technique is frequently used for FPGA implementations since it provides significant hardware reduction. A comparison between a common general purpose multiplication technique and the constant coefficient DA approach is shown below in Figure 2 for 8-bit multiplication. The implementation difference is considerable since the cyclic convolution of inputs and top two stages of addition are replaced by two 12-bit wide, 4-bit addressable LUTs.




(a) Parallel Add (b) Constant Coefficient

**Fig. 2.** 8-bit Multiplier and Constant Coefficient Multiplier functional diagrams.

While the constant DA multiplier provides significant hardware reduction, as a general purpose multiplier its disadvantage is that one input remains constant. By use of reconfiguration we can update the LUT contents at run time providing a two input multiplier. Previous work has shown reconfigurable multiplier implementations on the XC6200 series FPGA [12, 1]. The reconfiguration of LUT content is performed through the SRAM configuration interface where the configuration data is precomputed. This interface has physical bit width and access time limitations. It is shown that a single LUT can be updated in $1.45\mu s$ at 33MHz [1], which relates to 48 cycles for a design operating at this speed.

The self configurable multiplication technique eliminates this physical limitation by performing the LUT reconfiguration on chip. On new input, the configuration data is computed on chip and stored into the LUTs in parallel. No outside configuration intervention is required, hence describing the self configurable attribute of the multiplier. Reconfiguration is thus not physically limited and as a result faster. Typically, the required number of cycles for reconfiguring the logic content is that of the LUT addressable range.

Below, in Figure 3, a sixteen cycle 8-bit self configurable multiplier design is presented. Present are two 12-bit wide 4-bit addressable LUTs and a 12-bit adder which represent the basis of the 8-bit constant coefficient multiplier. The 4-bit counter and 12-bit accumulator represent the address and data generators

respectively used to reconfigure the LUTs with new values. The two eight bit inputs $A$ and $B$ represent the multiplier inputs and the input line $CHANGE$ is used to signify that input B has changed and the multiplier LUTs must be reconfigured. The multiplier reconfiguration time is 16 cycles. Multipliers of less reconfiguration time can be implemented by reducing the LUT addressable size. A generic design technique is presented [14] where the required reconfiguration time is a design parameter such that multipliers with 8- and 4-cycle reconfiguration times can be implemented.

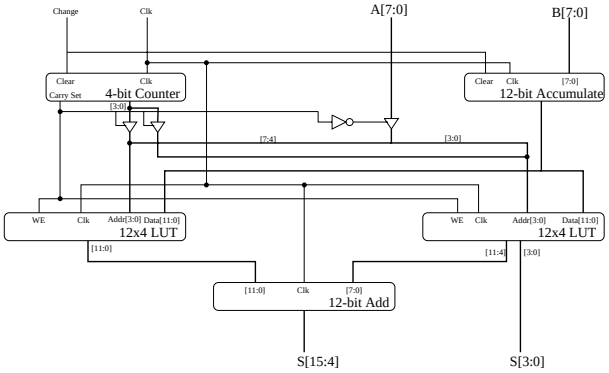



**Fig. 3.** Functional Block Diagram for an 8-bit 16-cycle Reconfigurable Multiplier.

Given this reconfigurable multiplier design, we extend it by increasing the configuration contexts to two. As shown in Figure 3 the two $12 \times 4$-bit addressable LUTs represent one configuration context. By providing another set of $12 \times 4$-bit LUTs we can guarantee continuous operation of the multiplier, i.e. while one context is operating the other is being reconfigured and vice versa, provided that $r \leq n$. In Tables 1(a), 1(b) and 1(c), we provide the hardware resource requirements (by using the Configurable Logic Block (CLB) count for implementations on Xilinx 4000 series FPGAs [7]) for 8-, 16- and 32-bit self configurable multipliers which use two configurable contexts. These values are compared to the implementation and associated CLB count for arbitrary value parallel add multipliers of the same bit size. Note that in every case the sixteen cycle reconfigurable multipliers require a lower CLB count than the parallel add technique, while the 8 cycle multipliers improve at 16-bits and up, and the four cycle multipliers provide no benefit.

| Reconfig. Steps | # CLBs |
|---|---|
| 16 | 44 |
| 8 | 74 |
| 4 | 142 |
| 0 (Parallel Add) | 70 |

(a) 8-bit implementations

| Reconfig. Steps | # CLBs |
|---|---|
| 16 | 132 |
| 8 | 213 |
| 4 | 426 |
| 0 (Parallel Add) | 273 |

(b) 16-bit implementations

| Reconfig. Steps | # CLBs |
|---|---|
| 16 | 455 |
| 8 | 765 |
| 4 | 1418 |
| 0 (Parallel Add) | 1049 |

(c) 32-bit implementations

**Table 1.** 8-, 16- and 32-bit multiplier implementations on Xilinx XC4000 FPGAs.

## 3 IDEA Encryption

The IDEA encryption algorithm [11] presents a high throughput computationally intense algorithm that has been suggested for use as a benchmark for reconfigurable computing [5]. IDEA uses a 128-bit key and operates on 64-bit blocks of data. Each block is split into 4 16-bit sub-blocks and passed through eight successive rounds of computation where each round is as illustrated in Figure 4. For each round of encryption, the encryption key values vary since they are each defined as different 16-bit subsequences of the 128-bit key which remains fixed for the duration of the encryption process. All primitive operations within the IDEA algorithm are performed on 16-bit data.

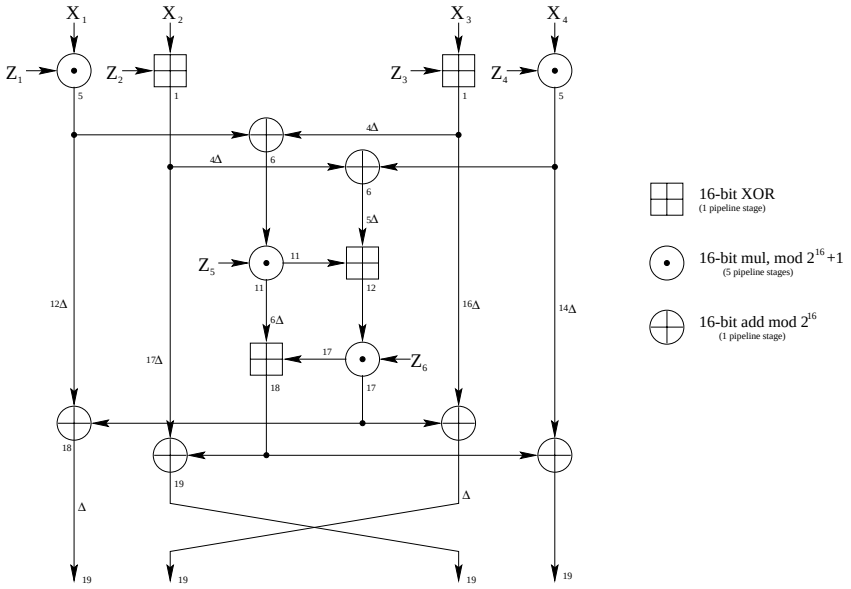



**Fig. 4.** One round of the IDEA Encryption algorithm.

Due to current logic capacity limitations of FPGAs, typical single chip implementations of the IDEA algorithm generally contain one instantiated single round computation unit [5]. Depending on the capacity of the FPGA this computation unit could be implemented as bit serial, bit parallel or some combination of both. The computation unit is equally time multiplexed between the eight successive rounds of required computation. With FPGA device logic capacities always increasing, parallel implementations of a single round of computation are compared with and without reconfiguration, i.e. with and without the use of the self configurable multiplier. To maximise throughput, each functional unit within the single round of computation is pipelined. Additional buffering for synchronisation between pipelined operations is used. The buffering infrastructure is set the same for implementations with and without reconfiguration. While the self configurable multiplier has a latency one cycle less than the general multiplier, we add an additional latency stage for buffer and control compatibility, and reuse. The buffering delays for synchronisation are implemented as displayed in Figure 4.

Between each round of computation, six new 16-bit sub-key sequences $Z_1 \ldots Z_6$ are selected from the 128-bit encryption key. These values remain constant for the duration of the round of computation. We observe in Figure 4 the pipeline latency to be 19 cycles. As a result, to achieve maximum data throughput from this architecture each round of computation is set to process 19 data values before the sub-key sequences are updated for the next round. In providing an implementation that uses the self configurable multiplier we identify that one input to the multiplier be constant for at least the number of required reconfiguration cycles, i.e. while one configuration context is being reconfigured, the other must be operating as a fixed coefficient multiplier. Since the single round latency is calculated at 19 cycles, using the 16-cycle self configurable multiplier will guarantee that the design performance will not be hampered by the time to reconfigure. New configuration contexts can be provided as they are required since each context has a total of 19 cycles to reconfigure its content.

Data streams are injected into the hardware implementation 19 blocks at a time. These 19 blocks of data ensure the pipeline is completely utilised and circulate through the single round computation unit eight times before exiting. At any time, the 19 blocks of data are stored within the 19 pipeline stages that exist within the implementation. For each round of computation, the 16-bit sub-key operators derived from the 128-bit key are constant. Each multiplier is configured to multiply by the constant 16-bit value for this time. During this time, the other multiplication configuration contexts are being configured for the next round of computation. However, we must ensure that the reconfiguration actions and switching of contexts are properly synchronised with the datapath latency within the implementation. Thus we must provide a sequencing of configurations such that correct operation is maintained with the data flow. Observing Figure 4, we see the multiply units situated in different latency positions. Relative to the time instance $t_0$ when data blocks first enter the round of computation, we provide configuration start times for each multiplication unit. For the two multiplication units multiplying by the constants $Z_1$ and $Z_4$, we begin reconfiguring their non-active configuration context at time $t_0 - 16$, and for the multiplication units multiplying by $Z_5$ and $Z_6$, we begin their reconfiguration at times $t_0 - 10$ and $t_0 - 4$ respectively. Once each context is configured it immediately becomes active reflecting the new constant value for the next round of computation.

Through the effective use of configuration sequencing we can guarantee that the constant value multiplications are scheduled at the right time to match the required data stream operations. We provide no negative impact on the performance of data flow, illustrating a hardware resource advantage over implementations that do not use reconfiguration. Preliminary implementation differences between designs which do and do not use reconfiguration have shown a CLB count saving of around 500. This comparative difference can be justified by referencing Table 1(b). Here we observe the CLB count difference between the 16 cycle and 0 cycle parallel add multiplier as 141. Given there are four multiplication units within the single round of IDEA computation, an implementation using reconfiguration should theoretically provide a savings of 564 CLBs.

## 4 Adaptive FIR Filtering

An FIR filter computes the dot product between a weighted coefficient vector and a finite series of time samples. Typically the coefficients are pre-computed for the filter and remain constant for the entire mode of operation. Samples stream through the filter such that the dot product result is computed every cycle from a window of samples equal to the coefficient vector length. The width of the sample window and coefficient vector is referred to as the number of taps the filter possesses. FIR implementations are well suited to FPGAs typically because the bit width of the operands is relatively small, i.e. between 8 and 16 bits, distributed arithmetic approaches can be used [8] for fixed coefficients, and pipelining and parallelism can be well exploited.

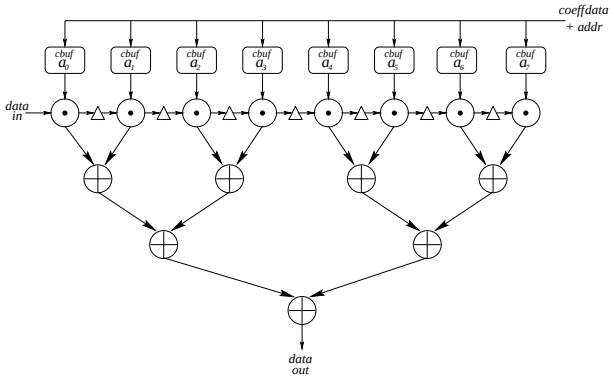



**Fig. 5.** 8 Tap FIR filter block diagram.

Adaptive FIR filtering allows the coefficient vector to be updated over time. The rate at which this vector can be updated is dependent on the application. Most adaptive FIR filters use multipliers which can multiply any two arbitrary values, and an interface which allows any coefficient values to be updated at any time. This is illustrated in Figure 5 with the addressable $coeff\_data$ interface and the coefficient buffers $cbuf$.

In order to utilise self configurable multipliers, we observe the data flow for the FIR filter. We see in Figure 5 that data flows into the filter from the left side and proceeds to the right where a delay stage is encountered between every tap (or multiplier). As a result we see every input sample present within the filter for a number of cycles equal to the number of taps. During this time, each sample is multiplied in order by the filter coefficients $a_0 \ldots a_7$. We observe the constant multiplication of $a_0 \ldots a_7$ by each input sample. If a self configurable multiplier with two configuration contexts is provided that can update a given context in the same or better time for which an input sample exists within the filter, then reconfiguration can be used to provide benefit. That is, the multiplier reconfiguration time $r$ must be less than or equal to the number of taps for the filter. In the case of the example 8-tap FIR filter provided, an 8-cycle configurable multiplier is required. For a 16-tap Adaptive FIR filter, a 16-cycle configurable multiplier is required.

The design for an Adaptive FIR filter using self configurable multipliers sees

changes to the internal data flow whereby the coefficients now propagate from multiplier to multiplier in the filter rather than the input samples. The operation of the filter is as follows. Filter coefficients are stored in the buffers labeled $cbuf$. These coefficients circulate through and around the filter multipliers such that each multiplier will repeatatively see the continuous sequence of all coefficients in order $a_0 \ldots a_7$. These coefficients can be updated at any time through the interface provided. The addressing mechanism for updating any particular coefficient must be aware of its current buffer location in order to update it correctly. Input samples stream in every cycle and are assigned to be configured within a multiplier. It is here that the configuration sequencing is critical to ensuring each input sample is multiplied by all coefficients in the order $a_0 \ldots a_7$. We must initiate the reconfiguration of each multiplier with an input sample such that its context switch occurs when coefficient $a_0$ is present. It is apparent that we are providing a circulating sequence of initiating reconfiguration of the multipliers within the filter which follows the circulation of coefficient values within. The initiation of configuration for each multiplier is offset $r$ cycles before the arrival of coefficient $a_0$. The *data in* interface is broadcast to each multiplier, such that when the configuration of one multiplier is initiated it takes the sample value from the data in bus and configures its switch-able context to this value. In Figure 6 we see an example of the incorporation of self configurable multipliers in an an 8-tap Adaptive FIR filter.

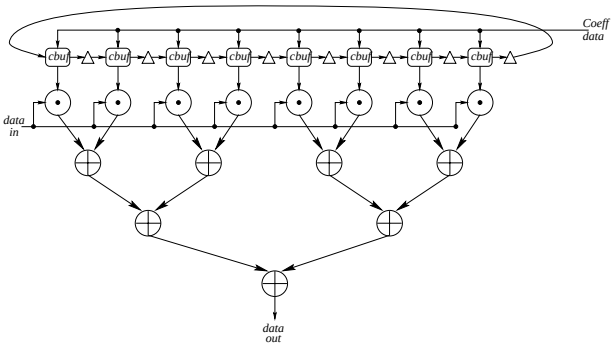



**Fig. 6.** 8 Tap Adaptive FIR filter block diagram using reconfiguration.

Provided that $r$ for the multiplier is equal to or less than the number of the taps for the filter, and its CLB count is less than that of the arbitrary value parallel multiplier then a non performance compromising benefit will be achieved. Below in Table 2, we see several cases of adaptive FIR filters where the use of self configurable multipliers provides a reduced CLB count for implementations on the Xilinx XC4000 device series while not compromising performance. This includes the additional overhead for the updating of rotating coefficients and configuration sequencing control.

| Filter Properties | #CLBs /w | #CLBs /wo | Ctrl o/head | △ CLB Count | % Saving |
|---|---|---|---|---|---|
| 16-tap 8-bit | 1006 | 1344 | 78 CLBs | 338 | 25.1 |
| 8-tap 16-bit | 1976 | 2384 | 72 CLBs | 408 | 17.1 |
| 16-tap 16-bit | 2663 | 4777 | 142 CLBs | 2114 | 44.3 |

**Table 2.** FIR implementations /w and /wo reconfiguration on Xilinx XC4000 FPGAs.

## 5 Conclusion

In this paper we have presented a case for the usefulness of RTR providing an alternative solution for traditional hardware implementations of given applications requiring multiplication. Considering the self configurable binary multiplication technique with two configuration contexts, application data flow analysis was required for its successful use. Two applications, IDEA encryption and adaptive FIR filtering were targeted. It was discovered that their data-flow could be sufficiently arranged such that one input to each multiplier in the application changed at a rate slower than the other. As a result, the self configurable multiplication technique was applied to these derived sequences of operations such that the reduced hardware resource requirements of the technique could be exploited. The results demonstrate a design alternative not impacting on achievable performance while providing an impetus to allow an area conscious advantage.

## References


1. J. Burns A. Donlin J. Hoggs S. Singh M. de Wit. A dynamic reconfiguration runtime system. In *FPGAs for Custom Computing Machines*, April 1997.
2. A. DeHon. *Reconfigurable Architectures for General-Purpose Computing.* PhD thesis, Massachusetts Institute of Technology, 1996.
3. A. DeHon. DPGA utilization and application. In *FPGA '96 ACM/SIGDA Fourth International Symposium on FPGAs*, Montery CA, February 1996.
4. H. Eggers P. Lysaght H. Dick and G. McGregor. Fast reconfigurable crossbar switching in FPGAs. In *FPL'96.*
5. N. Weaver E. Caspi. Idea as a benchmark for reconfigurable computing. Technical Report Available from http://www.cs.berkeley.edu/projects/brass/projects.html, BRASS Research Group, University of Berkeley, December 1996.
6. J. R. Hauser and J. Wawrzynek. Garp: A mips processor with a reconfigurable coprocessor. In *Proc. FPGAs for Custom Computing Machines*, pages 12–21. IEEE, April 1997.
7. Xilinx Incorporated. Xilinx XC4000 data sheet, 1996.
8. Les Mintzer. FIR filters with field-programmable gate arrays. *Journal of VLSI Signal Processing*, 6(2):119–127, 1993.
9. C. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, T. Arnold, and M. Gokhale. The NAPA adaptive processing architecture. In *Proc. FPGAs for Custom Computing Machines*, pages 28–37. IEEE, April 1998.
10. H. Schmit. Incremental reconfiguration for pipelined applications. In *Proc. FPGAs for Custom Computing Machines*, pages 47–55. IEEE, April 1997.
11. W. Stallings. *Network and Internetwork Security: Principles and Practice.* Prentice Hall, 1995.
12. B. Slous T. Kean, B. New. A multiplier for the XC6200. In *Sixth International Workshop on Field Programmable Logic and Appications*, 1996.
13. J. Vuillemin , P. Bertin , D. Roncin , M. Shand , H. Touati and P. Boucard. Programmable active memories: Reconfigurable systems come of age. *IEEE Transactions on VLSI Systems*, 4(1):56–69, March 1996.
14. M. Wojko and H. ElGindy. Self configurable binary multipliers for LUT addressable FPGAs. In Tam Shardi, editor, *Proceedings of PART'98*, Newcastle, New South Wales, Australia, September 1998. Springer.