Implementation of Parallel LFSR-based Applications on an Adaptive DSP featuring a Pipelined Configurable Gate Array

Claudio Mucci[1], Luca Vanzolini[1], Ilario Mirimin[1], Daniele Gazzola[1], Antonio Deledda[1], Sebastian Goller[2], Joachim Knaeblein[3], Axel Schneider[3], Luca Ciccarelli[4], Fabio Campi[4]

[1]ARCES - University of Bologna [2]Technische Universität Chemnitz
[3]Alcatel-Lucent Deutschland AG [4]FTM, STMicroelectronics, Agrate Brianza

Abstract

Linear feedback shift registers (LFSRs) are common structures in many application fields, including cryptography, digital broadcasting and communication. High-throughput requirements need highly parallel implementations, usually accomplished in state of the art system on chips (SoCs) with application specific coprocessors. Although this approach achieves the required performance, it rapidly shows lack of flexibility when those devices are proposed, as an example, for multi-standard modems or for security applications in which run-time update can provide added value. This paper shows the implementation of parallel LFSR-based applications on an embedded adaptive DSP featuring a Pipelined Configurable Gate Array (PiCoGA). With respect to standard embedded FPGAs, pipelined devices usually provide better performance, e.g. in terms of speed, but they commonly show the undeniable drawback of additional design constraints. As a test-case, we consider the implementation of the 32-bit CRC used in the Ethernet standard that achieves on the target architecture up to ∼25Gbit/sec throughput, with a parallel LFSR processing 128 bit at time, which is comparable to the performance offered by some ASIC devices.

1. Introduction

Linear feedback shift registers (LFSRs) are widely used circuits in modern multimedia and communication devices. As an example, they represent the mathematical background of the well known Cyclic Redundancy Check (CRC) code used in many telecommunication protocols to verify the correctness of transmitted data. Furthermore, thanks to their statistical proprieties, LFSRs are commonly used to scramble the information content of a data stream, for either security increase or noise effect reduction. In this case, the LFSR provides a pseudo-random sequence which can be correlated with the data stream to be transmitted or distributed. To provide a brief overview of LFSR applications, we can consider three main fields, analyzing added value provided by flexible implementations.

The work presented in this paper is done within the MORPHEUS project (IST FP6, project no. 027342), which is sponsored by the European Commission under the 6th Framework program.

CRC is the redundancy check commonly used in the physical layer of transmission protocols like Ethernet, SONET and Bluetooth. Only in the Wikipedia, ∼25 standards are reported, featuring different numbers of bits used in the shift register and polynomial generator. Most of them are also different for the required bit-rates, which can range from few Mbit/sec to the tens of Gbit/sec of Gigabit Ethernet protocols. Multi-mode devices need to handle this in a flexible way, requiring a dedicated circuit for each supported standard or a reconfigurable/reprogrammable implementation.

Digital Broadcasting and Communication use LFSRs to randomize the transmitted bitstream in order to avoid short repeating sequences of 0's or 1's which may complicate symbols tracking at the receiver or interfere with other transmissions. Depending on the standard, bit streams can be randomized correlating the original data with the sequence generated by an LFSR which can work at the same frequency (and in this case is termed scrambling) or with a different frequency (and in this case is termed spreading). 802.11 (WiFi), 802.15.4 (ZigBee), 802.16 (WiMax), Digital Audio/Video Broadcasting (DAB/DVB) are well known examples of standards including scrambling or spreading or both in their definition, making thus appealing reprogrammable solutions for flexible multi-standard devices.

Stream ciphers are symmetric encryption systems which correlate the plaintext bits with a pseudo-random sequence generated by the combination of the bit streams of one or more LFSRs working in parallel. Example of applications of stream ciphers are the A5/1 standard which ensures communication privacy of GSM telephones, E0 standard for the Bluetooth or the content scramble system used for digital right management which uses a 40-bit stream cipher.

Flexible solutions offered for the integration in SoCs include of course processors, be it general-purpose or application specific DSPs, embedded FPGAs [1] and reconfigurable datapaths [2–4]. Targeting the Gbit/sec bandwidth, most of them are not suitable solutions for parallel LFSR-based applications which couple high performance to bit-level programming. In fact, general-purpose processors provide word-level computation, while for LFSRs most of the computation is performed at bit-level. On the contrary, the full bit-level programmability offered by embedded FPGAs shows the undeniable drawback to be paid for added flexibility: the possible working frequency is reduced. Reconfigurable datapaths are in the middle of this scenario, and thanks to their deeply pipelined organization they can offer the required performance and flexibility.

This paper shows the implementation of LFSR-based applications on the Pipelined Configurable Gate Array (PiCoGA) integrated in the DREAM adaptive processor [5] featuring a working frequency of 200MHz. After an overview of the theory, the main parallelization methods proposed in literature for both processors and application-specific circuits will be analyzed. Then, the implementation on the target architecture of the 32-bit CRC defined for the Ethernet protocol will be described. Our implementation allows to achieve up to ∼25 Gbit/sec, which is a bandwidth comparable to the performance of some ASIC implementations, as will be shown at the end of the paper, and roughly three orders of magnitude better of the performance offered by embedded RISC processors.

2. Theoretical background and related work

The serial block diagrams of both CRC and Scrambler are reported in Fig. 1, which shows that the main difference between the two is the fact that in the case of CRC input bits are combined with bits flowing in the feedback loop, while in the case of the scrambler the LFSR is an autonomous system whose output bits are combined with the input bit stream. In both cases, as well as for most of the real LFSR applications, we consider feedback loops defined over an extension of the Galois Field GF(2). This means that the additions necessary in the loop are defined in GF(2) and thus implemented with exclusive-ORs.

Let us define $\mathbf{x}(\mathbf{n})$ the state of the serial LFSR at time

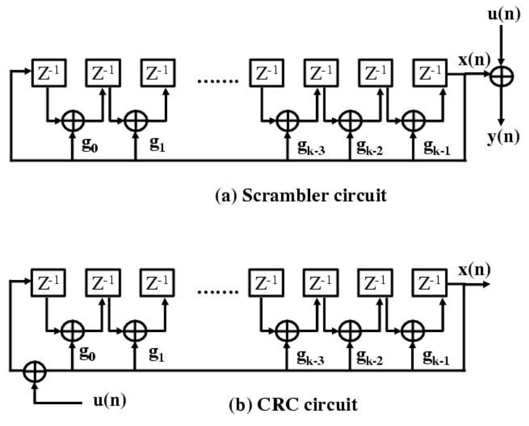

Figure 1. Scrambler and CRC block diagrams

n, thus the most general form in which an update can be represented is:

$$\mathbf{x}(n+1) = \mathbf{A}\,\mathbf{x}(n);$$

where $\mathbf{A}$ is the $k \times k$ companion matrix:

$$\mathbf{A} = \begin{pmatrix} 0 & 0 & \dots & 0 & 0 & -g_0 \\ 1 & 0 & \dots & 0 & 0 & -g_1 \\ 0 & 1 & \dots & 0 & 0 & -g_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & 0 & -g_{k-2} \\ 0 & 0 & \dots & 0 & 1 & -g_{k-1} \end{pmatrix};$$

and k is the degree of the polynomial generator for the LFSR, and g_i is the i^{th} root of the polynomial generator. Given an input sequence of bit $u(n)$, with $n \in 0, 1, \dots, N-1$, the CRC mathematical formulation is:

$$\mathbf{x}(n+1) = \mathbf{A}\,\mathbf{x}(n) + \mathbf{b}\,u(n);$$

where $\mathbf{b}$ is a the vector $[-g_0 \ -g_1 \ \dots \ -g_{k-2} \ -g_{k-1}]^T$. The checksum provided by the CRC computation will be the value of the state variable $\mathbf{x}(n)$. On the contrary, the mathematical formulation of a scrambler is:

$$\begin{array}{l} \mathbf{x}(n+1) = \mathbf{A}\,\mathbf{x}(n); \\ y(n) = \mathbf{C}\,\mathbf{x}(n) + \mathbf{d}\,u(n); \end{array}$$

where $\mathbf{C}$ is a $k \times k$ matrix which selects the state bits combined to the data stream (usually only one bit in the diagonal is used) and d is a single-1 vector in the form $[1\ 0 \dots\ 0]$, used to select the state bit to be correlated to the input stream. Putting all together, we can consider the system:

$$\begin{array}{l} \mathbf{x}(n+1) = \mathbf{A}\,\mathbf{x}(n) + \mathbf{b}\,u(n); \\ y(n) = \mathbf{C}\,\mathbf{x}(n) + \mathbf{d}\,u(n); \end{array}$$

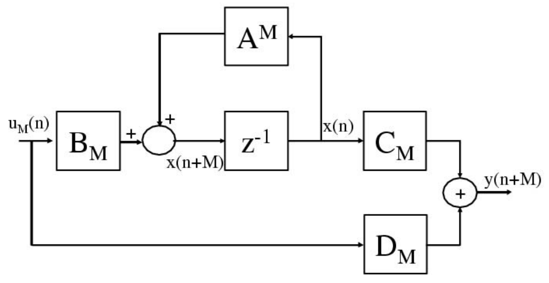

Figure 2. Generic scheme for an M-bit LFSR-based application

in which $\mathbf{C}$ is the identity matrix and $\mathbf{d}$ is a null vector for the CRC, while $\mathbf{b}$ is a null vector and $\mathbf{C}$ a single-1 diagonal matrix for the scrambler.

In both cases, parallelization required to work with multiple input bits at the same time implies the exponentiation of the matrices involved in the computation. If we consider elaborating 2 bits at time, then we need to apply a 2-level look-ahead:

$$\begin{aligned}\mathbf{x}(n+2) &= \mathbf{A}\,\mathbf{x}(n+1) + \mathbf{b}\,u(n+1)\\ &= \mathbf{A}\,(\mathbf{A}\,\mathbf{x}(n) + \mathbf{b}\,u(n)) + \mathbf{b}\,u(n+1)\\ &= \mathbf{A}^2\,\mathbf{x}(n) + \mathbf{A}\,\mathbf{b}\,u(n) + \mathbf{b}\,u(n+1);\\ \mathbf{y}(n+2) &= \mathbf{C}\,\mathbf{x}(n+1) + \mathbf{d}\,u(n+1)\\ &= \mathbf{C}\,(\mathbf{C}\,\mathbf{x}(n) + \mathbf{d}\,u(n)) + \mathbf{d}\,u(n+1);\\ &= \mathbf{C}^2\,\mathbf{x}(n) + \mathbf{C}\,\mathbf{d}\,u(n) + \mathbf{d}\,u(n+1);\end{aligned}$$

Previous result can be generalized for the M-level look-ahead in the following form:

$$\begin{aligned}\mathbf{x}(n+M) &= \mathbf{A}^M\,\mathbf{x}(n) + \mathbf{B}_M\,\mathbf{u}_M(n);\\ \mathbf{y}(n+M) &= \mathbf{C}^M\,\mathbf{x}(n) + \mathbf{D}_M\,\mathbf{u}_M(n);\end{aligned}$$

where $\mathbf{u}_M(n)$ is the M-element vector

$$[u(n+M-1)\; u(n+M-2)\; \ldots u(n+1)\; u(n)]^T$$

and $\mathbf{B}_M$ and $\mathbf{D}_M$ are the $k \times M$ matrices:

$$\begin{aligned}\mathbf{B}_M &= [\mathbf{b}\;\; \mathbf{Ab}\;\; \mathbf{A}^2\mathbf{b}\; \ldots\; \mathbf{A}^{M-1}\mathbf{b}]\\ \mathbf{D}_M &= [\mathbf{d}\;\; \mathbf{Cd}\;\; \mathbf{C}^2\mathbf{d}\; \ldots\; \mathbf{C}^{M-1}\mathbf{d}]\end{aligned}$$

From an implementation point of view, the resulting overall schema is reported in Fig. 2. The matrix $\mathbf{A}^M$ is no more a companion matrix and since it is part of the feedback loop its complexity directly impacts on the performance achieved and in particular on the clock period. On the contrary, the implementation of $\mathbf{B}_M$, $\mathbf{C}_M$ and $\mathbf{D}_M$ can be pipelined to match the performance requirement.

In [6], Pei et al. showed that exponentiation of $\mathbf{A}$, even if optimized, limit the achievable speed-up to $0.5M$ for $M \in [0, 1, \ldots 32]$. In [7], J.H. Derby proposes a novel state-space transformation method which moves computation and circuitry complexity out of the feedback loop. Let us consider the linear transformation of the state vector $\mathbf{x}(n)$ through a non-singular matrix $\mathbf{T}$

$$\mathbf{x}(n) = \mathbf{T}\,\mathbf{x}_t(n).$$

Both $\mathbf{T}$ and $\mathbf{T}^{-1}$ are defined over the field GF(2), thus we can rewrite the M-level look ahead as:

$$\begin{aligned}\mathbf{x}_t(n+M) &= \mathbf{T}^{-1}\mathbf{A}^M\mathbf{T}\,\mathbf{x}_t(n) + \mathbf{T}^{-1}\mathbf{B}_M\,\mathbf{u}_M(n);\\ \mathbf{y}(n+M) &= \mathbf{C}^M\mathbf{T}\,\mathbf{x}_t(n) + \mathbf{D}_M\,\mathbf{u}_M(n);\end{aligned}$$

The initial state $x(0)$ shall also be transformed in $x_t(0) = T^{-1}x(0)$. Given that $\mathbf{A}^M$ and $T^{-1}\mathbf{A}^M\mathbf{T}$ are similar matrices, there must exist an appropriate matrix $\mathbf{T}$ such that $\mathbf{A}_{Mt} = \mathbf{T}^{-1}\mathbf{A}^M\mathbf{T}$ is a companion matrix. In general, the matrix $\mathbf{T}$ is not unique and can be obtained selecting an arbitrary vector $\mathbf{f}$ such that the $\mathbf{k}$ vectors $\mathbf{A}^{kM}\mathbf{f}$ are linearly independent, and thus using those vectors as columns for the matrix:

$$[\mathbf{f}\;\; \mathbf{A}^M\mathbf{f}\;\; \mathbf{A}^{2M}\mathbf{f}\; \ldots\; \mathbf{A}^{(k-2)M}\mathbf{f}\;\; \mathbf{A}^{(k-1)M}\mathbf{f}]$$

Now, we can substitute blocks in Fig. 2 with:

$$\begin{aligned}\mathbf{A}^M &\to \mathbf{A}_{Mt} = \mathbf{T}^{-1}\mathbf{A}^M\mathbf{T}\\ \mathbf{B}_M &\to \mathbf{B}_{Mt} = \mathbf{T}^{-1}\mathbf{B}_M\\ \mathbf{C}_M &\to \mathbf{C}_{Mt} = \mathbf{C}_M\mathbf{T}\end{aligned}$$

Hence, $\mathbf{A}_{Mt}$ is in companion form, which implies an implementation with minimal loop complexity. On the contrary, $\mathbf{B}_{Mt}$ grows in complexity, but it can be fully pipelined being out of combinatorial loops.

Focusing now on CRC implementations, other parallelization methods have been proposed for software-oriented implementations. [8] proposes a fast implementation for processors. Look-ahead is applied to the serial implementation resulting in a byte-wise parallel implementation whose the feedback network is implemented as a look-up table plus shift-and-add operations. Another approach is proposed in [9, 10], where Galois field theory is applied to implement parallel subword GFMACs suitable for CRC calculation on customizable processor. Let us consider $a_{i\in[1\ldots n]}$ bits for the input message, and $\mathbf{A}(x) = \sum_{i=0}^{n} a_i x^i$ the corresponding polynomial form. Being $G(x)$ the M-order polynomial generator, the CRC is defined as:

$$CRC[A(x)] = (A(x)x^M) mod G(x)$$

It is possible to demostrate that CRC computation can be obtained working in parallel on M-bit message chunks W_i, such that:

$$CRC[A(x)] = \sum_i W_i \beta_i$$

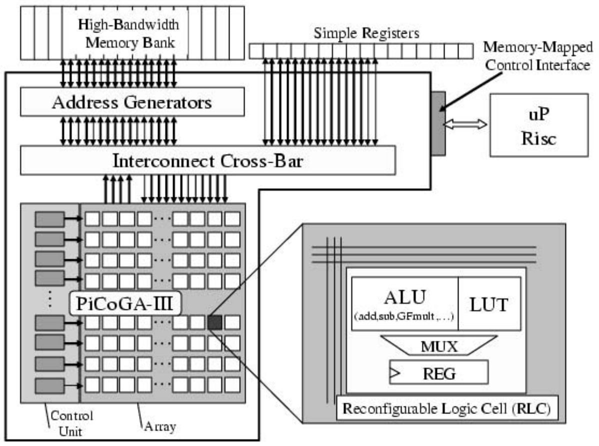

Figure 3. Simplified DREAM architecture

where β_i are N/M constants dependent on the message length N and the polynomial generators $G(x)$. CRC computation requires N/M GFMACs, and then to XOR the products.

3. DREAM Architecture Overview

DREAM adaptive DSP [5] is a dynamically reconfigurable processor featuring a Pipelined Configurable Gate Array (PiCoGA) directly accessing a local high-bandwidth memory sub-system. While a RISC core (the STxP70 core of STMicroelectronics) handles control and configuration of the platform, PiCoGA is responsible of data intensive computation. PiCoGA is a pipeline matrix of mixed-grain logic cells, featuring a 4-bit arithmetic/logic unit and a 64-bit look-up table. In addition, conditional operations, saturating and Galois Field arithmetic facilities are provided to improve the effectiveness of the computation. Routing architecture features 2-bit granularity segmented wires, although bit-wise interconnection is allowed with resource underutilization. Each PiCoGA row is the basic element for building a pipeline stage, under the supervision of a dedicated programmable pipeline control unit. Furthermore, PiCoGA provides 12 32-bit primary input ports and 4 32-bit output ports, and a 4-context internal configuration cache that allows exchanging the active layer in only 2 clock cycle. PiCoGA design is oriented to simplify system-on-chip integration (especially for processor-centric systems), featuring a fixed working frequency of 200MHz and an area occupation of $\sim 11mm^2$ in ST CMOS 90nm technology. Moreover, as part of DREAM, it allows to achieve efficient figures of merit like average 2 GOPS/mm^2 and 0.2 GOPS/mW, as shown in [5] for a heterogeneous spectrum of application kernels.

4. Implementation of the CRC on DREAM

This section presents the design exploration phase and the decision process we followed implementing the 32-bit CRC used in the Ethernet standard [12]. We analyzed the various approaches presented in section 2, trying to find that one best matching the DREAM computation paradigm. While we supposed to handle control parts (e.g. message start and stop) with the processor, we considered to map all the CRC computation on PiCoGA. As an additional consideration, we planned to exploit pipelining on PiCoGA as much as possible thus considering as appealing solutions all those which are not requiring pipeline breaks during the processing flow.

We selected the approach proposed by J.H. Derby in [7] since it allows exploiting pipelining without increasing the complexity of the feedback loop. Hence, the CRC is implemented as:

$$\begin{aligned} x_t(n+M) &= A_{Mt}\; x_t(n) + B_{Mt}\; u_M(n); \\ y(n+M) &= T\; x_t(n); \end{aligned}$$

As well as most of the coarse and mid grained reconfigurable fabrics, PiCoGA programming is performed through an assembly-like language. The programmer selects the appropriate instructions among the set offered by the architecture, in a way similar to that followed by DSP programmers with *intrinsics* or (*built-in functions*) instance. In our case, all the operations are defined over GF(2), thus additions are implemented by XORs. For that, we decided to massively use the 10-bit XOR operation which can be implemented in a single logic cell of PiCoGA.

The next step of our analysis is the selection of the look-ahead factor and the eventual partitioning on one or more PiCoGA operations, depending on both I/O bandwidth and computational resources available. In order to automate the investigation of the design space, we implemented a Matlab program which provides all the necessary matrices, starting from the size and polynomial generator of the CRC under construction. Furthermore, it maps the required matrices on 10-bit XORs, by an algorithm that reduces the number of required XORs detecting 10-bit common patterns among the rows of B_{Mt} and T. We also empirically analyzed the impact of the arbitrary vector f in the definition of the state-space transformation T, but we didn't find significant difference in the complexity of T (e.g. 1's per each row). As a result, we selected $f = [1\,0 \ldots\, 0]$.

We partitioned CRC on two PiCoGA operations: the first one implements the status update described by $x_t(n+M)$, and the second one implements the computation of $y(n+M)$ depending on the reached state. The main benefit of this approach is that we increase the resources available thus allowing greater look-ahead factors, hence the number of bits processed per cycle. On the other hand, this partitioning

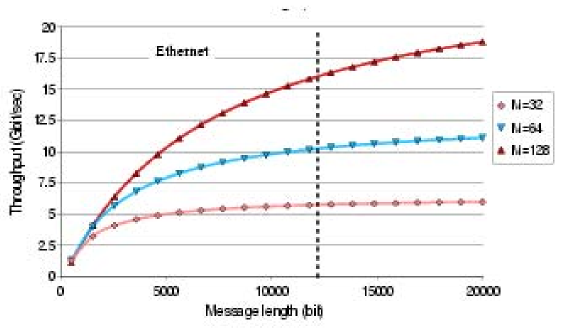

Figure 4. Throughput vs. message length (for a single message)

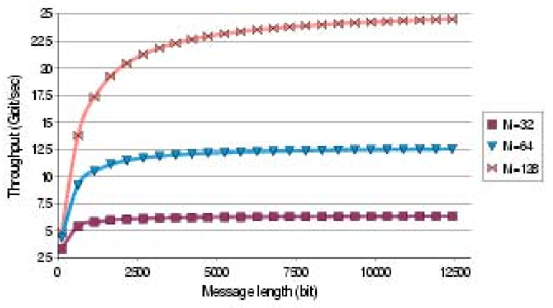

Figure 5. Throughput vs. message length (for 32 messages)

should not decrease performance because $y(n+M)$ is required only at the end of the message and it does not break the pipeline evolution. We generated PiCoGA operations for different values of M, finding that PiCoGA is able to elaborate up to 128 bit per cycle.

5. Experimental results

We have implemented on DREAM the 32-bit CRC defined for the Ethernet standard (but it is the same defined for MPEG-2) and we have analyzed as figures of merit the throughput and the computational efficiency.

Fig. 4 shows for different look-ahead factors (M) the bandwidth sustained with respect to the message length. The variation is due to the control overhead introduced by the processor and the pipeline break caused by the configuration switch when the second PiCoGA operation is triggered to provide the anti-transformed state. To give an idea of real cases, Fig. 4 also shows the message length windows supported in the Ethernet standard that range from 368 to 12000 bit. Bandwidths reported are achieved for the single message case, while Fig. 5 shows the case of interleaving multiple messages (in that case 32) as proposed in [13]. Message interleaving allows working concurrently on multiple messages reducing the impact of any configuration change. It is important to observe that in a message window compliant with Ethernet standard we can perform transfers at the Gbit/sec speed for M equal to 32, 64 and 128, thus allowing the user to explore additional trade-off point between area (resource utilization) and speed in the context of the final application.

	Speed-up		
Message Length	$M=32$	$M=64$	$M=128$
128	29.76	27.69	27.32
640	104.17	108.08	99.38
3200	255.96	347.68	389.93
7296	325.93	515.47	682.35
9856	345.23	571.79	805.55
12416	357.60	610.90	900.99

Table 1. Speed-up vs. Fast software CRC on RISC processor

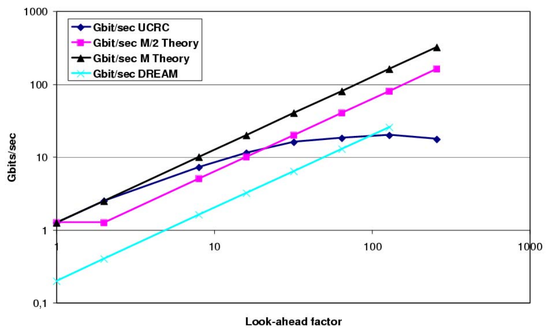

Figure 6. Application specific CRC: throughput vs. look-ahead factor

As a reference point, Table 1 shows the speed-up achieved by DREAM with respect to Fast software implementation on a RISC processor working at the same frequency. In [10], 2-3 cycles are required to compute the CRC for 128 bit message in a custom processor featuring 16 GF-MAC running at 200MHz. A rough analysis of performance figures in Table 1 also shows that the area increase due to a reconfigurable datapath, that can be estimated in $10\times$ the area of a basic processor, is returned by an adequate performance improvement, also for short messages.

A different comparison is proposed in Fig. 6, where we tried to compare our implementation with respect to some ASICs. We started our analysis from the Ultimate CRC (UCRC [14]) distributed in the OpenCore site, which allows implementing look-ahead factors from 2 to 512. We synthesized UCRC with Synopys Design Compiler on STM

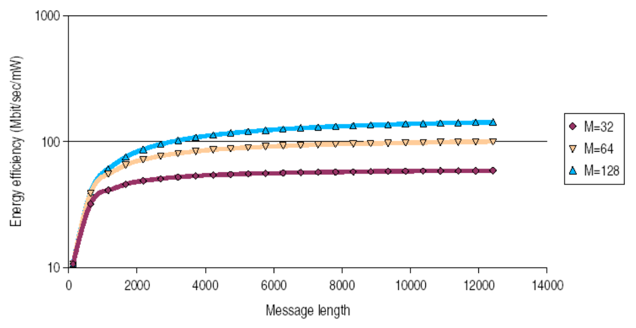

Figure 7. Energy efficiency

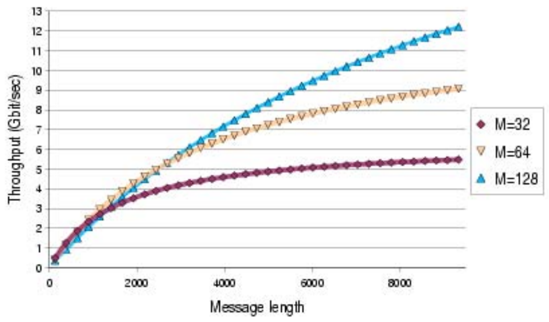

Figure 8. Throughput on 802.16e scrambler

CMOS LP 65nm technology, with different parallelization. We have also reported two theoretical bandwidths:

M theory , which refers to the bandwidth achievable applying to custom design the method proposed in [7],

M/2 theory , which refers to the bandwidth achievable applying to custom design the method proposed in [6].

For both, we consider the serial bandwidth achieved by UCRC ASIC synthesis, and then the theoretical speed-up factor is applied to obtain the curves. It should be observed that for small parallelization, performance of DREAM is limited by the fixed working frequency. Curves in Fig. 6 do not consider any communication overhead, but they analyze only the computational kernel. Hence, in the case of DREAM, as well as for the ASIC cases, we reported the case in which M bit are elaborated per cycle, without any configuration overhead, a condition that correspond to an infinite message. For $M = 128$, DREAM achieves a peak performance of ∼25 Gbit/sec, that is greater of the performance offered by UCRC.

Fig. 7 shows the energy efficiency of our approach for different message length and factors of parallelization. As a term of comparison, we can consider that a RISC processor requires for this CRC computation ∼400pJ/bit (independently from the message length), which is ∼5-60× more than on DREAM in 90nm technology [5].

To demonstrate the generality of the approach, we have implemented also a scrambler compliant with the 802.16e standard working with up to 128 bit in parallel, thus reaching the max output bandwidth achievable. The implementation requires a single operation on PiCoGA and Fig. 8 shows the throughput with respect to different look-head factors and block lengths. Although this second factor is probably not so interesting on wireless communication, mostly block based, it gives an idea of the performance that can be achieved when scrambling is used in cryptography as basis for stream ciphering.

6. Conclusions

In this paper we present the implementation of LFSR-based applications on the DREAM adaptive DSP, featuring a pipelined run-time programmable datapath. We considered the implementation of the 32-bit CRC defined for the Ethernet standard, achieving performance comparable with some ASIC implementations, with a peak of ∼25 Gbit/sec. This is in our opinion a very interesting result, since it is achieved with a software programmable solution and since it provides the bandwidth required for most of the standard considered.

References

[1] M2000 *Embedded FPGA* www.m2000.fr

[2] H. Singh et al. *MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications*, IEEE Transactions on Computers, May 2000.

[3] M. Vorbach, J. Becker, *Reconfigurable processor architectures for mobile phones* Proceedings on the IPDPS, 2003.

[4] R.R. Taylor, S.C. Goldstein *A High-Performance Flexible Architecture for Cryptography*, CHES 1999.

[5] Omitted for blind review

[6] T-B. Pei and C. Zukowski, *High-speed parallel CRC circuits in VLSI*,IEEE Trans. Commun. Apr 1992.

[7] J.H. Derby, *High-speed CRC computation using state-space transformations*, Global Telecom. Conf. 2001.

[8] G. Albertengo and R. Sisto, *Parallel CRC generation*, IEEE Micro,vol. 10, pp. 63-71, Oct. 1990.

[9] S. Roy, *A sub-word-parallel Galois field multiply-accumulate unit for digital signal processors*, IEEE ISCAS 2005.

[10] H.M. Ji, E. Killian *Fast parallel CRC algorithm and implementation on a configurable processor* IEEE International Conference on Communications, 2002. ICC 2002.

[11] J-S. Lin, C-K. Lee, M-D. Shieh, J-H. Chen *High-speed CRC design for 10 Gbps applications*, IEEE ISCAS 2006.

[12] IEEE Std 802.3, 2002 Edition

[13] J.J. Kong, K.K. Parhi *Interleaved cyclic redundancy check (CRC) code*, IEEE Conf. on Signals, Systems and Computers, 2003.

[14] OpenCore Ultimate CRC *http://www.opencores.org/projects.cgi/web/ultimate_crc/*