PAPER

High-Speed Fully-Adaptable CRC Accelerators

Amila AKAGIC[†a)], ***Nonmember and*** **Hideharu AMANO**[†b)], ***Member***

SUMMARY Cyclic Redundancy Check (CRC) is a well known error detection scheme used to detect corruption of digital content in digital networks and storage devices. Since it is a compute-intensive process which adversely affects performance, hardware acceleration using FPGAs has been tried and satisfactory performance has been achieved. However, recent extended usage of networks and storage systems require various correction capabilities for various CRC standards. Traditional hardware designs based on the LFSR (Linear Feedback Shift Register) tend to have fixed structure without such flexibility. Here, fully-adaptable CRC accelerator based on a table-based algorithm is proposed. The table-based algorithm is a flexible method commonly used in software implementations. It has been rarely implemented with the hardware, since it is believed that the operational speed is not enough. However, by using pipelined structure and efficient use of memory modules in FPGAs, it appeared that the table-based fixed CRC accelerators achieved better performance than traditional implementation. Based on the implementation, *fully-adaptable* CRC accelerator which eliminate the need for many non-adaptable CRC implementations is proposed. The accelerator has ability to process arbitrary number of input data and generates CRC for any known CRC standard, up to 65 bits of generator polynomial, during run-time. Further, we modify Table generation algorithm in order to decrease its space complexity from $O(nm)$ to $O(n)$. On Xilinx Virtex 6 LX550T board, the fully-adaptable accelerators occupy between 1 to 2% area to produce maximum of 289.8 Gbps at 283.1 MHz if BRAM is deployed, or between 1.6 - 14% of area for 418 Gbps at 408.9 MHz if tables are implemented in logic. Proposed architecture enables further expansion of throughput by increasing a number of input bits M processed at a time.

key words: *reconfigurable computing, FPGAs, cyclic redundancy checks, adaptability, accelerators*

1. Introduction

Cyclic Redundancy Check (CRC) is a well known error detection scheme used to detect corruption of digital content in digital networks and storage devices. Numerous applications use different CRC standards and algorithms for various types of network data transmissions, data compression (e.g. gzip and bzip2) and data encryption. However, since it is a compute-intensive process that adversely affects performance, it is often substituted with less efficient error detection schemes. It plays an important role in, for example, implementation of iSCSI protocol in Storage Area Networks (SANs) for detecting errors which occur between protocol transitions. It is common practice to disable iSCSI digests in order to decrease latency, thus the network must rely on other mechanisms to detect corrupted data, such as TCP and Ethernet error detection mechanisms. Unfortunately, these mechanisms use simple TCP checksum, which can result in undetected data corruption. This can lead to various problems such as failed integrity check of a database. Therefore, it is desired to (1) reduce the computational burden, (2) make architecture generic enough to support a variety of applications, (3) make architecture scalable so it can process arbitrary number of data input (4) achieve significant improvements in throughput and (5) make it area efficient.

The goal of traditional methods for designing CRC accelerators is acceleration of a specific application. In such accelerators, the resulting CRC value is determined by the CRC standard deployed by an application, which is usually fixed at the design time. We call these accelerators *non-adaptable*. On the other hand, *adaptable* CRC accelerator has ability to generate CRC for a variety of CRC standards and thus support a wide range of applications. They eliminate the need for many non-adaptable CRC implementations. The fully-adaptable CRC accelerator has ability to process arbitrary number of input data and generates CRC for all currently defined CRC standards during run-time.

In this paper, we describe the design and implementation of fully-adaptable CRC accelerators based on a table-based algorithm which is suited for the flexible implementation. Although the table-based algorithm has been used in software, it has been rarely implemented in hardware as its performance is believed to be lower than traditional implementation. We prove that this approach can be successfully implemented on an FPGA and achieve significant performance improvements over related work. Our contributions in the paper are as follows:

1. A design of CRC accelerator with sufficient performance and reasonable resource utilization using a table-based algorithm is proposed.
2. Based on the above design, a *fully-adaptable* CRC accelerator is proposed by integrating algorithm for generating CRCs and algorithm for generating contents of Tables. Resulting architecture generates CRC for any known CRC standard during run-time. It achieves throughput of up to 418.8 Gbps, when $M = 1024$.
3. We modify Table generation algorithm in order to decrease its space complexity from $O(nm)$ to $O(n)$.
4. Design of our architectures guarantees scalability/expandability by processing arbitrary number of input data M at minimal area cost. In order to show ef-

Manuscript received October 16, 2012.
Manuscript revised February 7, 2013.

[†]The authors are with the Department of Information and Computer Science, Keio University, Yokohama-shi, 223–8522 Japan.
a) E-mail: amila@am.ics.keio.ac.jp
b) E-mail: hunga@am.ics.keio.ac.jp

DOI: 10.1587/transinf.E96.D.1299

ficiency of our architecture in terms of area utilization and throughput, we design five implementations, where $M \in 64, 128, 256, 512, 1024$.

The organization of this paper is as follows: we start out with a brief overview of CRC algorithms, their implementations and related work in Sect. 2. In Sect. 3 we give a high level overview of the new architecture and describe its components in detail. In Sect. 4 we describe implementation on an FPGA. In Sect. 5 we present area utilization, clock speed and throughput after synthesis, place and route on Xilinx Virtex 6 LX550T, analyze and compare our results with related works. We conclude the paper in Sect. 6.

2. CRC Algorithms and Their Implementation

2.1 Overview of CRC Algorithms

CRC is calculated by performing long division operation between input message and a generator polynomial. At first, a message M is multiplied by x^w, which is equivalent to shifting to left by a polynomial length (w bits). This value is then divided by generator polynomial $G(x)$, and the remainder is called CRC value as shown in (1). The CRC is affixed to the original message M and transmitted to a receiver.

$$CRC(M) = M(x) \times x^w \bmod G(x) \tag{1}$$

An input data or a message M is treated as a polynomial, where bit values are coefficients of a dummy variable x. The coeffcients are all either 0 or 1, while the power of x corresponds to the bit position. For example, the message "01010100" is represented as $0 \times x^7 + 1 \times x^6 + 0 \times x^5 + 1 \times x^4 + 0 \times x^3 + 1 \times x^2 + 0 \times x^1 + 0 \times x^0$. If the length of M is defined as l, then M can be represented as:

$$M(x) = m_{l-1}x^{l-1} + m_{l-2}x^{l-2} + \ldots + m_0 \tag{2}$$

where m_{l-1} is the most significant bit of a message M and m_0 is the least significant bit. A generator polynomial of length w is represented in the same manner:

$$G(x) = g_w x^w + g_{w-1}x^{w-1} + g_{w-2}x^{w-2} + \ldots + g_0 \tag{3}$$

Due to interference during transmission, data might be corrupted during transport. Errors will be detected on a receiver's side by performing similar process as a sender. At first, a receiver will remove received CRC, then it will perform long division operation with the same generator polynomial specified by a protocol used. Then it will compare received CRC value and its own. Any discrepancy between these two values indicates the presence of transmission errors in the received pair. In this case, a receiver will discard the message and request re-transmission of the data.

2.2 CRC Algorithms

In this section we overview three mostly used approaches for generating CRC with emphasize on a newly proposed ones [1]. We highlight major overheads of these approaches.

2.2.1 Bitwise Approach

In this approach, CRC is calculated with N shifts and XOR operations for N-bit input message, which makes this algorithm computationally intensive. Early hardware implementations were based on this algorithm, which was implemented using linear feedback shift registers (LFSR) [2]. Input message were fed *serially* into a circuit, hence if implemented on an FPGA throughput would be limited by operating frequency of an FPGA (e.g. 200 MHz limits throughput to 200 Mbps, thus will not be suitable for high-speed links). Some level of parallelism must be introduced to gain higher throughput.

2.2.2 Table Based Approach

The main idea behind this approach is to pre-compute remainders for a specific input and store them into a table. Widely used algorithm with this approach is known as Sarwate algorithm [3]. This algorithm has been designed when computer architectures supported XOR operation with only *eight* bits, but it is still used today in low-performance implementations. In general, to process a message M of length l, Sarwate algorithm requires a table of $2^l \times (w-1)$ precomputed remainders. Today's processors support operations with 32 and 64 bit values, thus if this algorithm is to be extended it would require lookup tables of $2^{32} \times (w-1)$ and $2^{64} \times (w-1)$ for processing 32 or 64 bit input data, respectively. These tables cannot fit into a cache so their contents have to be constantly fetched from the main memory, causing significant performance drop.

2.2.3 Multiple Tables Approach

Looking to overcome limitations of processing only 8 bits of data at a time, two new algorithms have been proposed and evaluated in [1]: *Slicing–by–4* and *Slicing–by–8*. The main advantage is that they can read and process *arbitrarily large amounts* of data at the time.

The *Slicing–by–4* algorithm reads and processes 32 bits at a time, and it doubles the performance of existing implementations of Sarware algorithm. The algorithm deploys *four* tables with pre-computed remainders, which are accessed in parallel by using *four* 8 bit slices. Each table requires 1 KB of data in the cache (256×32 bits values for 33 bit generator polynomial), thus *Slicing–by–4* requires 4 KB of data in memory. This amount of data can easily fit in today's cache, resulting in faster execution, but it is still limited by the speed of a processor.

Similarly, the *Slicing–by–8* triples the performance of the Sarware algorithm. It reads and processes 64 bits at a time, and it deploys *eight* look-up tables accessed by *eight* 8 bit slices. The algorithm requires 8 KB of data in the cache.

Table 1 A list of parameters defined by a CRC standard shown on the example of CRC32c standard.

Parameter	Value
Name	CRC-32C
Width	32
Poly	1EDC6F41
Init	FFFFFFFF
RefIn	True
RefOut	True
XorOut	FFFFFFFF
Check	E3069283

2.3 CRC Standard

The content of a pre-computed table depends on the specific parameters of a CRC standard, as well as on the position of a byte in the input stream that is being processed. A CRC standard is defined by 8 parameters, as shown on the example of CRC32c standard in Table 1.

Width defines width of the algorithm. *Poly* defines hexadecimal value of a generator polynomial, with top bit omitted, since its value is always 1. *Init* defines initial value of a CRC register used only in the first iteration of a CRC algorithm. An input message is reflected before performing long division operation if parameter $RefIn$ is true, e.g. 8 bit value will be processed with bit 7 being treated as the least significant bit (LSB) and bit 0 as most significant bit (MSB). If $RefIn$ is false, input bits will not be reflected. Similarly, if $RefOut$ parameter is defined as true, the remainder is reflected before writing it into the table. *XorOut* parameter is defined as a hexadecimal value and it is used in a final stage before the value is returned as the official checksum. *Check* parameter is defined as hexadecimal value that represents CRC value of the ASCII string "123456789". It is used as a weak validator of implementations of the algorithm. The parameters *Name* and *Check* are not of any use for our implementation, thus we omit them.

2.4 Related Work

Implementations of CRC accelerators with fixed CRC Standards are presented in [4]–[6] and [7]. In [4] authors identified a recursive formula in serial implementation from which they derived parallel implementation, achieving maximum of 4.38 Gbps while processing 32 bits at a time, occupying only 162 LUTs. In [5] authors design a circuit with two parallel calculation units, capable of processing 32 and 64 bits of input in two different implementations. They operate on 180 MHz in 0.35 micron technology, with maximum throughput of 5.76 Gpbs for 32 and 64 bits processed every clock cycle. In [6] CRC is implemented in a pipeline structure, with a number of successive multiplications and divisions. The maximum reported throughput for processing 32 bits at a time with 16 bits generator polynomial is 4.585 Gbps with clock of 153.84 MHz, and 2.838 Gbps for processing 32 bits with 32 bits polynomial, with clock of 95.23 MHz on Altera FLEX 10KE device. In [7], the implementation of parallel LFSR-based applications on an adaptive DSP featuring a Pipelined Configurable Gate Array (PiCoGA) has been presented, with Ethernet's 32 bits CRC as a test-case. PiCoGA is integrated in the embedded digital signal processor based on run-time recongurable technology (named DREAM), featuring a working frequency of 200 MHz. On the target architecture, CRC circuit achieves up to 25 Gbps throughput with a parallel LFSR processing 128 bits at time.

There is relatively little work in the area of fully-adaptable CRCs on FPGAs. We found only two other hardware implementations [8], [9] that can support very limited number of generator polynomials. The re-generation in [8] is achieved with Galois Field Multiplication and Accumulation (GFMAC) with *soft-coded* and *hard-coded* generator polynomials. Soft-coded implementation is much slower than fixed hard-coded counterpart. The maximum throughput of soft-coded design with 32 bit CRC is 1.3 Gbps for a 128 bit message. Unfortunately, the reconfiguration time is not provided. The implementation [9] can process variable number of 32 bit generator polynomials, and can be modified to support 64 bit, but cannot support both at the same time in one circuit. The maximum throughput for processing 32 bits with a 32 bit generator polynomial is 4.92 Gbps. It is generic in its design, thus it can be scaled to 64, 128 or 256 bits, with maximal theoretical throughput of 40 Gbps at 256 bits. The reconfiguration time is not very specific - under 1 μs.

The mostly used software solution [1] achieves maximum of 3.6 Gbps for processing 64 bits at a time on Intel Pentium 1.7 GHz. In [10] we measured performance of the same algorithm on the state-of-the-art Xeon 3 GHz processor with 4 MB of L2 cache, and the throughput was 9.58 Gbps while processing 512 bits at a time, in idealized conditions without cache misses. There is no research about CRC circuits which support 64 bit generator polynomials.

3. Design of Fully-Adaptable Accelerator

3.1 Design Overview

In order to support variable number of CRC Standards, we propose the structure consisting of a non-adaptable CRC accelerator core and Table Generation Module (TGM) as shown in Fig. 1. From the viewpoint of an application, the total system is treated as a CRC IP Core. The accelerator core consists of CRC Generation Module (CGM) and tables with pre-computed values. The CGM calculates CRC for provided data input every clock cycle by accessing pre-computed remainders stored in tables in parallel. The TGM generates pre-computed remainders for a specified generator polynomial G, and stores them into tables. The IP Core Interface is responsible for managing generation of remainders, accessing and storing them from/into tables, and managing input/output buffers. Two main modules are not executed in parallel in order to maintain data consistency of ta-

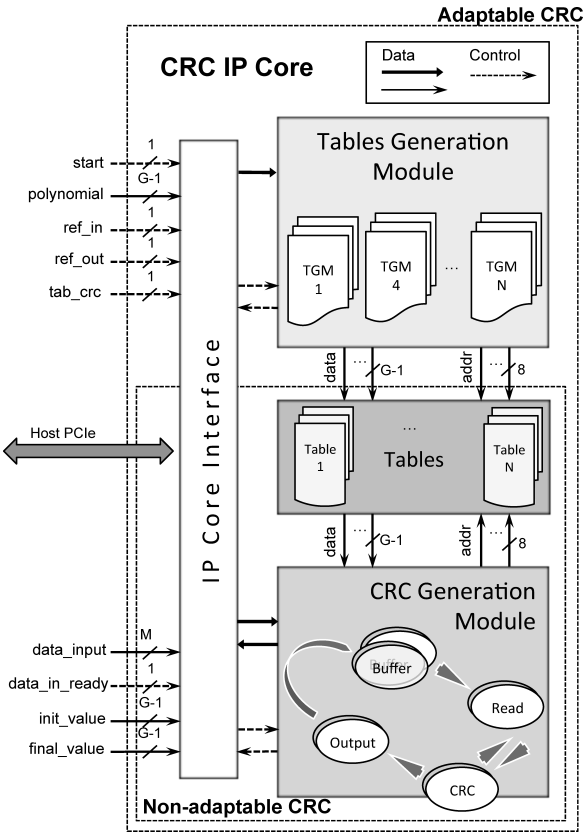

Fig. 1 Design overview of the non-adaptable (accelerator core) and fully-adaptable CRC accelerators. The *fully-adaptable accelerator* consists of two main modules: a CRC Generation Module and a Tables Generation Module (TGM), as well as supporting Tables and IP Core Interface for communication with external devices. Control signals *start*, *tab_crc* and *data_in_ready* represent request for service, type of service (Table regeneration or CRC generation) and availability of data input, respectively. Control signals *ref_in* and *ref_out* represent values of *RefIn* and *RefOut* parameters as explained in Sect. 2.3.

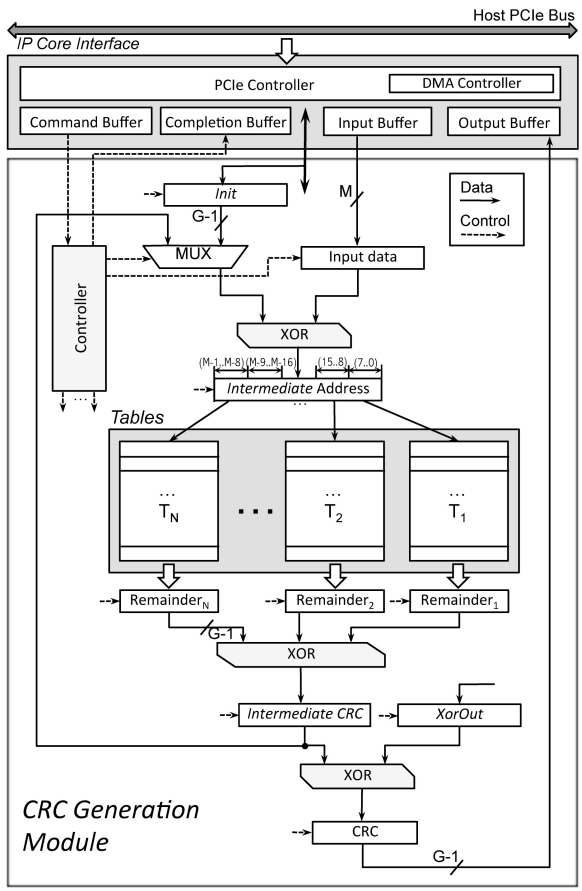

Fig. 2 The generic architecture of CRC Generator Module - CGM and accompanying IP Core Interface.

bles. The fully-adaptable accelerator accepts variable width generator polynomial G up to 65 bits, while different input stream widths M require different FPGA implementation for each. The most significant bit in a polynomial G is always considered to be 1, thus the input polynomial is always $G-1$. The number of tables N depends on an input data M and it is equal to $M/8$.

3.2 CRC Generation Module

Figure 2. shows the architecture of CRC Generator Module and IP Core Interface.

In the first iteration, *Intermediate Address* is formed by XORing input data with initial value (*Init*), while in other iterations *Intermediate CRC* is used instead of *Init*. *Intermediate Address* is then sliced into N eight bit slices, which are used as addresses to N Tables. The number N dependents on the number of input data M and it is equal to $M/8$. Then, Tables provide N Remainders, which are XORed to form *Intermediate CRC*. When the end of the Input Buffer is reached, *Intermediate CRC* is XORed with the final value (*XorOut*) and then stored in the Output Buffer.

For fully-adaptable implementation, the CGM must support variable number of CRC Standards. There are four parameters defined by a CRC Standard which affect execution of CGM: *Width*, *Poly*, *Init* and *XorOut*. We fix datapath's width to maximum number of bits in *Width* in order to support variable width generator polynomial. CRCs with higher degree polynomials are still not used in practice, thus there is no need to implemented them yet. The CGM indirectly depends on the *Poly* through TGM which generates reminders in Tables (which will be discussed in the next section). In the fully-adaptable architecture, the IP Core Interface permits usage of the CGM only until Tables are ready, thus the CGM doesn't have to check if Tables are ready or not.

The CGM is implemented in a pipeline with three stages as following. In the first stage, M bits are read from the Input Buffer and stored into a temporary register. This value is then XORed with either Init or previous *Intermediate CRC*, depending on the iteration. In the second stage, *Intermediate Address* is sliced into N slices and used to access N remainders from N tables. These remainders are then XORed to form *Intermediate CRC*. In the third stage, *Intermediate CRC* is XORed with XorOut and stored into the Output Buffer. The latency of CGM module is three cycles,

but CRC is generated every cycle (the throughput is one cycle).

Fully-adaptable CGM's architecture has the following aspects to provide adaptability:

- data-path is extended to maximum number of bits in *Width*,
- structure of tables $T_N..T_1$ is extended to maximum $2^{Slice} \times Width$, and Tables are *read-write*, instead of previously read-only,
- values in Tables *and* resulting CRC are aligned to right, while in the case when $G < 65$ bit positions $64-(G-1)$ are filled with zeros, since XORing a value with zeros will result with the value itself,
- Init and XorOut values are programmable, instead of hard-coded in the circuit,
- controller is modified in order to support integration with TGM and IP Core Interface.

3.3 Tables Generation Module

For all table-based CRC algorithms tables are generated by a separate algorithm. The results are used to form a data-set in the program for generating CRCs. In software implementations, tables are generated sequentially and independently from each other. In our design, we integrate these two algorithms and in this section we discuss the architecture of Tables Generation Module - TGM. This architecture enables support for a variable number of CRC Standards, as well as processing arbitrary number of bits at a time. We provide a pseudo-code for generating remainders in Fig. 3 based on the description in [11].

We present a general block diagram of our single TGM in Fig. 4a). Every clock cycle, the counter generates a message ranging from 0 to 2^{Slice} (line 2 in Fig. 3). This message then passes through the input reflection unit (line 4) or directly to the remainder generator (line 9 - 15), depending on a CRC standard. The input reflection unit reflects message bits by swapping them around its center. Prior to forwarding a message to the remainder generator, the message is shifted to left by $Width - Slice$ bits (line 8), depending on the width

```
1:  for Offset = 1 → N do
2:      for i = 0 → 2^Slice do
3:          if RefIn = true then
4:              input ← reflect(i, Slice)
5:          else
6:              input ← i
7:          end if
8:          r ← input << (Width − Slice)
9:          for j = 0 → Offset ∗ Slice do
10:             if r ∧ input then
11:                 r ← (r << 1) ⊕ Poly
12:             else
13:                 r ← r << 1
14:             end if
15:         end for
16:         if RefOut = true then
17:             r ← reflect(r, Width)
18:         end if
19:     end for
20: end for
```

Fig. 3 Pseudocode for generating contents of Tables based on the description in [11]. The $Offset$ is the position of a byte in an input message M that is being processed.

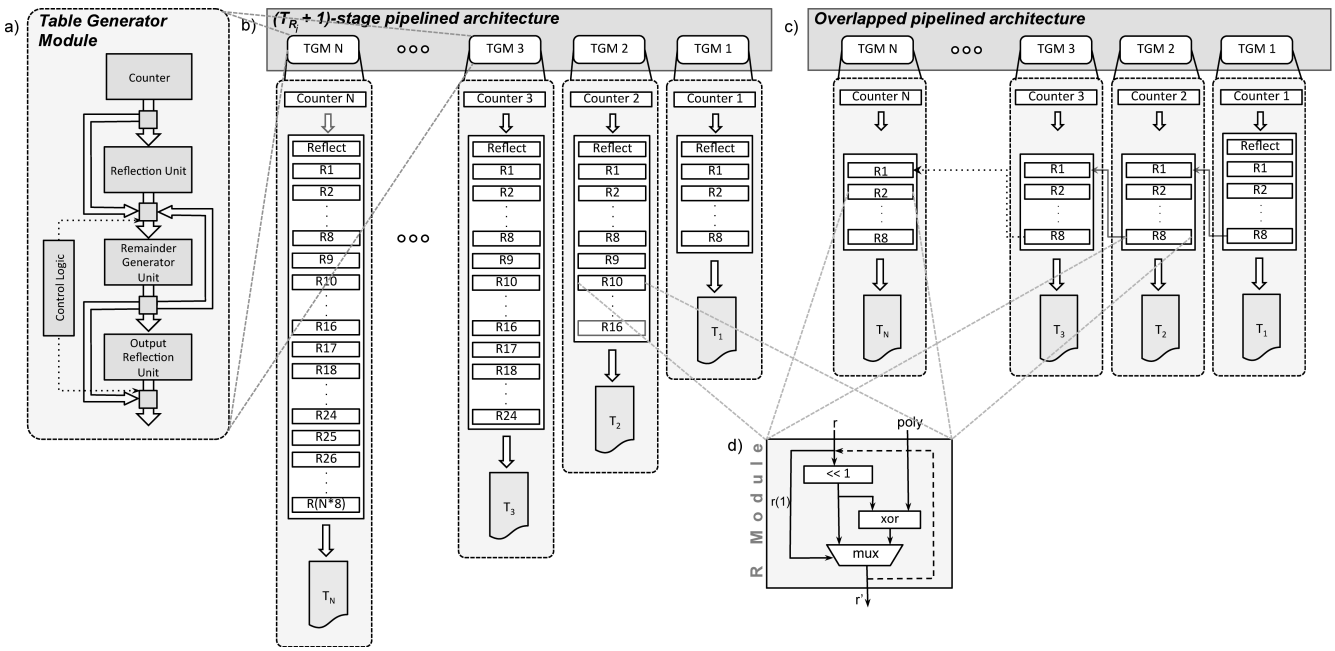

Fig. 4 a) A general block diagram of a single Table Generation Module - TGM; b) A schematic of $(T_{R_i}+1)$-*stage* pipelined architecture; c) A schematic of *Overlapped* pipelined architecture (preferred design); d) Architecture of a single operation of Remainder Generator Unit called *R Module*. The pipeline registers are placed after all *Reflect* and *R Modules* in both architectures.

of a data-path and a slice.

The remainder generator unit performs long division operation, consisting of series of sequential operations. Operations are inter-dependent from the results of a previous operation, thus it is impossible to execute them simultaneously. We described a single operation and defined it as *R Module* (Fig. 4d). Number of cycles required to generate one remainder depends on the offset of a byte in input message M. At the end, the remainder is reflected or forwarded to output. This unit determines the speed and the area of the circuit.

The TGM is designed to be independent of the width of a generator polynomial, thus the TGM in Fig. 1. does not require additional input for the width of a generator polynomial. The polynomial is aligned to left, while in the case when $G < 65$ bit positions from $(64 - (G - 1))$ to 0 are filled with zeros. This feature significantly simplified design of TGM.

In the next two sections we will describe two possible architectures for TGM and analyze them.

3.3.1 $(T_{R_i} + 1)$-Stage Pipelined Architecture

Straight-forward implementation of TGM in hardware is to design N circuits, and generate contents for each table in parallel. In order to generate one remainder each cycle we consider multiplying R Modules and using pipelining to exploit the intrinsic parallelism. The schematic of this architecture is presented in Fig. 4 b).

The number of R Modules depends on the width of input data M. In this paper, we explored possibility of processing $M \in 64, 128, 256, 512, 1024$. Thus, the total number of R Modules *per every consecutive table* $i \in 1, 2, 3, \ldots N$ is calculated with the following formula:

$$T_{R_i} = Offset_i \times Slice\_length, \tag{4}$$

where $Offset$ is the position of a byte in an input message M that is being processed. However, the space complexity of this approach is $O(nm)$ when we consider the total number of R Modules *per algorithm*:

$$T_{R_N} = \sum_{i=1}^{N} T_{R_i} = Slice\_length \times \sum_{i=1}^{N} Offset_i. \tag{5}$$

Consequently, the minimum required number of stages in the pipeline for the first table is 9 stages, and every consecutive table requires 8 additional stages (shown in Fig. 4b). The first remainder is generated after T_{R_i} clock cycles, followed by other remainders generated in every clock cycle.

By our estimations (detailed in Sect. 4.), the architecture with 64 and 128 tables couldn't fit in a moderate size FPGA, while the architecture for smaller number of tables would occupy substantial amount of area on a modern FPGA. In next section we introduce the method to reduce space complexity by overlapping specific operations.

3.3.2 Overlapped Pipelined Architecture

We noticed that we can reduce number of R Module *per table* to only 8 modules, by forwarding last non-reflected value from the R8 Module as an input to the R1 Module in the next TGM. The concept is illustrated in Fig. 4c). Doing so, we significantly reduced number of R Modules, and we simplified architecture of TGMs from 2 to N. Architecture of the first TGM was modified to output non-reflected value from R8 Module, and corresponding logic was added to connect this value to the following TGM. First non-reflected value can be forwarded in T_{R_i} clock cycle, just before output reflection. This also means that the second TGM can start processing one clock cycle before the first table outputs its first remainder. Initial reflection is not necessary for tables 2 to N, since the counter value was reflected in the first table. We keep counters in other generators to generate addresses for corresponding remainders. Thus, *TGM 1* has nine pipeline stages, while other TGMs have only eight. The input reflection is not needed due to the forwarding of non-reflected values from *TGM 1*. Thus, the latency of *TGM 1* is nine cycles, while the latency of *TGM 2* to *8* is eight cycles.

Any additional table will add the latency of 8 cycles to a number of cycles from the start of calculation. The total latency is the same as in $(T_{R_i} + 1)$-stage pipelined architecture, but the amount of resources used is significantly reduced.

In this architecture, the total number of R Modules *per algorithm* is calculated with:

$$T_{R_N} = N \times 8, \tag{6}$$

where $N \in 8, 16, 32, 64, 128$. Thus, the space complexity is reduced from $O(nm)$ to $O(n)$.

3.4 Effects of Architecture's Scalability

As we mentioned earlier, we explored possibility of processing arbitrary number of input data $M \in 64, 128, 256, 512,$ 1024 bits and in this section we discuss effects on our architecture. The first series of Table generations read $M =$ 64 bits (*Slicing–by–8*) and require eight TGMs. Every other extension in the number of processed bits doubles the number of required TGMs. In order to keep resource utilization at minimum, we decided to re-use these eight modules for other implementations. The basic idea is shown in Fig. 5. Clock cycles are presented as numbers on the left side from a table, while numbers inside the tables represent position of the remainder in a table. For the second series of Table generations (*Slicing–by–16*), which read 128 bits, we modified *TGM 1* to accept non-reflected values from *TGM 8*. This is essentially forwarding non-reflected values again to *TGM 1*. The only problem is that *TGM 1* can start generating remainders for Table 9 after 264 clock cycles, while *TGM 8* generates first non-reflected value in cycle 64. This means that non-reflected values cannot be forwarded directly to *TGM 1*. Thus, we decided to introduce a temporary table for *TGM*

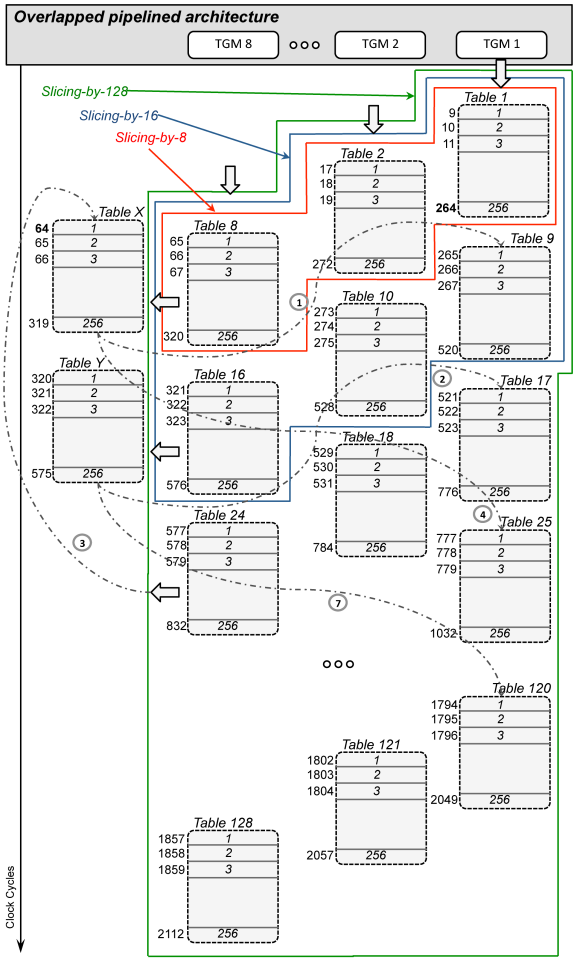

Fig. 5 All five implementations use only eight TGMs for generating contents of 1 - 128 Tables.

8's non-reflected values - *Table X*. *TGM 1* will start generating remainders for Table 9 just after it finishes generation of remainders for Table 1. In order to generate remainders for Table 9, *TGM 1* will read input values from Table X (shown with line 1 in Fig. 5.) instead from the counter shown in TGMs basic architecture. All the following tables will start executing with 8 clock cycles latency compared to a previous table.

For the third series of Table generations, which read 256 bits, we had to introduce another temporary table - *Table Y*. *TGM 8* generates new non-reflected values for Table 9. before *TGM 1* reads all the values from the first temporary table, thus these values will be temporarily stored in Table Y. Then, Table Y will be used for generation of Table's 17 remainders (line 2), while Table 24 will store its non-reflected values in Table X (line 3). Input into Table 25 will be non-reflected values stored in Table X (line 4), and so forth. Table Y will be lastly re-used by Table 120, in order to calculate remainders for total number of 128 tables for *Slicing–by–128*. This will be 7th usage of this table during re-generation of tables for *Slicing–by–128* implementation (line 7).

Table X and *Table Y* are implemented as dual-port BlockRAMs, thus there are no pipeline stalls in any *TGM*. Due to its pipelined design, *TGM 1* can start reading first non-reflected value from *Table X* in 257 cycle, instead of 319 cycle. Thus, the first value in Table 9 is generated in cycle 265. This technique also applies to remaining tables.

In order to support this idea, data-path and controller of each of these series of Table generations is significantly different from each other. Resource utilization is therefore kept minimal, as will be discussed in Sect. 4. This architecture enables further expansion of throughput with minimal resource utilization.

3.5 The IP Core Interface

When an application requests CRC IP Core service, the request is first delivered to the IP Core Interface (as seen in Fig. 2). The Interface is connected with external processing systems through the PCI Express bus. It consists of a PCI Express Controller, command buffer, completion buffer, input buffer and output buffer. A command request is delivered to the CRC IP Core through the command buffer. There are two types of command request: Table re-generation and CRC generation. After the command is processed, a completion result is delivered to the host CPU through the completion buffer. The input buffer is used to store data fetched from the main memory by DMA. Similarly, output buffer is used to store data that will be transferred into the main memory by DMA.

When a user invokes a routine in the CRC IP Core device driver, it creates a command request corresponding to the user's request and stores it into the command buffer. The IP Core Interface then reads the command request and performs operations corresponding to the request. After the request is processed, the IP Core Interface creates the completion result and stores it in the completion buffer. The IP Core Interface then interrupts the host CPU to report completion of a request. The CPU reports the result to the user program.

4. FPGA Implementation

We designed a prototype implementation using the Xilinx Virtex 6 LX550T device (xc6vlx550t-2ff1760). The design was written in VHDL and Xilinx's ISE 12.4 design environment was used for all parts of a design flow including synthesis, mapping and place and route. The behavioral correctness of each circuit has been manually checked through a series of simulatios in Xilinx ISim 12.4. The advantage of implementing table-based CRC architecture on an FPGA comes from FPGA's ability to access look-up tables in parallel, thus by increasing number of processed bits at a time we can maintain increase in throughput of an architecture.

After the implementation of TGMs on an FPGA, we came to the conclusion that *$(T_{R_i} + 1)$-stage pipelined architecture* is area consuming, when we consider re-generating remainders for 32, 64 or 128 tables in parallel. We measured number of resources for a single *R Module* which is

8 slices or 32 LUTs. Then, we implemented two versions of this architecture with $M = 64$ and $M = 128$, with 8 and 16 TGMs/Tables, respectively. We also roughly estimated number of resources used for R Modules in $M \in 64, 128, 256, 512, 1024$ architectures. The results are presented in Table 2. The estimated resources were very close to the measured resource utilization, thus we concluded that the majority of resources were used for R Modules. Most likely, architectures with $M = 512$ and $M = 1024$ would not fit on this fairly large FPGA chip, while architecture with $M = 256$ would occupy substantial amount of resources - 39% on the Xilinx Virtex 6 LX550T.

For implementation of TGM, *Overlapped pipelined architecture* is used. Maximum of eight TGM's are used in *all* five implementations, where $M \in 64, 128, 256, 512, 1024$. This feature ensured minimal resource usage for a number of input bits at a time, and also enabled further expansion of the architecture.

Number of tables grow proportionally with the number of bits processed at a time, thus it is important to choose storage elements with fastest access time. To measure performance of this architecture, we implemented Tables in a) BRAM and in b) logic. Tables are read-write, but writing and reading operations are not overlapped in order to maintain consistency of Tables. For implementation *a)* Tables are implemented as RAM modules in Block RAM.

Table 2 Resource utilization of R Modules in $(T_{R_i} + 1)$-stage pipelined architecture for $M \in 64, 128, 256, 512, 1024$. Column three represents roughly estimated resources utilization based on the resources required by a single R Module, while column four represents actual resource utilization on the Xilinx Virtex 6 LX550T.

Xilinx Virtex 6 LX550T	TGMs (Tables)	LUTs **est.**	LUTs
$M = 64$ bits	8	9.2k (2%)	10.5k (3%)
$M = 128$ bits	16	34.8k (10%)	37.6k (10%)
$M = 256$ bits	32	**135k (39%)**	-
$M = 512$ bits	64	**532k (154%)**	-
$M = 1024$ bits	128	**2113k (614%)**	-

5. Evaluation

5.1 Non-adaptable CRC Accelerator Core

First of all, we evaluated the performance and resource usage of the CRC accelerator core when it is used as a fixed non-adaptable one. In Table 3, Slicing–by–N_{32} and Slicing–by–N_{64} show the case when the resulting CRC value is 32 and 64, respectively.

As shown later, the throughput achieved is superior or comparable to the traditional LFSR implementation. Thus, it appears that the accelerator with table-based algorithm achieved enough performance with reasonable cost.

5.2 Fully-Adaptable CRC Accelerator

5.2.1 Throughput and Resource Usage

In Table 4, the performance and resource usage of fully-adaptable CRC accelerators are shown. In this implementation, *Overlapped pipelined architecture* is used in TGM, and Tables are implemented in: a) BRAM and b) logic. Note that the *fully-adaptable* architecture is capable of generating remainders for any known CRC standard, up to 65 bits of generator polynomial, during run-time. Its usability is much broader than non-adaptable architecture.

The throughputs of four implementations of non-adaptable and fully-adaptable CRC accelerators are shown in Fig. 6. The maximum throughput was achieved with fully-adaptable architecture with Tables implemented in logic (Table 4b). It is 418.8 Gbps when M = 1024, and it

Table 3 Resource utilization of *non-adaptable* Slicing–by–N_{32} and Slicing–by–N_{64} algorithms on the Xilinx Virtex 6 LX550T, where $N = M/8$, $M \in 64, 128, 256, 512, 1024$, and "32" and "64" represent the width of a resulting CRC value (related to 33 and 65 generator polynomial). Tables are implemented in BRAM.

	Slicing–by–N_{32}					**Slicing–by–N_{64}**				
Xilinx Virtex 6 LX550T	$M = 64$	$M = 128$	$M = 256$	$M = 512$	$M = 1024$	$M = 64$	$M = 128$	$M = 256$	$M = 512$	$M = 1024$
LUTs (max 343680)	205	404	676	916	1180	540	780	1186	1328	2230
Fully used LUT-FF pairs	159	359	493	714	979	458	698	849	946	1824
BRAM (max 632)	8	16	32	64	128	8	16	32	64	128
Max. operating freq. (MHz)	469.14	431.86	332.92	332.92	347.98	468.02	430.91	332.36	332.36	347.37
Throughput (Gpbs)	29.32	53.98	83.23	166.46	347.98	29.25	53.86	83.09	166.18	347.37

Table 4 Resource utilization of *fully-adaptable* CRC accelerator, where $N = M/8$ and $M \in 64, 128, 256, 512, 1024$. We present two implementations with Tables implemented in a) BRAM and b) in logic on the Xilinx Virtex 6 LX550T.

	a) Slicing–by–N [BRAM]					**b) Slicing–by–N [logic]**				
Xilinx Virtex 6 LX550T	$M = 64$	$M = 128$	$M = 256$	$M = 512$	$M = 1024$	$M = 64$	$M = 128$	$M = 256$	$M = 512$	$M = 1024$
LUTs (max 343680)	3398	3405	3949	5119	6774	5571	9151	14861	26114	48756
Fully used LUT-FF pairs	3082	2977	3481	3604	3617	3306	5084	6185	10839	17218
BRAM (max 632)	8	16+1	32+2	64+2	128+2	–	–	–	–	–
Max. operating freq. (MHz)	352.37	345.31	337.93	321.31	283.1	443.9	417.8	414.61	415.14	408.9
Throughput (Gpbs)	27.8	44.2	86.50	164.51	289.8	28.41	53.47	106.14	212.6	418.8

is up to 31% higher than adaptable architecture with Tables implemented in BRAM. Compared to non-adaptable CRCs, the configuration of BRAM was changed from read-only to read-write, thus the total critical path was increased to approximately 1.78 ns, and between 0.512 to 0.696 ns of routing delay. This is why fully-adaptable CRCs with BRAM show decrease in throughput, but they can still support most demanding applications. *Slicing–by–*N_{32} and *Slicing–by–*N_{64} exhibit almost the same trend in throughput, because their architecture is not noticeable different. We show that each time we double number of processed bit at a time, architecture's throughput also doubles in all four implementations. Thus, when choosing a type of accelerator, the trade off is between flexibility/adaptability, throughput and resource utilization.

Even though fully-adaptable architecture a) exhibits highest throughput among all implementations, it also exhibits highest resource utilization on Xilinx Virtex 6 LX550T board. It occupies between 1.6% and 14.2% of LUTs resources, while the architecture b) occupies only 1-2% of LUTs resources. LUTs resource utilization of non-adaptable accelerators is around 1% on the same device. Resource utilization is kept at minimum in each implementation with different M's, as explained in Section 3.4. Every consecutive implementation in a) adds insignificantly more resources, while in b) most resources are occupied by contents of Tables. We think that resource utilization is still acceptably low, especially for fully-adaptable architecture a).

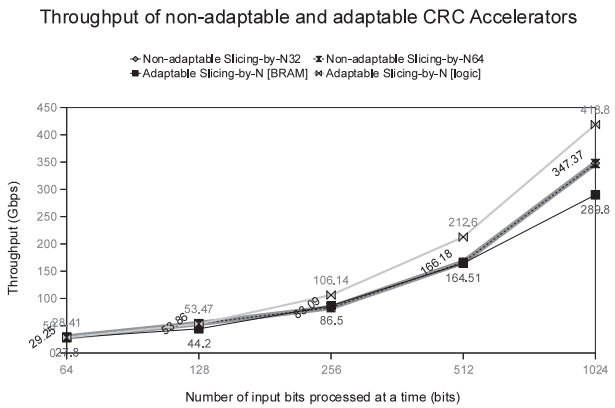

Fig. 6 Throughputs of four implementations of non-adaptable and fully-adaptable CRC accelerators.

5.2.2 Time for Re-Generation of Tables

It's important to consider time required for regeneration of content in Tables. This happens only when CRC standard changes. In Table 5. we present number of cycles and time required for each algorithm we implemented. We consider this to be reasonably quick and not noticeable by a user.

5.3 Comparison to Related Work

Table 6. provides a summary of related works for generating CRC implemented on a different technologies. Although it is difcult to compare performance and area parameters for different technologies, some valid comparisons can be made.

Compared to all non-adaptable hardware designs, [7] achieves the best throughput, but our circuit is more than 2x faster (M=128). Unfortunately, the area used in this design is not provided. When compared with fastest software solution [1], our design is 8x faster with further ability to extend number of bits processed at a time. In [4], [5] and [6] the throughput is significantly lower than our implementations and the circuits have to be taken off-line in order to support other CRC Standards.

There are only two adaptable hardware implementa-

Table 5 Number of clock cycles and time required for re-generation of Tables when generator polynomial changes.

Input data	Tables	Clock Cycles	BRAM (μs)	logic (μs)
M = 64 bits	8	320	0.91	0.72
M = 128 bits	16	576	1.67	1.38
M = 256 bits	32	1088	3.22	2.62
M = 512 bits	64	1600	4.98	3.85
M = 1024 bits	128	2112	7.46	5.17

Table 6 A summary of different CRC designs from Related Work and our implementations on the Xilinx Virtex 6 LX550T: a) fully-adaptable CRC with *Overlapped architecture* with Tables in BRAM and b) Tables in logic; c) non-adaptable Slicing–by–N_{32} CRC and d) non-adaptable Slicing–by–N_{64} CRC.

Design	Polynomial	M	Adaptable	Re-generation time	Technology	Area	Throughput
[1]	32	32, 64	√	N/A	Pentium 1.7 GHz (90 nm)	-	1.4, 3.6
[4]	32	32	–	N/A	350 nm (137 MHz)	162 LUTs	4.38
[5]	32	32, 64	–	N/A	350 nm AMS (180 MHz)	7.73 mm^2	5.76
[6]	16, 32	8, 16, 32	–	N/A	FLEX10KE ALTERA family	149 - 1849 LC	1.1 (8b) - 4.6 (32b)
[7]	32	128	–	N/A	90 nm ST CMOS (200 MHz)	N/A	~25
[8]	8, 32	128	–/√	N/A	180 nm (200 MHz)	N/A	1.3 - 3.7
[9]	32	32 (64)	√	< 1 μs	130 nm UMC standard cell	0.15 mm^2	4.92 (9.84)
Our a)	up to 64	64-1024	√	0.91-7.46 μs	Virtex 6 LX550T	3398 - 6774 LUTs	27.8 - 289.8
Our b)	up to 64	64-1024	√	0.72-5.17 μs	Virtex 6 LX550T	5571 - 48756 LUTs	28.41 - 418.8
Our c)	32	64-1024	–	N/A	Virtex 6 LX550T	205 - 1180 LUTs	29.32 - 347.98
Our d)	64	64-1024	–	N/A	Virtex 6 LX550T	540 - 2230 LUTs	29.25 - 347.37

tions with limited support for a number of generator polynomials [8], [9]. They differ from our implementation in that they require separate implementation for every generator polynomial, while our circuits support variable number of CRC standards with only one implementation. Our fully-adaptable *Slicing–by–16* (M=128) implementation is 41x faster than [8] soft-coded design with 32 bit CRC and 14.5x faster than hard-coded design. Unfortunately, the reconfiguration time and area utilization are not provided. [9] is generic in its design, thus it can be scaled to process 64, 128 or 256 bits, with maximal theoretical throughput of 40 Gbps at 256 bits. It is 6x slower compared with our adaptable *Slicing–by–8* implementation, and 5x slower compared with adaptable *Slicing–by–16*. The reconfiguration time is not very specific - under 1 μs, just as our adaptable *Slicing–by–8*. It uses different implementation technology, thus it is very difficult to compare area used.

As can be seen from Table 6., there is no much research about CRC circuits that support 64 generator polynomial, so we cannot compare our implementation to any other in that terms.

6. Conclusions

In this paper we presented a methodology for designing non-adaptable and fully-adaptable CRC accelerators based on a table-based algorithm intended for software implementation. We reduced computational burden by offloading processing of CRC on an FPGA. Both architectures process arbitrary number of data input and exhibit significant improvements in throughput over related works. They are very efficient in area utilization, especially non-adaptable and fully-adaptable architectures with Tables implemented in BRAM, which occupy 1-2% of resources on Virtex 6 LX550T. Our fully-adaptable architecture supports any known CRC standard, up to 65 bits of generator polynomial, during run-time. The architecture is able to re-generate contents of Tables very fast while occupying minimum amount of resources. The usability of non-adaptable CRC architecture is limited, but it exhibits significant increase in throughput.

We accomplished efficient area utilization by modifying Table Generation algorithm in order to decrease its space complexity from $O(nm)$ to $O(n)$. We also explored possibility of processing arbitrary number of input data M, where $M \in 64, 128, 256, 512, 1024$ and we discussed effects of this scalability. This enabled further expansion in a number of input bits processed at a time, and thus higher throughput. Our accelerators are between 1.65 to 41x faster than related work, depending on a number of bits processed at a time.

References

[1] F.L. Berry and M.E. Kounavis, "A systematic approach to building high performance software-based CRC generators," ISCC '05: Proc. 10th IEEE Symposium on Computers and Communications, pp.855–862, IEEE Computer Society, Washington, DC, USA, 2005.
[2] C. Borrelli, IEEE 802.3 Cyclic Redundancy Check, http://www.xilinx.com/support/documentation/application_notes/xapp209.pdf
[3] D.V. Sarwate, "Computation of cyclic redundancy checks via table look-up," Commun. ACM, vol.31, no.8, pp.1008–1013, 1988.
[4] G. Campobello, G. Patane, and M. Russo, "Parallel CRC realization," IEEE Trans. Comput., vol.52, no.10, pp.1312–1319, Oct. 2003.
[5] T. Henriksson and D. Liu, "Implementation of fast CRC calculation," Proc. Asia and South Pacific Design Automation Conference (ASP-DAC) 2003, pp.563–564, Jan. 2003.
[6] F. Monteiro, A. Dandache, A. M'sir, and B. Lepley, "A fast CRC implementation on FPGA using a pipelined architecture for the polynomial division," Proc. 8th IEEE International Conference on Electronics, Circuits and Systems, ICECS 2001, vol.3, pp.1231–1234, 2001.
[7] C. Mucci, L. Vanzolini, I. Mirimin, D. Gazzola, A. Deledda, S. Goller, J. Knaeblein, A. Schneider, L. Ciccarelli, and F. Campi, "Implementation of parallel LFSR-based applications on an adaptive DSP featuring a pipelined configurable gate array," Design, Automation and Test in Europe, 2008, DATE '08, pp.1444–1449, March 2008.
[8] H.M. Ji and E. Killian, "Fast parallel CRC algorithm and implementation on a configurable processor," IEEE International Conference on Communications (ICC) 2002, vol.3, pp.1813–1817, 2002.
[9] C. Toal, K. McLaughlin, S. Sezer, and X. Yang, "Design and implementation of a field programmable CRC circuit architecture," IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol.17, no.8, pp.1142–1147, Aug. 2009.
[10] A. Akagic and H. Amano, "An FPGA implementation of CRC slicing-by-N algorithms," IEICE Technical Report, RECONF2010-42, Nov. 2010.
[11] M.E. Kounavis and F.L. Berry, "Novel table lookup-based algorithms for high-performance CRC generation," IEEE Trans. Comput., vol.57, no.11, pp.1555–1560, Nov. 2008.
[12] A. Akagic and H. Amano, "Performance evaluation of multiple lookup tables algorithms for generating CRC on an FPGA," International Symposium on Access Spaces (IEEE-ISAS 2011), pp.164–169, June 2011.
[13] A. Akagic and H. Amano, "High speed CRC with 64-bit generator polynomial on an FPGA," International Workshop on Highly Efficient Accelerators and Reconfigurable Technologies (HEART), Imperial College, London, UK, June 2011.

Amila Akagic received Dipl. el. Ing. and M.Sc. in Computer Science from Faculty for Electrical Engineering, University of Sarajevo, Bosnia and Herzegovina in 2006 and 2009, respectively. She is currently a Ph.D. student with Amano Lab at Keio University, Yokohama, Japan. Her research interests include computer architectures, reconfigurable computing and acceleration of algorithms in hardware.

Hideharu Amano received Ph.D. degree from the Deparment of Electronic Engineering, Keio University, Japan in 1986. He is currently a professor in the Department of Information and Computer Science, Keio University. His research interests include parallel architectures and reconfigurable systems.