Effective FPGA Architecture for General CRC

Lukáš Kekely[1(✉)], Jakub Cabal[1], and Jan Kořenek[2]

[1] CESNET a. l. e., Zikova 4, 160 00 Prague, Czech Republic
{kekely,cabal}@cesnet.cz
[2] IT4Innovations Centre of Excellence, FIT BUT, Božetěchova 2, 612 66 Brno, Czech Republic
korenek@fit.vutbr.cz

Abstract. As throughputs of digital networks and memory interfaces are on a constant rise, there is a need for ever-faster implementations of error-detecting codes. Cyclic redundancy checks (CRC) are a common and widely used type of codes to ensure consistency or detect accidental changes of transferred data. We propose a novel FPGA architecture for the computation of the CRC values designed for general high-speed data transfers. Its key feature is allowing a processing of multiple independent data packets (transactions) in each clock cycle, what is a necessity for achieving high overall throughput on very wide data buses. The proposed approach can be effectively used in Ethernet MACs for different speeds, in Hybrid Memory Cube (HMC) controller, and in many other technologies utilizing any kind of CRC. Experimental results confirm that the proposed architecture enables reaching an effective throughput sufficient for utilization in multi-terabit Ethernet networks (over 2 Tbps or over 3000 Mpps) on a single Xilinx UltraScale+ FPGA. Furthermore, a better utilization of FPGA resources is achieved compared to existing CRC implementation for HMC controller (up to 70% savings).

Keywords: FPGA · CRC · High-speed processing · Ethernet · HMC

1 Introduction

The Cyclic Redundancy Check (CRC) codes are widely deployed in digital communications and storage systems to detect accidental error introduced into data. The binary data are divided into transactions (packets) and each transaction is subjected to a CRC which results in a fixed-length binary check sequence. The computed check sequence value is then attached to the original data to determine its correctness. After being transferred/processed, the data are subject to the same CRC computation one more time and the new result is compared with the older attached CRC value. In case of a match, the data transaction is most likely not corrupted. Because of their simple implementation in hardware and good characteristics, the utilization of CRCs is very popular [5,6].

M. Schoeberl et al. (Eds.): ARCS 2019, LNCS 11479, pp. 211–223, 2019.
https://doi.org/10.1007/978-3-030-18656-2_16

The computation of CRC codes is based on the remainder of a polynomial division where coefficients are elements of the finite field $GF(2)$. There are many different CRC codes, each defined by a specific dividing polynomial and output (code) width. The mathematical background of CRC and forms of its hardware representation have been extensively studied in various works like [10,11,13] and is not the primary focus of this paper. All we need to know is that an approach capable of processing multiple input bits in parallel exists and is based on XOR equations set up for each output bit. A specific set of these equations (CRC table) can be easily constructed for any given dividing polynomial and input data word width. Furthermore, multiple results of these CRC tables can be aggregated (accumulated) together to obtain code value of longer data transaction.

Although basic CRC computation can be easily represented, practical processing of high-speed data is much harder. The data packets usually have variable lengths and are not completely aligned with data bus words. Unaligned ends and starts must be handled correctly, which requires additional logic and more complex architecture than a single CRC table. Furthermore, as the data bus width is growing to raise throughput, transfers of multiple packets per clock cycle (data bus word) must be supported. This challenge must be addressed in practical high-speed CRC implementation and that is indeed the main focus of our work.

We propose a novel FPGA architecture for practical computation of CRC codes for general high-speed transfers of data packets with variable lengths. The architecture enables effective computation of multiple values per clock cycle in a single pipeline thus allows handling of multiple packets in each data bus word. Furthermore, it supports configurable pipelining (before synthesis) so optimal tradeoff between frequency (throughput) and utilized resources can be selected. When fully pipelined, the CRC architecture achieves unprecedented throughput of over 2 Tbps or 3000 millions of packets per second (Mpps) in a single FPGA.

2 Related Work

The mathematical background of CRC computation has been extensively studied in many previous works like [7,10,11,13] and it is not the focus of this paper. Rather, we want to use the results and proposed effective hardware representation of basic CRC calculations from these papers as primary constructional blocks of a new architecture. However, the challenge of practical high-speed CRC computation for variable-length data packets is more complicated.

Some attempts to address this additional challenges are made in [4]. Architectures arranging basic CRC calculation into more complex structures are proposed to enable processing of unaligned packets ending. However, the proposed architectures are shown to scale well only up to throughputs around 10 Gbps (256 b wide bus) what is insufficient for current high-speed data handling.

More advanced general CRC implementations are described in many papers like [1,3,15]. All of them use a kind of advanced pipelining and parallelization to achieve higher frequencies (throughputs) than other simpler solutions. The Ethernet CRC-32 implementations by these architectures use input data widths of

64 to 512 bits and can run at hundreds of MHz. This leads to reported throughputs sufficient for wire-speed traffic processing of up to 100 Gbps Ethernet. But scaling for higher speeds is not properly addressed in any of these works and would bring exponential growth in required FPGA area or significant degradation of effective throughput on short packets (i.e. data rate limited by packet rate). Furthermore, the extension of these architectures to allow multiple packets per clock cycle (i.e. sufficiently increasing their packet rate) would be non-trivial.

Interesting CRC architecture [8,9] uses pipelining similar to the above works to achieve high throughput and focuses primarily on reconfigurability of CRC polynomial, but it also partially addresses the challenge of limited packet rate on short packets. The architecture can process parts of two subsequent packets in a single clock cycle (data bus word). A maximal throughput of 40 Gbps reported in the paper can be thus easily scaled up to 100 or even 200 Gbps. But because the parallel processing is structurally limited to only two packet parts, further scaling would again hit the same obstacles as mentioned above.

Fastest commercially available CRC architecture is part of Ethernet MAC IP core [14]. In the product description, its authors claim to be able to achieve up to 400 Gbps line-rate processing of packets using only a small portion of FPGA area. But no throughput measurements nor exact resource requirements are provided to back up those claims. Furthermore, any details about their CRC architecture or its parameters (e.g. frequency, data bus width) are also lacking.

3 Architecture Design

Here we describe the proposed CRC architecture. First, data bus format with multiple packets per clock cycle is defined. This is crucial for efficient scaling above 100 Gbps. After that, basic utilized CRC computing blocks are introduced. Finally, the architecture itself is presented in serial and parallel versions.

3.1 Input Bus Format

To enable multiple packets per clock cycle, we define the input data bus word format as illustrated in Fig. 1. The figure also shows an example of possible packet placement under the proposed format. One should notice that without the support of multiple packets per clock cycle, each of the depicted data frames should occupy separate word on the bus (5 words would be required), but word sharing enables more dense packing (only 3 words are needed in the example). The proposed bus format is shown at the bottom of the figure, each data word is divided into several *regions*. These restrain the maximum number of data packets per word as at most one packet can start and one end (can be a different one) in each region. Each region is further separated into multiple *blocks* of basic data *elements* (items) to constraint possible positioning of packet starts. Notice that each packet must start aligned with the start of a block, but can end on any data element (packets A and B both end in the middle of a block).

To support the described bus format, additional metadata must accompany each data word. For each region the following information must be given:

– a flag for the presence of a packet start (SOP),
– a flag for the presence of a packet end (EOP),
– a position of packet start if present (SOP_POS),
– a position of packet end if present (EOP_POS).

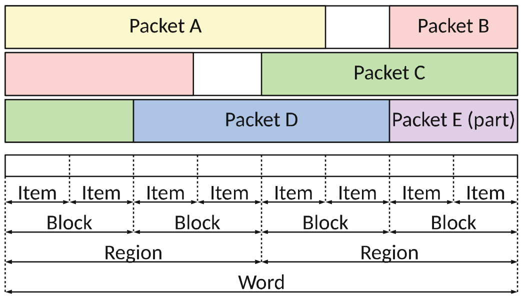


Fig. 1. Data bus format illustration.

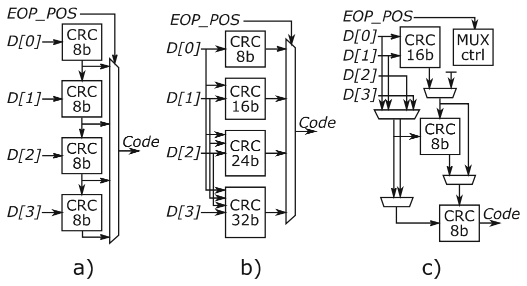


Fig. 2. CRC end realizations possibilities.

The proposed data word format enables definitions of multiple bus versions with different parameters. We describe them by these four attributes:

– *Number of regions (n)* match the maximal number of packets per word.
– *Region size (r)* defines the number of blocks in a region.
– *Block size (b)* states the number of elements in a block.
– *Element width (e)* defines the size of the smallest piece of data in bits.

Using these attributes, we derive bus word width in bits like $dw = n \times r \times b \times e$.

3.2 CRC Computation Blocks

In both versions of the proposed architecture, we utilize 4 basic computational units: (1) basic CRC table for fixed input width, (2) accumulation logic capable of aggregating multiple intermediate CRCs, (3) correction of input data based on packet start position, and (4) finalization of CRC based on packet end position.

As already mentioned in the Introduction, based on given dividing polynomial and input width a specific implementation of basic **CRC table** can be easily generated [13]. It has a form of parallel XOR equations on input bits, one equation for each output (code) bit. In FPGAs, these XORs are implemented in LUTs. The CRC table basically only converts the input data word into an intermediary CRC code value without regard to packet borders.

Specific **CRC accumulation** can be similarly generated for any polynomial. It has a form of parallel XOR equations and it aggregates two or more intermediary CRC values computed from separate parts of data (e.g. by CRC tables). This enables to divide handling of longer data packets in multiple smaller steps.

Correction of **CRC start** based on packet position can be achieved by masking – the part of the input before packet start is filled with zeros. CRC computations are based on XOR operations and zero is a neutral value for them ($0\ xor a = a$ for any a). Therefore, it is possible to show that extension of any data by prepending any number of leading zeros has no effect on computed CRC value, which remains the same as for original data [7]. Note, that also the initial (intermediary) value of CRC register must be shifted and applied accordingly.

Finally, correct handling of **CRC end** is a bit more complicated. Masking similar to start correction cannot be directly applied, as appending trailing zeros to data will change the computed CRC value. A workaround is to use a barrel-shifter to shift the last data part so that the end of the packet is aligned with the end of the region. This way, the masking operation is converted from trailing zeros into leading zeros and can be applied in the same way as in CRC start. Another possible type of approach is to utilize some arrangement of multiple smaller CRC tables [4]. Illustration of these arrangements for 32 bit wide region and $e = 8$ are shown in Fig. 2. On the left, we can see a serial version, where multiple tables are pipelined each processing one input data element and correct output code is selected afterward based on packet end position. In the middle, there is a parallel version, where each possible position of the end has its own accordingly wide table. These basic approaches do not scale well for wider data buses – depth of the pipeline (critical path) in (a) or amount of resources in (b). To issue the scaling challenge a more sophisticated approach illustrated as (c) can be used. Each pipeline step corresponds to one layer of a binary search tree and performs CRC computation with a gradually halving table width which can be applied or bypassed. The binary tree is evaluated for a given packet end position (MUX ctrl) and bypass multiplexors at each pipeline step are controlled accordingly. At the end an implicit CRC finalization table with width e is present. For example, for the computation of 24 bit long packet end only the middle 8 bit table is bypassed, and for 16 bit end the top 16 bit table is bypassed.

Thanks to division and encapsulation of all basic CRC computations into the described blocks, the subsequently designed architecture will be general and easily usable for any given division polynomial. Because, the change of the polynomial only requires re-generation of used CRC tables (XOR equations) in these blocks and will not affect structure of the whole architecture.

3.3 Serial and Parallel Architectures

Both versions of the proposed CRC architecture divide processing of input data word between n submodules – one for each region. Each submodule can process an independent packet in each clock cycle or they can cooperate together and handle longer packets. Serial and parallel version differ primarily in the distribution of intermediate CRC values between these submodules. Figure 3 shows top level structure of the serial implementation. One region of the input bus (width $rw = r \times b \times e$) is connected to each submodule. The submodule calculates final CRC value if an end of the packet is present in his part of input bus. To support cooperation on longer packets, each submodule is passing its intermediate CRC

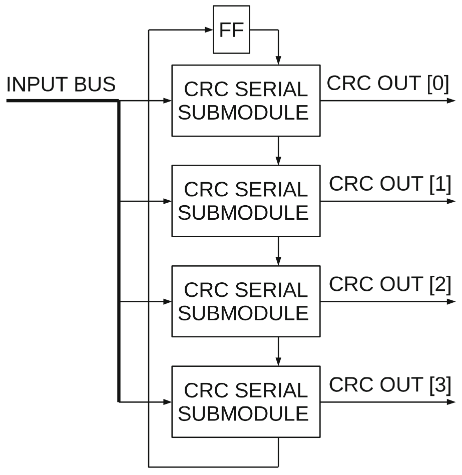


Fig. 3. Serial top level architecture.

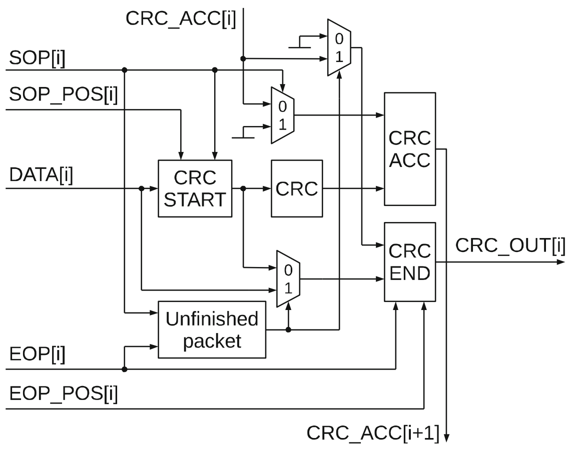
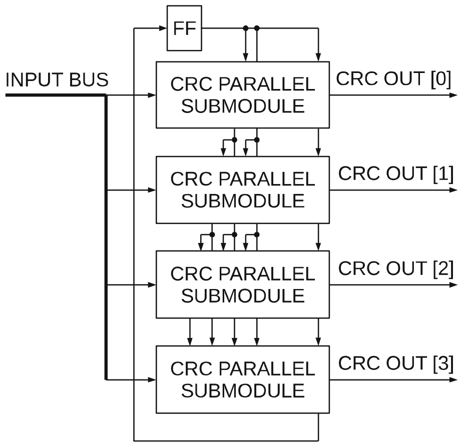


Fig. 4. Serial submodule internal structure.

result to the next submodule. The last submodule is passing its result to the first over a register, so the calculation can continue in the next data bus word.

In Fig. 4 we can see internal structure of one serial submodule. It is composed of several logic stages and optional registers for better timing. Base CRC table of width rw is used as a core computational block and handling of corrections required for starting, ending or continuing packets is realized by multiple separate blocks around it. They are controlled by metadata about packet positioning in the assigned region of the bus. The CRC start block masks input data before the packet start so that subsequent base CRC calculation is performed correctly for starting packets. If no start of a packet is present, the input data word is not altered. If a packet continuing from the previous words is present in the input data, the output value of CRC table is aggregated with an intermediate CRC value from the previous submodule in accumulation block. Otherwise (starting packet), the input CRC value is masked and no aggregation is performed – only locally computed result is passed. The output of accumulation block is used as intermediate CRC value on the input of the next submodule. Finally, CRC end block performs CRC calculation for packets ending in data region assigned to this submodule. When the whole packet data (start and end) are present in the region of this submodule, the final CRC value is calculated only from masked input data. Otherwise, output CRC value is calculated from the intermediate result from the previous submodule and unaltered input data.

The serial implementation has a weak point – long critical path from the output of the CRC register, through CRC aggregation in all submodules, and back to the register (Fig. 3). This critical path cannot be resolved using pipelining as correct CRC intermediate value must be present in the register when processing of the next word starts. That is why, we propose the parallel version of CRC aggregation. In Fig. 5 we can see that, the output value of CRC submodule is shared with each subsequent submodule not just with the next one. In Fig. 6 we can see internal structure of CRC submodule accommodated for the parallel aggregation. There are several major changes present. The output value of CRC accumulation block now serves only for the final CRC calculation in the CRC end block of the next submodule. So, the intermediate CRC results are not accumulated in steps through the whole pipeline of submodules. Now, each CRC accumulation block must independently aggregate intermediate CRC values from all previous submodules including value stored in the top-level register. The other parts of the parallel implementation remain the same as in the serial one. This version has significantly improved critical path and allows to achieve much higher operating frequencies. On the other hand, it requires considerably more logic resources as more complicated CRC accumulation modules are used.



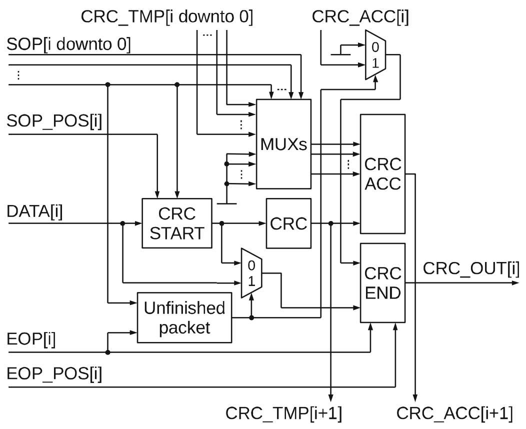


Fig. 5. Parallel top level architecture. **Fig. 6.** Parallel submodule internal structure.

4 Measured Results

We evaluate the proposed CRC architecture in two high-speed cases: Ethernet networks and HMC controller. CRC ensures consistency of data packets in both cases, but different polynomials are used. A detailed evaluation is performed for the networking case, where effects of various architecture parameters are explored. In HMC case we directly select the best configurations and compare them with existing CRC implementation in the OpenHMC controller.

4.1 Ethernet Based Networks

Ethernet uses CRC with the CRC-32 division polynomial [6] as a frames check sequence. As already discussed in the Related Work, published architectures can be effectively used for Ethernet traffic processing at speeds up to 200 Gbps and commercially available solutions promise throughputs of up to 400 Gbps. Their scaling towards higher speeds is limited by insufficient packet rates on the shortest packets for wider data buses. The proposed architecture addresses exactly this issue and should be able to scale well even at higher throughputs.

When adjusting the proposed architecture for Ethernet, the parameters of the bus format should be configured to appropriate values. Ethernet operates with bytes (octets) as the smallest data elements – therefore $e = 8$. Lower layers of Ethernet (PCS/PMA layers) usually operate with frame starts aligned at 8 B lanes – so $b = 8$ is convenient. Size of a region should correspond with the size of the smallest allowed packets (64 B) – so $r = 64/b = 8$. Smaller regions would needlessly allow more packets per word than possible and larger regions would reduce bus saturation for the shortest packets. Using these attributes ($r = b = e = 8$) and considering the shortest packets to be 64 B long, the bus format impose no more than $b - 1 = 7$ bytes of alignment overhead per packet. Furthermore, as lower layers of Ethernet operate with larger overhead per packet (20 B of preamble and IFG), our bus enables us to achieve effective throughput sufficient for wire-speed processing of Ethernet packets even in the worst case.

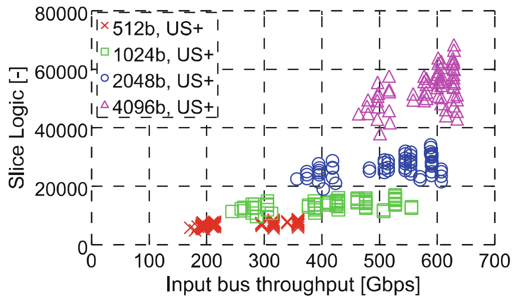


Fig. 7. Throughput and logic of S-S.

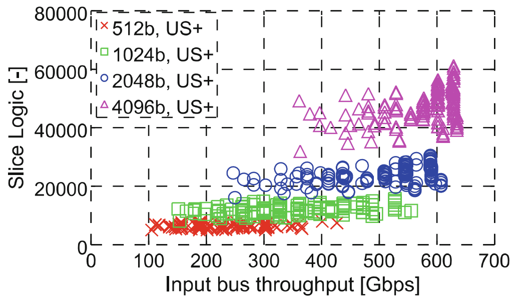


Fig. 8. Throughput and logic of S-T.

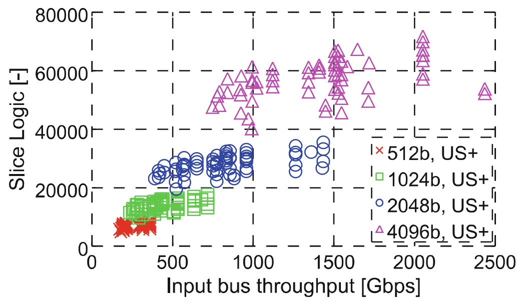


Fig. 9. Throughput and logic of P-S.

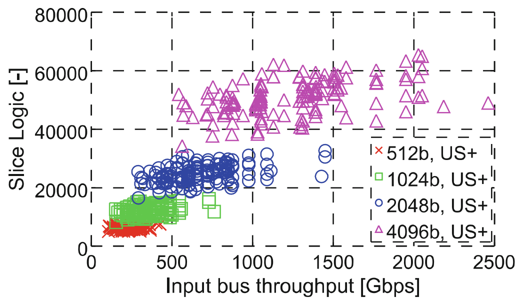


Fig. 10. Throughput and logic of P-T.

Evaluation for Ethernet compares four versions of the proposed architecture: (1) **S-S** – *serial* architecture with *shifter* CRC end, (2) **S-T** – *serial* architecture with *tree* CRC end, (3) **P-S** – *parallel* architecture with *shifter* CRC end, and (4) **P-T** – *parallel* architecture with *tree* CRC end. For each, we measure results for different data bus widths ($dw = 512, 1024, 2048, 4096$) and various combinations of pipeline registers. In all cases, we use data bus parameters $r = b = e = 8$ that are sufficient for wire-speed processing of even the shortest frames, only the value of n is changing with the width of the bus. The results of all the combinations form the state space of CRC implementations with different throughput, working frequency, latency and resource usage. All values are obtained for the Xilinx Virtex-7 XCVH870T or UltraScale+ XCVU7P FPGAs using the Vivado 2017.3.

Figures 7, 8, 9, and 10 compare the four versions of Ethernet CRC architecture. They show resource utilization and achieved throughput on the UltraScale+ FPGA. Each point represents one specific implementation with a different combination of parameters (data width and pipeline enabling). The resources utilization linearly increases with the achieved throughput in both parallel versions (Figs. 9 and 10). Unfortunately, in both serial versions (Figs. 7 and 8) the resources increase considerably faster with throughput. In the case of the P-S and the P-T implementations, we are able to reach effective throughputs of well over 2 Tbps (over 3000 Mpps). Achieved throughputs for the S-S and the S-T implementations are notably worse while the used resources remain similarly high. This is because the serial CRC generators reach notably lower frequencies due to an expected longer critical path.

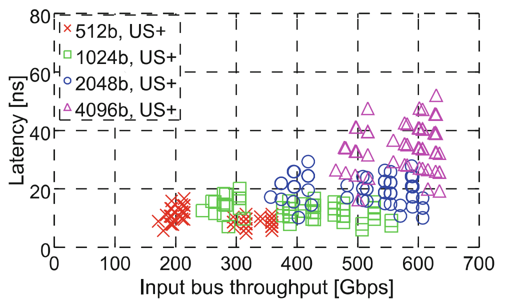


Fig. 11. Throughput and latency of S-S.

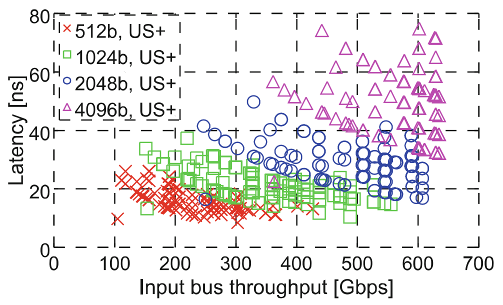


Fig. 12. Throughput and latency of S-T.

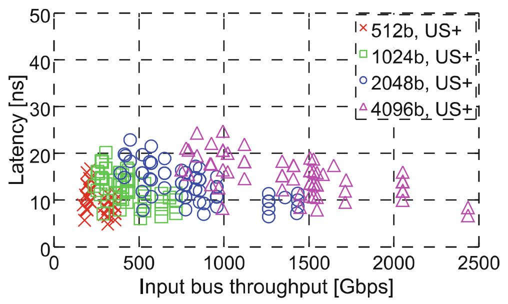


Fig. 13. Throughput and latency of P-S.

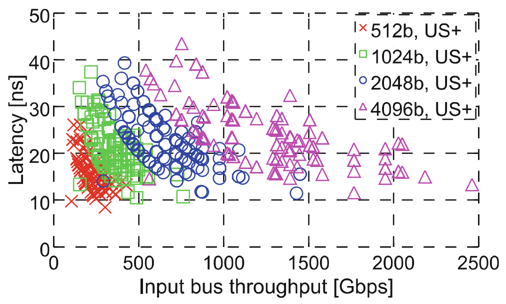


Fig. 14. Throughput and latency of P-T.

Figures 11, 12, 13, and 14 bring the latency into the picture. Generally, the latency depends on the number of enabled pipeline registers in CRC implementations and achieved working frequency. From the graphs, we can see that the latencies of the serial implementations are increasing notably as the achieved throughput (word width) is rising. On the other hand, the latencies of the parallel implementations remain approximately within the same bounds. This is again due to the higher frequency of parallel implementations even for wider buses.

Figure 15 shows the evaluated four versions of Ethernet CRC implementations together. Only Pareto optimal set of results in resource utilization and achieved throughput space is selected for each implementation version. From the graph, we can more clearly see the difference between the serial (dashed lines) and the parallel (full lines) implementations in achieved throughput. The parallel implementations are able to reach the effective throughput of over 2 Tbps, while the serial implementations cannot reach significantly more than 600 Gbps. Furthermore, parallel-tree has slightly better resource utilization than parallel-shifter.

Figure 16 shows Pareto optimal results of latency to achieved throughput for the four evaluated versions of Ethernet CRC implementations. Again, we can see the notable difference between the serial (dashed lines) and the parallel (full lines) implementations. The latency of the serial implementations steeply increases with the throughput (bus width), but the latency of parallel implementations raises only rather slowly. Better parallel implementation version in terms of latency is the parallel-shifter one. This is due to smaller number of registers in CRC end module for shifter version compared to tree version.

Figure 17 compares results between different FPGAs – magenta for the UltraScale+ and blue for the Virtex-7 FPGA. It shows the best parallel implementations in resource utilization to achieved throughput space. To compensate for

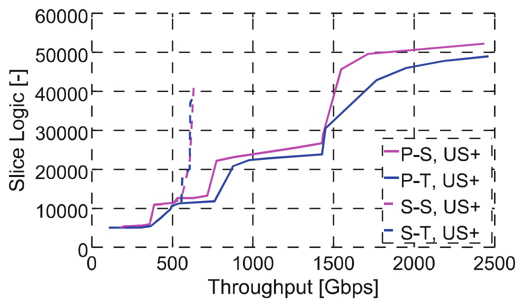


Fig. 15. Best throughput × logic results.

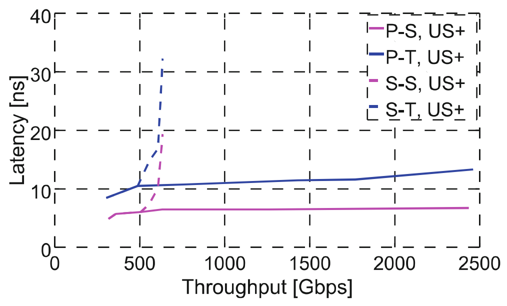


Fig. 16. Best throughput × latency results.

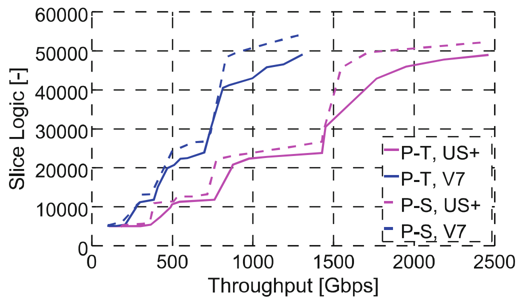


Fig. 17. Throughput × logic, varied chip.

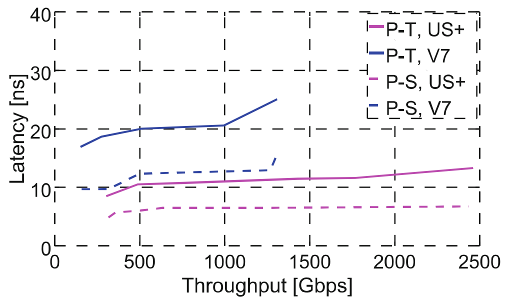


Fig. 18. Throughput × latency, varied chip.

the 1.5-2 times lower achieved frequencies on the Virtex-7 chip compared to the UltraScale+, nearly 2 times larger implementations must be used when trying to achieve the same throughputs. The stairs in the graphs are caused by the changing data bus width. Finally, Fig. 18 compares latencies of the best parallel implementations of Ethernet CRC generator between different FPGAs. Again, we can see the positive effect of the higher frequencies and sparser registering on the UltraScale+, where the latency is nearly 2 times better.

4.2 OpenHMC Controller

Hybrid Memory Cube (HMC) is a high-performance RAM interface that uses a 32 bit version of CRC with the CRC-32K (Koopman) division polynomial [2,12]. Again, an appropriate adjustment of the data bus format parameters should be considered first. HMC operates with data divided into 128 bit wide 'flits' as the smallest data elements, therefore $e = 128$. Each data transaction (packet) is a continuous sequence of 1 or more flits, so $r = b = 1$. Finally, the number of regions n depends on the width of the memory interface, commonly used widths are 4, 6 or 8 flits. This kind of bus arrangement leads to a considerably simplified computation in each submodule. As packets start and end only aligned to the region borders, CRC start and CRC end blocks are not needed.

Table 1. Comparison of OpenHMC CRC implementation to the proposed.

Bus width	Implementation	LUTs	FFs	Fmax
512	OpenHMC	4988	2477	700 MHz
	proposed	2262	1858	807 MHz
768	OpenHMC	12071	3778	594 MHz
	proposed	3935	2791	802 MHz
1024	OpenHMC	23599	5125	517 MHz
	proposed	6340	3728	798 MHz

An existing opensource controller implementation of HMC interface is called OpenHMC controller [2]. It utilizes its own specific implementation of CRC architecture capable of handling multiple flits per clock cycle. The CRC implementation is a critical part of the whole controller, as it consumes the majority of all FPGA logic required. We compare this default implementation to our proposed CRC architecture in the parallel version for different data widths. The results for the UltraScale+ FPGA are provided in the Table 1. While our architecture is configured to have the same latency and throughput as the OpenHMC default CRC implementation, a clear difference in resource utilization is visible. Our implementation requires less than half of the logic and around 75% of registers for 512 b (4 flits) wide bus. Resource saving increases even further for wider data buses, up to only a quarter of logic and around 70% of registers. Achieved frequency is also better in our implementations, it especially scales considerably better with rising bus width compared to default OpenHMC implementation.

5 Conclusion

This paper introduces and elaborates a novel FPGA architecture of general CRC computation that enables achieving very high processing throughputs. The proposed architecture is able to process multiple packets per clock cycle and offers good scalability even for very wide data buses. Thanks to a well defined and configurable structure, the architecture can be easily adjusted for CRC computation based on any given polynomial. Furthermore, we can optimize achieved parameters for specific application requirements in terms of processing latency, FPGA resources utilization, and total computational throughput.

Our experimental evaluation shows, that when computing CRC (FCS) for Ethernet frames in high-speed networks the proposed concept enables to achieve unprecedented wire-speed throughput. At a cost of just a few percents of total resources available in a single UltraScale+ FPGA, the achieved throughput can be as high as 2.4 Tbps (over 3500 Mpps). That is, to our knowledge, considerably higher than in any other published work. It is especially thanks to favorable frequency scaling of the designed parallel version of the proposed architecture. The second part of the measurements shows results of our CRC architecture adjusted for high-speed HMC interface. Our approach achieves much better results than default CRC implementation inside OpenHMC controller in terms of both resources as well as frequency (throughput). For the same data width (number of parallel flits), we can save up to 73% logic and 27% registers.

The proposed architecture has been verified in simulations and is also currently tested on a real FPGA as part of our semi-finished implementation of 400 GbE MAC. As part of our future work, we want to propose a feasible approach to high-speed RS-FEC computation in a single FPGA. RS-FEC is based on similar mathematical principles as CRC (finite fields) and is required part of 400G Ethernet implementation.

Acknowledgments. This research has been supported by the MEYS of the Czech Republic project Reg. No. CZ.02.1.01/0.0/0.0/16_013/0001797, the IT4Innovations excellence in science project IT4I XS−LQ1602, and by the Ministry of the Interior of the Czech Republic project VI20172020064.

References

1. Bajarangbali, Anand, P.A.: Design of high speed CRC algorithm for ethernet on FPGA using reduced lookup table algorithm. In: IEEE India Conference (2016)
2. Computer Architecture Group and Micron Foundation: OpenHMC: a configurable open-source hybrid memory cube controller. University of Heidelberg (2014)
3. Hamed, H.F.A., Elmisery, F., Elkader, A.A.H.A.: Implementation of low area and high data throughput CRC design on FPGA. Int. J. Adv. Res. Comput. Sci. Electron. Eng. **1**(9) (2012)

4. Henriksson, T., Liu, D.: Implementation of fast CRC calculation. In: Proceedings of the Asia and South Pacific, Design Automatation Conference, pp. 563–564 (2003)
5. HMC Consortium: hybrid memory cube specification 2.1. Altera Corp. (2015)
6. IEEE Computer Society: Amendment 10: media access control parameters, physical layers and management parameters for 200 Gb/s and 400 Gb/s operation. IEEE Standard 802.3bs-2017, pp. 1–372 (2017)
7. Kennedy, C., Reyhani-Masoleh, A.: High-speed parallel CRC circuits. In: 42nd Asilomar Conference on Signals, Systems and Computers, pp. 1823–1829 (2008)
8. Mitra, J., Nayak, T.K.: Reconfigurable concurrent VLSI (FPGA) design architecture of CRC-32 for high-speed data communication. In: IEEE International Symposium on Nanoelectronic and Information Systems, pp. 112–117 (2015)
9. Mitra, J., Nayak, T.: Reconfigurable very high throughput low latency VLSI (FPGA) design architecture of CRC 32. Integr. VLSI J. **56**, 1–14 (2017)
10. Pei, T.B., Zukowski, C.: High-speed parallel CRC circuits in VLSI. IEEE Trans. Commun. **40**(4), 653–657 (1992)
11. Perez, A.: Byte-wise CRC calculations. IEEE Micro **3**(3), 40–50 (1983)
12. Schmidt, J., Bruning, U.: OpenHMC: a configurable open-source hybrid memory cube controller. In: ReConFigurable Computing and FPGAs. IEEE (2015)
13. Shieh, M.D., Sheu, M.H., Chen, C.H., Lo, H.F.: A systematic approach for parallel CRC computations. J. Inf. Sci. Eng. **17**(3), 445–461 (2001)
14. Tamba Networks: Datacenter Ethernet. Tamba Networks, LLC (2018). http://www.tambanetworks.com/products/datacenter-ethernet/
15. Walma, M.: Pipelined cyclic redundancy check (CRC) calculation. In: International Conference on Computer Communications and Networks, pp. 365–370 (2007)