CRC Based Hashing in FPGA Using DSP Blocks

Tomáš Závodník
Faculty of Information Technology
Brno University of Technology
Božetěchova 2, 612 66 Brno, Czech Republic
Email: xzavod12@stud.fit.vutbr.cz

Lukáš Kekely, Viktor Puš
CESNET a. l. e.
Zikova 4, 160 00 Prague, Czech Republic
Email: kekely,pus@cesnet.cz

***Abstract*—We propose a novel approach to the computation of the CRC functions, commonly used for bit error checking purposes when handling binary data. This approach is designed for general hashing purposes in FPGA, for which the CRCs are usable as well. The method is suitable for applications which use parallel inputs of fixed size and require high throughput, such as hash tables. We employ the DSP blocks present in modern FPGAs to perform all the necessary XOR operations, so that our solution does not consume any LUTs. We propose a Monte Carlo based heuristic to reduce the number of the DSP blocks required by the computation. Our experimental results show that one DSP block capable of 48 XOR operations can replace around eleven 6-input LUTs.**

***Keywords*—FPGA; CRC; DSP; Hash**

I. Introduction

The Cyclic Redundancy Check functions (CRCs) are widely deployed in digital communications and storage to detect accidental data corruptions. The binary data being handled are subjected to a CRC which results in a fixed-length binary check sequence. The check sequence is then attached to the original data and serves to derermine its correctness. After being processed, the data are subjected to the CRC one more time and the result is compared to the attached check sequence. In case of match, the data, most likely, were not corrupted. The CRCs are based on the remainder of a polynomial division and each one of them defines a specific dividing polynom and output width. Because of their simple implementation in hardware and good characteristics, the CRCs are very popular.

Common ways to implement lookup operations in FPGA are hash tables and their variations. The process of adding to, deleting from, and searching in the hash table uses one or more hash functions to compute the address to the table. A suitable hash function must meet statistical properties such as uniform distribution, use of all input bits, large change of output based on small change of input. These properties are further described in [1]. Other desirable parameters when implementing a hash function in FPGA are high throughput and low resources usage.

The hashing task of a hash table in FPGA can be met by a CRC function, for example the CRC-32 function. The CRCs generate fixed-length binary values that can be used to address a hash table cell. Additionally, their implementation requires only few resources. The statistical properties of the CRC-32 function have been thoroughly examined and the results confirmed that the CRC-32 can meet the requirements for a suitable hash function. Further details can be found for example in [2].

978-1-4799-4558-0/14/$31.00 ©2014 IEEE

Unfortunately, the implementation of the CRCs for hashing purposes is often suboptimal due to its origins in bit error checking modules. The CRC functions were originally designed to check for bit errors in variable-length data transmissions and not to compute a hash of fixed-length data. Consequently, the implementation may not meet some requirements, especially the throughput, and therefore must be revised.

II. Hashing using CRC

As mentioned, a CRC function can be succesfully used for general hashing purposes. However, the common implementations of the CRCs are focused on variable-length data processing in bit error checking modules. This is considered a drawback when designing a hash function that computes indexes to a hash table based on fixed-length keys.

In bit error checking modules, the input is subjected to a CRC function to compute the check sequence. To support variable-length data (such as Ethernet frames), the input is passed in form of a stream of bits that are processed one by one. Every bit is processed in a separate clock cycle and therefore the intermediate result is ready in the next cycle. Such approach, though, cause the overall latency to be as long as the number of bits being processed, without any possibilities to increase the throughput.

The described method of data processing suitable for a bit error checker is entirely inappropriate for a general hash function. Indexes to a hash table are computed by a hash function on the basis of fixed-length values passed each in one clock cycle. The inputs to the hash function need to be processed with the highest throughput possible to assure high-speed access to the hash table. The overall latency, on the other hand, is not as critical and pipelining is allowed. To fulfill these requirements, the method to compute the CRCs must be redesigned to support fast processing of parallel inputs.

III. Computation of CRC

The computation of a CRC can be done with two approaches. Both ways of the computation fundamentally differ from each other, offer distinct characteristics and lead to divergent implementations.

The classic way (derived directly from the CRCs definition) is based on a linear feedback shift register (LFSR). A LFSR is entirely serial and processes bit by bit. This approach

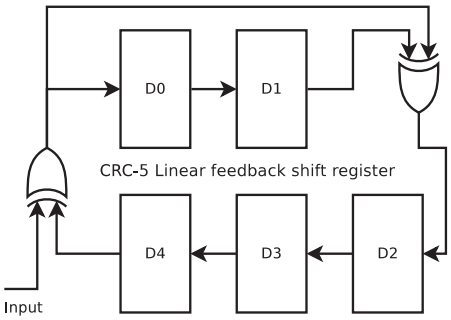

Fig. 1. Linear feedback shift register (LFSR) for CRC-5

requires minimal resources and assures high frequencies, but is only applicable on bit streams. The throughput is directly limited by the FPGA working frequency. The serial approach is therefore unsuitable for general hashing in most applications. An example LFSR for the CRC-5 is shown in the Fig. 1.

The second approach involves parallelism and is capable of performing operations with more input bits within one clock cycle. The parallel approach is based on equations set up for each output bit in compliance with the classic LFSR way of computation. The equations are typically in the form of wide XOR (exclusive OR) operations. Each output bit can be therefore described as a selection of input bits to be XORed. The construction of the equations is described in detail for example in [3]. The resulting equations are then used to build computational structures (trees). Taking advantage of the parallelism, the overall latency can be significantly reduced and the throughput improved at the cost of additional resources.

A. Parallel computation approach

When implementing the parallel CRC computation using the FPGA technology, several types of resources can be employed. Modern FPGAs offer a variety of specialized blocks, such as DSP slices or BlockRAMs, which can be used alongside the general logic.

The general FPGA logic, formed by look-up tables (LUTs), is the most common way of a CRC implementation. Each LUT in a modern FPGA is able to realize any binary function of six inputs, including the exclusive OR (XOR). In the context of a CRC, this usually means that one LUT computes five binary XOR operations. The implementation in LUTs brings some advantages including the possibility to use the prepared equations directly without any modifications. This can be done due to engaging the HDL to FPGA synthesizer's internal optimizer to build up the logic implementation. However, this way of implementation may not be appropriate when the number of LUTs is a limiting factor of the firmware design.

We propose another way to implement the computation. It is based on the use of specialized digital signal processing slices (DSPs). The DSP slices are present in most current FPGAs, yet remain largely unused in some applications, such as communications and networking. In this paper we are working with the DSP slices present in Xilinx FPGAs. Each DSP slice is (among other operations) capable of performing 48 parallel 1-bit exclusive ORs, which are essential for the computation of a CRC. When using the DSPs we can not count on automatic optimizations and must manually create the computational trees. The trees' structure is fundamental for the optimality of the computation. The minimal tree height is computed as $\lceil log_2(n+1) \rceil$ where n is the number of elements of the longest original equation. While the height is determined by the longest equation and can not be reduced by using DSPs, the tree's width offers a large space for optimization and should be considered in relation to the amount of required resources.

B. Optimizing DSP implementation

The original equations share some bit operands, which results in performing the same operations multiple times. In other words, the computational trees contain some common sub-trees when the operations associated with them can be performed only once. The objective is to build the computational trees to contain as many common sub-trees as possible in order to remove redundant operations (or duplicates) and save resources.

1) Basic optimization: The first step to reduce the resources could be building the computational trees upon equations with their bit operands in ascending or descending order accordingly to their position in the input word. This operand rearrangement is possible due to the fact that the equations use only the XOR operation, which is commutative and associative. Because of similarities among the equations with sorted operands, this approach is capable of revealing a significant part of common computational sub-trees and contributes to the reduction of resources. Although the operand ordering is an easy way of optimization, it is a blind and locally driven approach and most likely will not result in optimal computational structures.

2) Proposed heuristic: We propose a solution leading to better results that introduces actual global relations between the original equations into the optimization process. Our approach, built on a version of Monte Carlo method, uses randomness tied to a large number of experiments. Concretely, we propose to build the computational trees by choosing random pairs of bit operands from a random original equation and tying them to the target tree. If another equation contains the same pair of operands, this pair would be tied to the corresponding tree as well to form a common sub-tree designated to removal during an optimization process. This approach successively leads to forming larger common sub-trees and to allowing deeper optimization by offering the best of many results.

The building of the computational trees is described by the Algorithm 1. It starts with the original equations as inputs and empty trees as outputs. The entire process is done in several steps resulting each to a lower level of the trees being generated. During each step we choose a random (not empty) input and then select a couple of random bit operands it includes. If there is only one bit operand left, it is associated to a XOR-neutral 0. This operands couple is removed from all inputs containing it and tied to the corresponding trees as a sub-tree representing a XORing operation with the operands and its result. The results are conserved and enter the next building step as inputs when the current one is over. Each step is finished when all its inputs contain no operands to process. The building process continues until all computational trees are complete. When the trees are built, all common sub-trees are removed and replaced by only one sub-tree for each

Algorithm 1 Computational trees building algorithm

```
1: level_inputs = orig_equations;
2: trees = empty_trees;
3: repeat
4:    while level_inputs not empty do
5:       couple = get_random_couple(level_inputs);
6:       for all level_input from level_inputs do
7:          remove_from_level_input(couple);
8:          result = trees_add_subtree(couple);
9:          append_to_level_output(result);
10:      end for
11:   end while
12:   level_inputs = level_outputs;
13: until trees complete
14: trees = eliminate_duplicates(trees);
15: score = evaluate_optimality(trees);
```

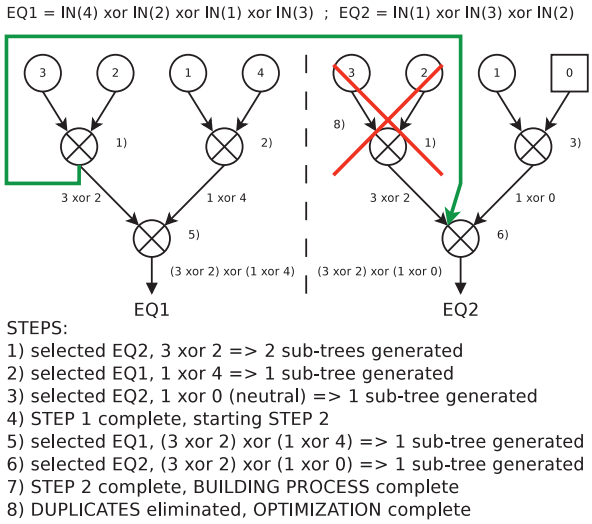

Fig. 2. Computational trees building process example

distinct involved operation. Finally, we compute the number of DSPs required to build the resulting computational trees and transform it into an optimality score. The less DSPs are needed, the higher the score is. An example process is shown in the Fig. 2.

The whole building process is repeated many times accordingly to the Monte Carlo method. The final result is chosen from all generated outputs as the one with the highest optimality score, which assures the highest reduction of resources. The output is written in the form of complete VHDL source code, which is ready for use in any firmware. The generator polynomial and the input data width are fixed, but the use of the DSPs' internal pipelining registers is optional via generic parameters. This allows to tune the frequency and the latency of the final CRC computation.

IV. Results

We use the CRC-32 for our experiments. This is one of the most common CRCs, using the generator polynomial of $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$. The output of the CRC-32 is 32 bits wide, which suffices for most hash table applications, since it is capable of addressing 2^{32} items. Our method is however generic and can be used for any other generator polynomial. The synthesis results are from Xilinx ISE 14.6.

TABLE I. Resources requirements for CRC-32

Input width	128 b	256 b	512 b	1024 b
LUTs with auto-optimization	347	711	1314	2433
DSPs with no optimization	45	87	172	344
DSPs with duplicates elimination	43	85	169	342
DSPs with operands ordering	33	63	122	241
DSPs with proposed heuristic	30	58	113	227

A. Using LUTs

When implementing the parallel CRC computation by using the general FPGA logic (LUTs), the computational structures are automatically created by the synthesizer with all the important optimizations being included. The concrete quantitative results for the CRC-32 are shown in the Table I (LUTs with auto-optimization), including numbers of LUTs required for different input widths. The resources requirements rise nearly linearly with the increasing number of input bits and become significant when this number is high. In that case, it is important to decide if such resource usage is acceptable or not. Thanks to the parallelism and simple operations being performed, the theoretic latency of such implementation is only a few nanoseconds (considering Xilinx Virtex-7 FPGA).

B. Using DSPs

The implementation in the DSPs is not as straightforward, which is caused mainly by the need of manual optimizations. When building the CRC-32 computational trees by using the ASAP scheduling and omitting any optimizations, we get the results that are shown in the Table I (DSPs with no optimization), including numbers of DSPs required for different input widths. The resources requirements rise almost linearly with the increasing number of input bits as in the case of LUTs. Notice that no LUTs are involved.

When using DSPs, some latency issues should be considered. Latency of such implementation is about four times longer than the latency of the implementation using LUTs. To get an acceptable frequency, pipelining can be engaged but the total latency increases because of additional registers. On the other hand, pipelining is natively supported by the DSPs (no CLB flip-flops are used) and can be used as advantage in applications that use pipelining and do not require low latencies.

1) Duplicates elimination: When we introduce the elimination of common sub-trees (or duplicates) in the computational trees, the numbers change. Eliminating the sub-trees cause a reduction of required resources, which, however, is only slight and demands improvement. The exact numbers for the CRC-32 are shown in the Table I (DSPs with duplicates elimination).

TABLE II. DSP TO LUT CRC-32 COMPARISON

Input width	128 b	256 b	512 b	1024 b
XOR (LUTs)	1735	3555	6570	12165
XOR (DSPs)	1440	2784	5424	10896
XOR/bit (LUTs)	13.55	13.88	12.83	11.87
XOR/bit (DSPs)	10.25	10.87	10.59	10.64
LUT/DSP	11.56	12.25	11.62	10.71

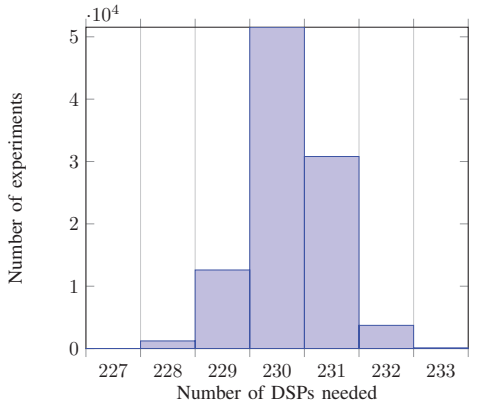

Fig. 3. Experimental histogram of the proposed heuristic for CRC-32 with 1024 input bits

2) Basic optimization: Building the computational trees upon the original equations with their bit operands arranged in ascending order brings a large improvement. The number of required DSPs decreases by tens of percents. The resulting values can be seen in the Table I (DSPs with operands ordering). Despite the reduction, when we consider the nature of this method, we notice that it can not be optimal.

3) Proposed heuristic: When using our proposed solution, we are able to decrease the resources requirements even more. By connecting the randomness to the large number of experiments and the global equation's relations, we get a powerful tool that is able to bring better results. The results for the CRC-32 from a hundred thousand experiments are shown in the Table I (DSPs with proposed heuristic). We managed to achieve an additional and stable reduction of resources, which is visible in the experimental histogram (Fig. 3).

4) Comparison to implementation in LUTs: While our results can not be directly compared to the implementation using LUTs, we can use an indirect comparison. Given the fact that a 6-input LUT can compute five XORs (XOR of all six input bits in any order), we can compute the number of XOR operations needed for the whole computation for both implementations in LUTs and DSPs. This can be further related to the number of input bits. The results for the CRC-32 are shown in the Table II. We compare only the best of our solutions (the last line of the Table I).

One can see that our solution saves around 11 LUTs per DSP block. Our XOR per input bit ratio is even better than that of synthesizer optimized LUT-based implementation. This is caused by the fact that our heuristic is specifically tailored to the particular domain and it performs a very intensive state space searching. Even if the state space searching can become time-consuming, it is performed only once and may reduce the time needed by the place and route process.

V. RELATED WORK

One of the most advanced general logic CRC implementations for FPGAs is described in [4]. It uses pipelining to achieve higher frequencies than the simple solution optimized by the synthesizer. The implementations of the CRC-32 for the input data width of 128 and 256 bits can run over 500 MHz and require 458 and 909 LUTs respectively at Virtex-6 FPGA.

Since the FPGA chips offer specialized logic blocks, there have been attempts to use them in various applications instead of the general logic. With focus on the hash functions, we can find solutions using diverse specialized blocks like DSPs or block memories to implement different cryptographic functions. The solution described in [5] uses block memories to store round constants, S-box and T-box tables and message expansion tables for the computation of various SHA-2 and SHA-3 functions. DSPs are used to compute sums in some of these functions. There are no available solutions using specialized blocks to implement the CRC based hashing.

VI. CONCLUSION

Our contribution is the design and experimental evaluation of the CRCs computation using the DSP FPGA blocks. This approach can be very useful in applications using hash tables and when the number of utilized LUTs needs to be reduced in order to fit another functionality to the firmware. Our solution is based on the capability of the DSP blocks to perform a large number of binary XORs. By using the Monte Carlo based heuristic, we are able to substitute around 11 LUTS by one DSP block. Our results further show that our solution performs less XOR operations than the solution with LUTs optimized by the synthesizer.

ACKNOWLEDGEMENT

This research has been supported by the BUT project FIT-S-14-2297, "CESNET Large Infrastructure" project no. LM2010005 funded by the Ministry of Education, Youth and Sports of the Czech Republic and the "DMON100" project no. TA03010561 funded by the Technology Agency of the Czech Republic.

REFERENCES

[1] D. E. Knuth, *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998.

[2] B. Mulvey, "Hash functions," http://home.comcast.net/~bretm/hash/, 2010, [Online].

[3] A. Perez, "Byte-wise crc calculations," *Micro, IEEE*, vol. 3, no. 3, pp. 40–50, 1983.

[4] H. F. A. Hamed, F. Elmisery, and A. A. H. A. Elkader, "Implementation of low area and high data throughput crc design on fpga," *International Journal of Advanced Research in Computer Science and Electronics Engineering (IJARCSEE)*, vol. 1, no. 9, 2012. [Online]. Available: http://ijarcsee.org/index.php/IJARCSEE/article/view/247

[5] R. Shahid, M. Sharif, M. Rogawski, and K. Gaj, "Use of embedded fpga resources in implementations of 14 round 2 sha-3 candidates," in *Field-Programmable Technology (FPT), 2011 International Conference on*, 2011, pp. 1–9.