Parallelism in a CRC Coprocessor

Andreas Döring
IBM Zurich Research Laboratory
ado@zurich.ibm.com

Abstract:
Cyclic Redundancy Checks (CRC) constitute an important class of hash functions for detecting changes in data blocks after transmission, storage and retrieval, or distributed processing. Currently, most sequential methods based on Horner's scheme are applied with some extensions or modifications. The flexibility of these methods with respect to the generator polynomial and the sequence of data processing is limited. A newly proposed algorithm and architecture [DW03, DW04] offer a high degree of flexibility in several aspects and provide high performance with a modest investment in hardware. The algorithm has inherent freedom for parallel processing on several levels, which is exploited in the proposed architecture. An early implementation gives quantitative results on cost and performance and suggests possible extensions and improvements. The algorithm, a typical system architecture, and the coprocessor's structure are described in this paper with an emphasis on parallelism.

1 Introduction

Cyclic Redundancy Checks (CRC) constitute a very widespread class of hash functions for verifying data integrity in data storage and communication applications. Although their property of covering all single- and double-bit and many multi-bit errors with a small number of check bits is highly desirable, their computation is costly when done in software. The growing integration of different applications, the stacking of protocols, and the high rate at which new protocols such as RDMA over TCP/IP, SCTP, and SCSI over IP, called iSCSI, are being introduced, limit the use of traditional hardware methods for CRC calculation. At the same time the speed of data communication standards has reached the 10-Gb/s region, and storage devices provide interfaces with a very high bandwidth. In such an environment the demands on both throughput and latency for the checksum calculation are high, whereas mobility and density require a low power consumption.

To meet this combination of high performance and flexibility requirements a new algorithm for CRC calculation [DW04] has been developed that allows efficient implementation with a high degree of parallelism on several levels. The algorithm allows the use of any generator polynomial up to a fixed degree (e.g. 32). Furthermore, the data words contributing to the checksum can be processed in any sequence. The algorithm can be used to maintain a checksum over individual updates of a data frame, and the checksum of a frame resulting from the concatenation of sub-frames with known checksum can be determined efficiently. This paper demonstrates how the instruction set architecture of the coprocessor

supports parallelism by enabling independent use of the coprocessor by multiple threads. Furthermore, on the next lower level, the application of known methods is illustrated for increasing parallelism in a special-purpose coprocessor, although these methods have been developed for general purpose processors, like instruction level SIMD parallelism or multithreading. On the lowest level, pipelining in the execution units and bit-level parallelism is found.

In this paper, the algorithm (Section 2), an assumed system architecture (Section 3), and the architecture of the coprocessor(Section 4) based thereon are presented with a focus on parallelism. The conclusion provides selected results from a prototype implementation.

2 Advanced CRC Algorithm

If we interpret a given data block A of length n as a vector of bits $(a_0, \ldots, a_{n-1})$ we can define a polynomial $F_A = \sum_{i=0}^{n-1} a_i x^i$ by convolution with a vector of symbolic powers of x. Given this vector and a generator polynomial p of degree $d(p)$ over the Galois field $GF(2)$, we obtain the CRC $C_p(A)$ as the remainder

$$C_p(A) = F_A \mod p = \sum_{i=0}^{n-1} a_i x^i \mod p \tag{1}$$

of division by p. Hence, the CRC checksum depends on both the input data frame and the generator polynomial, that is, each generator polynomial defines a CRC function. In most applications the bit a_0 is transmitted or written last and/or stored at the highest address in working memory. In order to detect data drop at the beginning of a data object, the checksum is sometimes defined with an added set of non-zero bits before the data frame, i.e. a length-dependent term qx^n with an appropriate polynomial q is added.

The traditional algorithm for calculating of a CRC checksum is based on Horner's Scheme. This means that a partial sum s is incrementally updated by adding another term. Starting with $s := a_{n-1}$, in the i-th step a term is added by $s := sx \mod p + a_{n-i-1}$, arriving at $C_p(A)$ in s after $n-1$ steps. To improve performance, word-level parallelism is added by working on words of width w. This changes the incremental step to $s := (sx^w \mod p) + x^{w-1}a_{i+w-1} + \ldots + a_i$. Only $\lceil n/w \rceil - 1$ steps are needed and i is incremented by w in each step. Moreover, in the initialization step up to w bits are used. The width w is chosen depending on the implementation method for the multiplication modulo p. In software, where look-up tables are preferred, w is typically rather small, e.g. 8. In hardware with a fixed polynomial, w can be greater, e.g. 32. The resulting circuit for the mod-p multiplier is a network of xor gates that can be optimized for cost by using common subexpressions. This technique is well established, and there are even free tools available on the Internet to generate a netlist for CRC calculation for a given polynomial p and word width w, see [Ea03]. A variant reported in [Gl97] chooses a polynomial r such that the multiplication modulo pr is simpler than the multiplication modulo p even though the partial sum s will have a higher degree and a final reduction step $s \mod p$ is needed. For applications with a fixed polynomial and sequential data appearance (streaming), the existing techniques

are very well suited and are for instance found in every Ethernet MAC (Media Access Contoller).

In other applications the main problem of Horner's scheme is its inherent sequential processing. As s is the input with the more complex computation in the incremental step, the result of the previous step has to be available before the next step can be started. A recent approach [CPR03] uses an additional operation involving multiplication with a fixed power of x for breaking up a data frame into several subblocks for parallel processing. For this step again the use of a look-up table is considered. Hence, by doubling the complexity — from one fixed-factor multiplication modulo p to two of them — thread-level parallelism for a fixed subtask size is achieved. The same method is used in [BW01] for adapting the processing granularity to the size of the data items (ATM cells). If the number of different supported polynomials grows, either a higher processing effort for the generation of the look-up tables on every change of the polynomial results or a high amount of memory or combinatorial circuitry is needed. For instance, for a degree-32 polynomial and $w = 8$, each constant multiplication requires 1024 bytes of memory (preferably in cache) and 8 to 10 instructions for each invocation for one input byte on a typical RISC processor.

The core concept of the new algorithm is to compute the terms $a_i x^i \mod p$ of the sum (1) directly. In particular, for all non-zero contributions a_i at a position i the term $a_i x^i \mod p$ is determined, and all those terms are summed up. During this process, computations can be saved by factoring out common terms x^k as shown below (`Op2`). This operation principle implies the need for a multiplication of two variable parameters (for instance the scaling factor x^i and the coefficient a_i) together with the modulo operation. To reduce the number of multiplication operations, selected powers of x are precomputed. In particular for each polynomial p the powers $x^{2^j} \mod p$ for a set of values j are determined and stored in an appropriate memory. The number of powers expressed by the possible values of j to be used depends on the maximum size of a data object and the addressing unit to identify the position in a frame. By using a two-factor multiplication, the evaluation of terms is not bound to any sequence and the universal character of powers-of-2 for the construction of the required x-powers allows any sequence for the generation of the individual scaling factors.

With the introduction of a two-factor polynomial multiplier the remaining problem is the modulo reduction with an arbitrary generator polynomial. In order to provide this flexible operation, the fact is exploited that modulo-p is a linear vector space operation:
$(a + b) \mod p = (a \mod p) + (b \mod p)$
$(\alpha a) \mod p = \alpha(a \mod p)$,
where α is an element of the base field.

Therefore, we can represent the modulo-operation as a vector-matrix multiplication, where the matrix depends on the polynomial p. Because $a \mod p = a$ if the degree of a is less than $d(p)$, the matrix representing the modulo operation consists of a lower part, which is a unity matrix (diagonal of ones), and an upper part, which maps the higher positions. Therefore, we can reduce the computational effort by using a matrix multiplication only on the upper part (starting at the degree of p) and adding the lower part afterwards. Altogether, we compute $t \mod p = t_h M_p + t_l$, where $t = t_h + t_l$ is an arbitrary product from the polynomial multiplier, d(t_l) $< d(p)$ and t_h is divisible by $x^{d(p)}$.

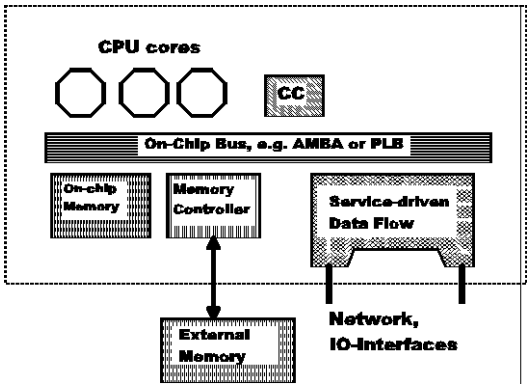

Figure 1: Structure of a System-On-Chip employing the proposed CRC coprocessor (CC)

As the vector-matrix multiplication operates on the result of the polynomial multiplication both multiplications can work in a pipelined manner. Also, both multipliers work on whole words exploiting bit-level parallelism. The third basic operation of the algorithm is the summing of terms, which is comparatively simple (bitwise exclusive or). Depending on the technology and clock frequency, this can even be done together with one of the multiplications within one clock cycle.

3 System Architecture

Both basic multiplication operations of the proposed algorithm require a considerable amount of hardware resources. Therefore, a high utilization of them is important for system efficiency. As only few applications will require a very high portion of CRC calculations for one thread or processor alone, sharing of one hardware instance of the CRC accelerator by several processors is required. Here, one area with a probable usage of a CRC accelerator protocol processing, in particular network processing, is considered. In [GDD+03] a framework for a network processor is presented that is based on standard processor cores enriched with hardware accelerators. The components relevant for this paper are sketched in Figure 1.

When a coprocessor is coupled to one or several processors, different degrees of coupling can be chosen. The difference between a tightly coupled and a loosely coupled coprocessor lies in the the number of instructions needed for the transfer of parameters and results and for synchronization. A shared coprocessor has an inherent uncertainty in timing, as seen from one processor by the competitive use by the other processors. Therefore, the typically higher design and circuit effort of a tight coupling cannot be fully translated into higher performance by optimally scheduling processor and coprocessor instructions. Consequently, a loosely coupled access, e.g. by memory mapping of coprocessor parameter, command and result registers, promises a better cost, design and performance compromise. To still achieve high single-thread performance, the frequency of interactions between the

coprocessor and the threads on the main processor has to be reduced. In particular, those instructions for which the progress of the thread on the main processor depends on the outcome of an coprocessor computation should be rare. Whereas commands and parameters for the coprocessor can be buffered in the coprocessor, typical processor cores do not have a mechanism for buffering coprocessor results that have not yet been requested.

In summary, for achieving a high parallelism between coprocessor and the general-purpose processor cores, the number of interactions between them should be minimized, and in particular result or state requests from the coprocessor should be rare. The processors (or processor threads) should be able to use the coprocessor independently from each other, i.e. without requiring locking.

4 Coprocessor Architecture

4.1 Core Operations

In order to reduce the number of results returned to the main processing threads, not only the calculation of a term of the sum $a_i x^i$ but also the summing of such terms are done in the coprocessor. The advantage lies less in the number of operations saved, which are just bitwise exclusive or, but rather in the more comfortable program organization: for instance, an application can construct a new data frame, and each time a word or data block is added, the corresponding contribution to the checksum is transferred to the coprocessor. Only when the entire frame is finished the processor retrieves the final checksum and stores it into the frame before transmission.

As discussed in the previous section, several threads should be able to use the coprocessor independently. This can be achieved by providing a set of checksum accumulation registers (CAR), where each register is used to hold a partial checksum. The way these registers are allocated to the threads is not discussed here, but several options are available, e.g. managed by the programmer, by a system library, or by hardware. In the latter case we can view the physical registers implemented in the coprocessor as a cache, and a much higher number can be kept in separate on- or off-chip memory.

One disadvantage of the proposed algorithm compared with Horner's Scheme is the potentially higher number of multiplications. If in a given frame of length n all terms $a_i x^i$ are computed directly, approximately $\frac{n}{2} \log n$ multiplications, compared with $n - 1$ multiplications for Horner's Scheme, would be needed. To avoid this, two extensions to the basic algorithm are used:

1. we add a cache for recently calculated powers of x, and
2. we provide two basic checksum-update operations,
   (a) `Op1`$(a_i, i, k) : s[k] := s[k] + a_i x^i \mod p[k]$
   (b) `Op2`$(a_i, i, k) : s[k] := x^i s[k] \mod p[k] + a_i$.

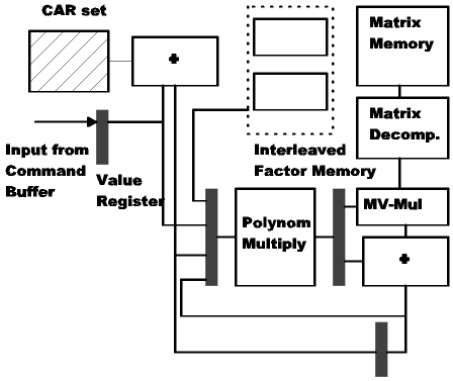

Figure 2: Core architecture data path: the gray boxes are pipeline registers

In these operations, $s[k]$ is the k-th CAR and $p[k]$ is a link between summing registers and generator polynomials. Thus, the polynomial is required only in an initialization instruction that sets $p[k]$ and clears $s[k]$ for the start of a new checksum. Whereas the first update operation `Op1` does what one would expect for accumulating terms of a sum, the second operation is more similar to Horner's Scheme by scaling the old sum and adding a new contribution. In data communication terms one can interpret it as appending $i - 1$ zeroes and a_i to an existing frame. In [DW04] a use of these two operations in combination with a more comfortable address-handling scheme is described.

The resulting architecture is shown in Figure 2. It shows the multipliers (MV-Mul is the vector-matrix multiplier for the mod reduction), the memories for the precomputed factors and matrices, and the checksum accumulation register set. A command buffer for decoupling computation and command acceptance is not pictured. Depending on the implementation, the exchange of the working matrix can require several clock cycles. To achieve a high efficiency, in these cases a large number of computations with the same matrix has to be carried out together. The command buffer can be used to rearrange the incoming commands in terms of this aspect. The details of parameter and result registers for the processor interface are omitted here, as they depend more on the bus and processor architecture than on the coprocessor. Since the matrix depends only on the polynomial, it can be compressed. The details of the decompressor are beyond the scope of this paper, but note that decompression is another process that can work in parallel with checksum calculation if double buffering of the matrix is used. Moreover, the matrix is quite large, e.g. for a word width of 32, the matrix consists of 992 (32×31) bits.

To take advantage of caching of powers of x, the product evaluation of $a_i x^i$ has to start with the computation of x^i before the coefficient a_i is used. Therefore, in the beginning of the computation of a new term, two factors are needed from the factor memory, unless i is a power of 2. As the two pipelined multiplications have a latency of 2 before their result can be used, another new multiplication can be started in the cycle following the first multiplication. If this scheme is used, eventually one product has to be delayed one cycle. Table 1 shows several examples of multiplication sequences.

Table 1: Examples of computation sequences for the determination of $a_i x^i$ with an interleaved factor memory

Cycle	$i = 119$	$i = 7$	$i = 2720$
0	(F0,F1)	(F1,F2)	(F5,F7)
1	(F4,F5)	–	(F9,F11)
2	(R,F2)	(R,F3)	–
3	(R,F6)	–	(R,D)
4	–	(R,C)	–
5	(R,D)	–	(R,C)
6	–	*	–
7	(R,C)		*
8	–		
9	*		

4.2 Operation Sequence

In Table 1, the factors for the polynomial multiplier are given, where Fj indicates a precomputed power $x^{2^j} \mod p$, R is the result of the vector-matrix-multiply, that is, the reduced product from the pair of factors from the step before the previous one. D indicates a delayed product, i.e. the reduced product from the third recent step, and C indicates the input coefficient a_i, which is always used as the last factor. In the cycle marked with an asterisk (*) the result is available. For example, for the exponent of 119, the factors 0, 1, 2, 4, 5 and 6 are needed, because these are the digits in the binary representation of 119, which are set to one. In cycle 0, two factors are taken from memory and their multiplication is started. As the result will not be available before cycle 2, another multiplication is started in cycle 1 with two more factors from memory. In cycles 2 and 3, the results from the previous cycles are fed back to the multiplier together with only one factor from memory. In cycle 4, no more new x-powers have to be processed, and the result is stored in the delay register. In cycle 5 the delayed product is multiplied with the result from cycle 3, the product of which ($x^{119} \mod p$) is available in cycle 7. At this point, the result can be stored in cache, and the multiplication with the coefficient can be started.

As a first important observation it can be seen that there are cycles, in which two new factors are needed from the factor memory, cycles with one precomputed factor, and cycles where the memory is not needed at all. If we have sufficiently long exponents, in most cycles one factor is needed because one multiplication can be started. Therefore, a dual-ported memory for the precomputed factors is not reasonable. To achieve a high probability that the required factors can be provided in one cycle anyway a two-way interleaved memory layout appears to be appropriate. In each bank of such a memory half of the factors for each polynomial are stored. As there are at most two cycles per term in which two factors from the memory are needed, an exponent must have at least two factors from each memory for optimal performance with such an interleaved memory. As a typical distribution of the factors, those belonging to even positions in the exponent binary

representation can be stored in one memory, and those for the odd positions in the other. This promises a good distribution of the factors, and has a comparably good behavior over the whole exponent range. An example with weak performance for this distribution is the case $i = 2720$ shown in Table 1. The table indicates the ideal case (e.g. with a dual-ported memory), when all factors would be immediately available. However, as all factors correspond to odd exponent digits, they would be stored in the same memory. Therefore, an intermediate register is needed to hold one factor while the second is read from memory. This creates an additional delay before the multiplication can start.

4.3 Multithreaded Operation

A second observation from the example computations is that there are cycles in which the multipliers are idle. The dash (–) indicates that no factors are sent to the polynomial multiplier and consequently the vector-matrix-multiplier is not needed in the subsequent cycle. As the multipliers are the most costly components in terms of chip area, it is desirable to use them extensively. Combining the two facts that both the factor memory and the multipliers have idle cycles in the computation of one term suggests the parallel computation of two or more terms. This can be compared to a multithreaded processor, where several instruction streams share common execution resources. While the next instructions in one thread have to wait for the availability of the result of the current instruction, the execution resources can be used for a different thread. An interesting characteristic of the CRC coprocessor is that there is no feedback from the data path to the control: the timing of the computation depends only on the input commands and can be planned — at least theoretically — ahead. If two terms are computed in parallel, the required use of the multipliers of one computation and the idle cycles in the other computation will not always fit perfectly. Therefore, additional registers are needed to store intermediate results. In the CRC coprocessor, this is the result of the addition after the vector-matrix multiply. Hence, this register can be viewed as a second delay register. The delay registers can also be used for the intermediate storage of factors from the factor memory. In this way the factor registers are free to accept parameters, for instance from a completed multiplication or from the checksum accumulator registers.

4.4 Extension: SIMD Parallelism

When polynomials of a low degree are used, only a small fraction of the multipliers is used, because most digits of the intermediate results and the precomputed factors are zero. Only for the input data value a_i all digits might be relevant. Consider, for example a degree of eight of the generator polynomial: for the polynomial multiplier this means that the product has only 15 relevant bits of 63 available in the architecture when two precomputed factors are multiplied. In consequence, the vector-matrix multiplication has to process only seven $(15 - 8)$ bits, and only eight result bits are relevant, resulting in a utilization of only 5% of the logic. To improve this utilization for low degrees, a SIMD-style parallelism can

be used. This is similar to the instruction set extensions for multimedia, such as SSE for Pentium processors, where a word is divided into subwords. For instance, if a coprocessor word is 32 bit, it can be split into four bytes. The multipliers are then used to multiply and reduce the corresponding bytes from each factor, i.e. the lowest bytes from each factor are multiplied, and in parallel the second bytes etc. The polynomial multiplier will already generate the corresponding partial products and their sums, but for full-word operation more terms are added. Therefore, to support the subword operation, some of the partial products have to be forced to be 0. In particular, if a 32-bit product is generated, result bit k is the sum of the partial products $f_i g_{k-i}$ for $0 \leq i \leq k$ when f and g are the factor words. For SIMD operation only those i and $k-i$ indices may be used that belong to the same byte as k. If $k' = k - (k \mod 8)$ is the index of the least significant bit of the subword to which k belongs, only the partial products for $k' \leq i \leq k' + 7$ are used. This separation of relevant partial products in SIMD and normal mode can be as simple as the insertion of one AND-gate into the summing tree for each resulting digit. The matrix-vector multiply itself needs no modification at all; it is sufficient to provide an appropriate matrix. However, the separation of the polynomial product into a lower and upper part requires an additional multiplexer as in SIMD mode there are four lower product parts and four upper product parts. Another modification applies to the factor memory, which has to be split into byte-wide memory blocks. This implies only a marginal size increase for the additional row decoders.

5 Conclusion

5.1 Implementation Results

To estimate the cost and the achievable performance, we have developed a VHDL prototype implementation targeting FPGA and standard cell technologies. So far it does not support caching but the functionality for standard operation as well as the SIMD mode has been verified with benchmarks. In Cu-11 technology [IB02] the core requires an area of $0.3\mathrm{mm}^2$ and achieves about 250 MHz. Synthesis results for Xilinx Virtex II [Xi03] devices indicate a possible clock frequency of about 100 MHz and about 1700 slices. In both cases all memories (precomputed factors, matrix and command buffers) are included. There is certainly room for optimization, especially in the controller of the core, which currently limits the clock frequency. Because of the mentioned absence of feed back from the data path, the controller could work ahead and plan the processing of the next commands beforehand. This would allow finer pipelining of the controller but requires additional design effort.

5.2 Outlook

For the FPGA implementation, a more detailed investigation of alternative configuration methods is being addressed. As the combinatorial logic in an FPGA is typically provided by small look-up tables (LUTs), one could reload parts of the vector-matrix multiplier LUTs instead of storing the matrix into registers and providing them as inputs to the LUTs. This reduces cost in the multiplier and could also improve clock speed, but the reconfiguration latency implied by switching from one polynomial to another is increased and the multiplier cannot be used during reconfiguration.

The coprocessor presented is a specialized component for a class of frequently found functions. Its architecture reveals parallelism on several levels, and trade-offs between cost and performance for various aspects allow tailoring the CRC coprocessor to different applications. We hope to gain a better understanding of the relevance of these trade-offs by simulating CRC-intensive applications.

References

[BW01] Braun, F. und Waldvogel, M.: Fast incremental CRC updates for ip over atm networks. In: *Proceeding of 2001 IEEE Workshop on High Performance Switching and Routing*. May 2001.

[CPR03] Campobello, G., Patane, G., und Russo, M.: Parallel CRC realization. *IEEE Transactions on Computers*. 52(10):1312–1319. October 2003.

[DW03] Döring, A. und Waldvogel, M. Method and apparatus for fast computation in a polynomial remainder ring with application to cyclic redundancy check including the possibility of updating the checksum after punctual modification. Patent Application No. 03405331.4. May 2003. European Patent Office.

[DW04] Döring, A. und Waldvogel, M.: Fast and flexible CRC calculation. *IEE Electronic Letters*. 40(1). January 2004.

[Ea03] Easics. CRC tool. http://www.easics.be/webtools/crctool. 2003. last modification, available much longer.

[GDD+03] Gabrani, M., Dittmann, G., Döring, A., Herkersdorf, A., Sagmeister, P., und van Lunteren, J.: Design methodology for a modular service-driven network processor architecture. *Computer Networks*. 41(5):623–640. April 2003.

[Gl97] Glaise, R. J.: A two-step computation of cyclic redundancy code CRC-32 for ATM networks. *IBM Journal of Research and Development*. 41(6):705–710. November 1997.

[IB02] IBM. Blue logic Cu-11 ASIC. www.chips.ibm.com. 2002.

[Xi03] Xilinx. Virtex™-II platform FPGAs: Complete data sheet. www.xilinx.com. October 2003.