RZ 3507 (# 99410) 10/06/03
Electrical Engineering 4 pages

Research Report

Fast and Flexible CRC Calculation

Andreas Doering and Marcel Waldvogel

IBM Research
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

Fast and Flexible CRC Calculation

Andreas Doering and Marcel Waldvogel

IBM Resarch, Zurich Research Laboratoy, 8803 Rüschlikon, Switzerland

Abstract: An algorithm for software or hardware implementation is presented, allowing fast computation of Cyclic Redundancy Checks with arbitrary polynomials and a high flexibility, such as updating of checksums after modifying data block parts with a known old checksum.

Introduction: Cyclic Redundancy Checks (CRCs) are important hash functions used in data storage and communications. A CRC is the remainder resulting from a division of a polynomial $A = (a_{l-1},\ldots,a_0)$, the raw data, over the Galois field with two elements GF(2) by a generator polynomial p:

$$C(A) = (\sum_{i=0}^{l-1} a_i x^i) \bmod p$$

Traditionally, CRCs are computed sequentially over all data, applying Horner's Rule:

$$C_{l-1}(A) = a_{l-1} \qquad C_{i-1}(A) = (xC_i(A) \bmod p) + a_{i-1} \qquad C(A) = C_0(A)$$

Efficient implementations use bit-level parallelism by working on words of an appropriate size w, combining w iterative steps into one. Hardware implementations typically use an XOR matrix [3] to implement the resulting function $F(a) = ax^w \bmod p$, while software prefers look-up tables [1]. In both cases the per-polynomial cost (reduction matrix respectively table memory space) is high. Furthermore, the runtime cost is proportional to l irrespective of the number of non-zero polynomial coefficients a_i. Here, an algorithm is presented whose complexity is linear in the input size for a sparse input vector A and logarithmic in the frame length. It supports multiple concurrent polynomials up to an arbitrary maximum degree with low cost per polynomial. Furthermore, it supports higher level functions such as update of a checksum after a modification of a data frame without revisiting the unaffected parts of the frame. In consequence, this algorithm provides a high flexibility for data handling and protocol design, because the sequence of inclusion into the checksum is arbitrary and sparse. It can be efficiently implemented with configurable logic or ASIC technology. Software implementation for current processor instruction sets is efficient only for frames of approximately 1 kB or more. The algorithm's use is shown in two application scenarios.

Core Operation: The core operation computed by the algorithm is $r = dx^{ua} \bmod p$ where u is a constant addressing unit, a is the address measured in this unit, d is the value at this address and r is the contributory result of d to the final checksum. The symbols r and d are represented in words of width w. The word width w determines the maximum supported degree of the generator polynomial.

The algorithm can be considered to be square-and-multiply [2] without squaring, modified as follows, using:

1. a set of pre-computed factors for each generator polynomial,
2. a generator-independent polynomial multiplication, and
3. a vector-matrix multiplication for generator-dependent modulo-reduction.

Altogether, the algorithm is as follows:

```
(1)  const word prec_factors[alength,pols];
(2)  const word matrix[wordlength-1,pols];
(3)  word calc_term(word d; address exp; int pol,degree)
(4)  {word factor,res;
(5)   int findex; // index into precomputed factor table
(6)   address h;
(7)   dword pprod;
(8)   res = d;
(9)   while (exp!=0) {
(10)    h=exp&(exp-1); findex=intlog2(h); // lowest bit number
(11)    exp=exp xor h;
(12)    pprod=polmul(res,prec_factors[findex,pol]);
(13)    res=mvmul(matrix[pol],pprod[degree..degree+w])
(14)         xor pprod[0..degree];}
(15)  return res;}
```

The internal loop (lines 9..14) computes the *x*-power with the corresponding exponent. Line (10) extracts one bit from the exponent, which implies that the number of iterations is limited by the number of bits representing the exponent. Each multiplication is split into a polynomial multiplication (`polmul`) and a matrix-vector multiplication (`mvmul`). Because the function $a \mapsto a \bmod p$ is a linear vector space projection, this covers all possible polynomials *p* by using an appropriate precomputed matrix. Since the lower digits of the polynomial product are already reduced, they can be added directly to the matrix-vector product, thereby reducing the size of the matrix.
Per polynomial *w(w-1)* memory bits are needed for the reduction matrix and $w \log_2 l$ bits for the precomputed factors.

Implementation Options: The algorithm can be implemented either in software or multiple hardware variants (application specific instruction set processor, field programmable gate array, or ASIC). Since typical processors support neither polynomial multiplication nor vector-matrix multiplications as part of their instruction sets, the modulo-p multiplication has to be done bitwise in software which is slower than the typical byte-wise lookup-table method used for the Horner's Rule.
For hardware implementation the two multiplications can be pipelined resulting in the following structure for the data path (Fig. 1).

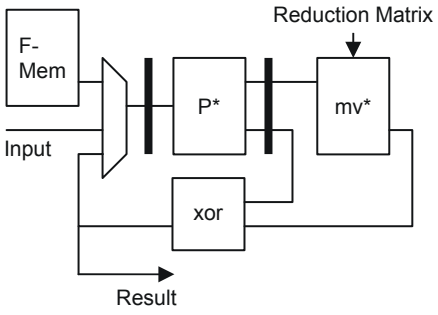

Fig. 1: Hardware implementation of core operation. Black bars: pipeline registers. p* is the polynomial, mv* the matrix/vector input multiplier.

This structure allows for high clock frequencies and a high degree of parallelism. Since there is no conditional feedback from the data path, the control circuits for generation of addresses and multiplexer control signals can run one clock cycle ahead for increased speed. The block marked "F-Mem" holds the precomputed factors. It feeds the multiplier pipeline through a multiplexer which either selects a precomputed factor, the primary input, or the product from a previous iteration.

Software Comfort Layer: In order to provide a higher abstraction layer when implementing a loosely coupled coprocessor, the following extensions to the core operation are suggested. Several (namely, *c*) registers for accumulating sums of core operation results support integrated layer processing and multi-threading. As each register accumulates only terms relating to the same polynomial, the polynomial (with degree) can be associated with the sum register to reduce the number of instruction parameters.
Furthermore, an address computation is valuable for communication tasks, in which a data frame is typically placed in memory such that the bit transmitted first and – because of the Horner's Rule – corresponding to the highest power of *x*, is located at the lowest address.

```
(1)  word sum[c];
(2)  address maxaddr[c];
(3)  word pols[c]; int degs[c];
(4)  update_word(address pos, word d, int reg) {
(5)    address e;word t,p,s;
(6)    e=abs(pos-maxaddr[c]);
(7)    if (pos>maxaddr)[c]) {t=sum[reg];s=d;}
(8)    else                 {s=sum[reg];t=d;}
(9)    maxaddr[c]=max(pos,maxaddr[c]);
(10)   t=calc_term(e,t,pols[reg],degs[reg]);
(11)   sum[reg]=s xor t;}
```

Therefore, the exponent needed in the core operation is the difference of the length of the data frame and the relative position taken from the beginning of the affected data bit. In order to allow the accumulation of data words in frame construction of unknown length, without loss of generality, the highest address seen so far is used as frame length.

Application to Communication Tasks: The characteristics of protocol processing in communication systems create typical usage pattern which can be supported by the proposed algorithm. When a data item is transferred through multiple subnetworks, parts of the message header containing local routing information have to be modified along the path. Because of the linearity of the modulo-operation, the exclusive-or difference of the new and the old value can be scaled according to the address of the modification and XORed to the existing checksum [4].
Another frequent case is the concatenation of a data frame from shorter pieces. Examples include iSCSI (Small Computer Systems Interface over Internet Protocol), where comparatively small commands to a storage device are transferred together with much longer data frames; address translation at iSCSI or Ethernet gateways; and integrated

layer processing, where CRCs of multiple overlaying protocol layers are to be computed in a single pass (e.g., iSCSI over Stream Control Transport Protocol (SCTP) over Ethernet. Given the mathematical properties of multiplication, our algorithm also allows precomputation of CRC contributions when packets arrive out of order. For this, each partial CRC is simply scaled according to the position of the packet relative to the entire message.

Conclusion: The proposed method for CRC calculation is especially efficient if support for polynomial and vector-matrix multiplications is available in hardware. Implementing these operations with the instruction sets of current processors is not as efficient and the advantage over a lookup-based Horner's Rule is reduced. As data packet sizes are increasing and sparse modifications become more common, the benefit of using our scheme will become apparent even in software. Using a common-source non-optimized prototype VHDL implementation a high performance was demonstrated reaching 100 and 250 MHz respectively in FPGA and standard cell (0.13-µm copper CMOS) technologies. The pipelining allows one multiplication per clock cycle. The ASIC core requires 0.35 mm^2 including the memories for matrices and precomputed factors as well as command buffering for four concurrent polynomials and 32-bit words.

References:

[1] TENKASI V. RAMABADRAN, SUNIL S. GAITONDE: A Tutorial on CRC Computations, IEEE Micro Vol. 8 No. 4 (July/August 1988) pp. 62-75

[2] KNUTH D. E.: The Art of Computer Programming – Seminumerical Algorithms, Addison Wesley, 1981

[3] SPRACHMANN M.: Automatic Generation of Parallel CRC Circuits, IEEE Design and Test of Computers Vol. 18 No. 3 (May/June 2001) pp. 108-114

[4] BRAUN F., WALDVOGEL M.: Fast Incremental CRC Updates for IP over ATM Networks, High Performance Switching and Routing (HPSR 2001), Dallas, TX, USA (May 2001) pp. 48-52.