Research Article

Galois Field Instructions in the Sandblaster 2.0 Architecture

Mayan Moudgill,¹ Andrei Iancu,¹ and Daniel Iancu^{1,2}

¹ Sandbridge Technologies, Inc., White Plains, NY 10601, USA

² Tampere University of Technology, Korkeakoulunkatu 1, 33720 Tampere, Finland

Correspondence should be addressed to Daniel Iancu, diancu@sandbridgetech.com

Received 20 November 2008; Revised 17 May 2009; Accepted 26 August 2009

Recommended by Mihai Sima

This paper presents a novel approach to implementing multiplication of Galois Fields with 2^N . Elements of $GF(2^N)$ can be represented as polynomials of degree less than N over $GF(2)$. Operations are performed modulo an irreducible polynomial of degree n over $GF(2)$. Our approach splits a Galois Field multiply into two operations, polynomial-multiply and polynomial-remainder over $GF(2)$. We show how these two operations can be implemented using the same hardware. Further, we show that in many cases several polynomial-multiply operations can be combined before needing to a polynomial-remainder. The Sandblaster 2.0 is a SIMD architecture. It has SIMD variants of the poly-multiply and poly-remainder instructions. We use a Reed-Solomon encoder and decoder to demonstrate the performance of our approach. Our new approach achieves speedup of 11.5x compared to the standard SIMD processor of 8x.

Copyright © 2009 Mayan Moudgill et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

1. Introduction

Galois Field arithmetic is widely used in applications such as error-correcting codes and cryptography. Generally, the Galois Fields used are $GF(2^N)$ for some N . Elements of $GF(2^N)$ may be represented as polynomials of degree less than N over $GF(2)$. Operations are performed modulo some polynomial P , where P is an irreducible polynomial of degree N over $GF(2)$. P is known as the prime polynomial. The multiplication of two elements X and Y can be accomplished by multiplying their polynomial representations, and then computing the remainder modulo P .

Conventionally, these polynomials are represented as binary numbers, where the i th term is represented by setting the i th bit to 1 or 0 depending on whether that term is present. Thus, the polynomial $x^4 + x + 1$ would be represented as 10011.

Addition of $GF(2^N)$ values in this representation is straightforward; it is simply an exclusive-or (xor) of the two binary numbers. Galois Field multiplication (GFM), however, is much more complicated. It involves the following steps.

- (i) Do a polynomial-multiply of two inputs.

- (ii) Do a polynomial-remainder of the product modulo a third input, the prime polynomial.

In software, GF multiplications are usually performed using Look-up Tables (LUTs). For large N , the LUT becomes rather large, requiring prohibitive large memory size. The processing time also becomes prohibitive at high data rates.

To further complicate issues, processors need to be able to handle Galois Fields of different lengths. Consequently, several processors have added instructions for Galois Field Multiplication (GFM). The most representative is the TI C64x DSP.

A general purpose GFM instruction needs 4 inputs, at least 3 of which must be in registers, the two inputs and the prime polynomial. Since most instruction-sets do not provide for 4 inputs fields, a GFM instruction generally uses a special-purpose register that provides either the length or the prime, or both. For instance, the GMPY4 operation in the TI C64x DSP uses the GFPGFR special purpose register to specify the length and prime polynomial [1].

In this paper, we describe an approach that implements GFM using two instructions, one of which implements polynomial-multiply over $GF(2)$, and the other of which implements the polynomial-remainder over $GF(2)$.

In the Sandblaster 2.0 architecture [2], these instructions are called `gfmul` and `gfnormi`, respectively. Both of these instructions use two register inputs. The `gfnormi` additionally has a third immediate input, the length of the polynomial.

The Sandblaster 2.0 has a 16-way SIMD unit; consequently, we also have 16 way variants of polynomial-multiply and polynomial-remainder instructions called `rgfmul` and `rgfnorm`. Additionally, the SIMD unit supports polynomial-multiply-and-add and polynomial-multiply-and-reduce instructions called `rgfmac` and `rgfmulred`.

It turns out that it is possible to specify the `gfmul` and `gfnormi` operations in such a fashion that we can use almost identical hardware to implement both functions. Consequently, there is no hardware overhead to split the GFM operation into two operations.

The Galois Field sum of several GFM operations can be simplified to the polynomial sum (i.e., xor) of several polynomial-multiplies, followed by a single polynomial remainder. This is quite common in several algorithms that use Galois Field arithmetic. In those cases, we can implement the sum of N GFM operations using N polynomial-multiplies and 1 polynomial-remainder, incurring a 1 instruction overhead because of our split implementation.

Section 2 of this paper describes the GFM instructions in the Sandblaster 2.0 architecture. Section 3 focuses on the implementation of the operations. Section 4 examines the performance of GFM instructions in the context of Reed-Solomon encoding/decoding. We conclude in Section 5.

2. Galois Field Multiply Instructions

The Sandblaster scalar unit has 16 32-bit general purpose registers. Like most RISC architectures, at most 2 registers can be read and 1 written per integer operation. An integer operation has fields to specify up to 3 registers. An extended immediate variant of the instruction can additionally provide up to 12 bits of immediate data.

2.1. Polynomial Representation. In the customary binary representation of polynomials over $GF(2)$, the bits are right-aligned, with the LSB bit representing the coefficient of term x^0 , and bit i representing the coefficient of term x^i . By contrast, we left-align the coefficients, so that the coefficient of the largest term of the polynomial is represented by MSB. For a polynomial of degree N , bit $31-i$ of the general purpose register represents the coefficient of term x^{N-i} .

Note that in this representation, without knowing the length of the polynomial, we cannot be sure which polynomial is represented by a specific number. For instance, `0xb000_0000` could be interpreted as $x^3 + x + 1$ if the polynomials are of degree 3 or $x^5 + x^3 + x^2$ if the polynomial is of degree 5.

We picked this representation to make it easier to compute the remainder. Since the high-order term of the divisor and dividend is left-aligned, we can start subtraction without requiring any shifting to line up the start of the polynomials.

For correctness, it is assumed that all unused bits in the register are 0. Both polynomial-multiply and -remainder are implemented so that they leave their results left-aligned with unused bits as 0.

There is one wrinkle about the representation. We assume that the polynomial remainder is performed with a left-aligned divisor so that the MSB is always 1. In this case, representing the leading coefficient is redundant. So, we do not represent the leading bit of a divisor polynomial. Instead the MSB represents the coefficient of the second highest term x^{N-1} . For instance, the divisor polynomial $x^6 + x^3 + 1$ is represented as `0x2400_0000`; that is, with the leading 6 bits being `001001`.

2.2. Polynomial Operations. The poly-multiply instruction in the Sandblaster architecture, `gfmul`, has the following format:

```
gfmul rc,ra,rb
```

It does a polynomial multiplication of the upper-most 8 bits of `ra` and with the upper 8 bits of `rb`, and writes the 15 bit result of the poly-multiply in the upper bits of the target register `rc`. The remaining bits of `rc` are zeroed.

The poly-remainder instruction in the Sandblaster architecture, `gfnormi`, has the format

```
gfnormi rt,rc,rp,J
```

The dividend is the 32-bit number formed by the upper 16 bits of `rc` right padded with 0. The divisor is the 17 bit number formed by prepending a 1 to the upper 16 bits of `rp`. `J` is an immediate operand ranging from 0 to 7. The `gfnormi` instruction performs $J + 1$ poly-division steps, leaving the remainder in the $16-(J + 1)$ upper-most bits of the target register `rt`.

2.3. Galois Field Multiplication. Implementing a GFM over $GF(2^K)$ with $K + 1$ bit prime polynomial P uses the following setup:

- (i) the product inputs are stored in the upper K bits of two registers, `ra`, `rb`,
- (ii) the leading bit of P is dropped and the remaining K bits are stored in the upper K bits of a register, `rp`,
- (iii) all unused bits are set to 0.

After executing the following code sequence, the final result of the GFM will be the upper K bits of `rt`:

```
gfmul rc,ra,rb
gfnormi rt,rc,rp,K-1
```

Table 1 shows an example of Galois Field multiplication over $GF(2^6)$ of the two numbers `101100` and `011011` with the prime polynomial `1001001`. This results in an intermediate product of `01111010100` with a final remainder of `101010`. In Table 1, the column on the right shows how the values will be stored in the corresponding registers.

TABLE 1: Galois field multiply GF(2⁶).

	10_1100	ra	0xb000_0000
⊗	01_1011	rb	0x6c00_0000
=	011_1101_0100	rc	0x7a80_0000
%	100_1001	rp	0x2400_0000
=	10_1010	rt	0xa800_0000

2.4. *SIMD*. The SIMD unit in Sandbridge 2.0 architecture has eight 16 × 16-bit SIMD registers as well as four accumulator registers. The instruction encoding for SIMD operations allows for 4 input fields. The SIMD unit allows three registers to be read and 2 to be written by an instruction.

The SIMD unit supports GFM through the *rgfmul* and *rgfnorm* instructions, which have the following format:

```
rgfmul vc, va, vb
rgfnorm vt, vc, vp, J
```

These instructions do 16 poly-multiplies/poly-remainders in parallel. Since the SIMD register elements are 16-bit wide, the *rgfmul* uses the upper 8 bits of each element, while the *rgfnorm* uses the entire 16 bits of the element. Other than that, their behavior is identical to the *gfmul*/*gfnormi* instructions.

Upto three SIMD registers can be read per cycle; we use the extra read-port to implement a poly-multiply-and-add instruction with the format:

```
rgfmac vc, va, vb, vs
```

The *rgfmac* instruction 16 poly-multiplies of the 16 elements of *va* and *vb*, and then poly-adds (xor's) the products with the corresponding elements of *vs*.

The SIMD unit has an idiom where the 16 results of element-wise operations (such as *rgfmul*) are combined together to form a single value that is written to the accumulator. The poly-multiply-and-sum-reduce instruction follows this idiom

```
rgfmulred act, va, vb
```

The 16 elements of *va* and *vb* are poly-multiplied together, and the 16 resulting products are poly-summed (xor-ed) together to form a single 16-bit value that is written to the accumulator register.

3. Implementation

The *gfnormi* and *gfmul* instructions can be implemented by the same block with very little overhead. As can be seen from the pseudocode in Algorithm 1, the algorithms for the two involve the same computational kernel and differ in their setup and controls. They are described in detail below.

3.1. *gfnormi*. The *gfnormi* instruction computes the remainder using polynomial long division. Since the values are left-aligned, we start the process at bit 31 of the dividend value.

```
gfp (ismul, ra, rb, J)
/*setup*/
if (ismul)
    res = 0 × 00000
    div = 0 × 00.rb [31:24]
    N = 8
else
    res = ra [31:17]
    div = rb [31:17]
    N = J + 1
/*shift/xor stages*/
for (i = 0; i < 8; i++)
    if (ismul)
        isxor = ra [31 - i]
        issh = true
    else
        isxor = res [15] && i < N
        issh = i < N
    if (isxor)
        res = (res << 1) ^ div
    else if (issh)
        res = (res << 1)
```

ALGORITHM 1: Gf-op pseudocode.

The divisor consists of a leading 1 and the upper 16 bits of the divisor register. The immediate argument to the *gfnormi* instruction specifies the number of divide steps executed, *J* + 1. For example, 5 steps of the division of 011.1101.0100 by 100.1001 will proceed as follows:

```
01111010100000000
00000000000000000
11110101000000000
10010010000000000
11001110000000000
10010010000000000
10111000000000000
10010010000000000
01010100000000000
00000000000000000
10101000000000000
```

At each step, the result is xor-ed with 0 or with the divisor, depending on the leading bit being 0/1. The result is then left-shifted by 1 to ensure that the remainder after the division step is left-aligned. Note that xor-ing with 0 is the identity operation; this results in just a left-shift. This is done when the intermediate remainder is smaller than the divisor.

3.2. *gfmul*. Each poly-multiply step needs to follow the same pattern as the poly-remainder so that much of the hardware is common. If we are going to *J* + 1 steps, we do the following:

- (i) the partial result is initialized to all 0s,
- (ii) the “divisor” at each step is one of the multiply inputs prepended with *J* + 1 zeroes,

TABLE 2: Polynomial operations in RS(208,188) encoder/decoder. In the decoder, by contrast, the number of poly remainder operations is 25% of the number of poly-multiplies. This allows us to get a 11.5x speed up.

Operation		Encoder	Decoder PGZ)	Decoder FMB one iteration
Galois field multiply		3384	8864	9267
Polynomial multiply	gfmul			
	rgfmul	203	134	134
	rgfmac	16	56	0
	rgfmulred		423	529
Total		219	613	663
Polynomial remainder	gfnormi		2	2
	rgfnorm	204	152	152
	Total	204	154	154
polynomial operations		423	767	817

The experiments that were run encode one RS(204,188) packet artificially introduce enough errors to require 8 erasures and then decode the packet. Note that this is the worst-case decode situation; in practice 98% of all packets have all syndromes equal to zero, so no error decoding is required.

Table 2 gives the details of the results of our experiments. In the encoder, the number of poly-remainder operations is almost the same as the poly-multiplies. Consequently, SIMD only achieves a 8x speed up, even though the SIMD variants of the polynomial instructions perform 16 poly-operations in parallel.

4.3. End-to-end Simulation Results. For the DVB-T case, for the highest bitrate of 31.67 Mbps, the decoder is called 21 763 times per second. The total number of cycles spent by the processor in vector mode for the GF operations only is less than 18 MHz (a fraction of the SBX processor capabilities) compared to 277 MHz in scalar mode. The iterative decoding algorithm was tested in the end-to-end DVB-T/H simulated system, specified by ETSI EN 744 V1.4.1 (2001-01). The simulations were performed by using the SBX simulation tools. Using our GF instructions, the total number of cycles per second consumed in the SBX processor, for the highest bit rates specified in the standards and assuming that every packet has eight errors and eight erasures, is the following: 29 MHz for the 31.67 Mbps DVB-T, 9 MHz for the 4.4 Mbps DVB-H including the optional second RS decoder at the link level.

5. Conclusions

The method for implementing GFM we have described, that implements a GFM using 2 instructions a poly-multiply and a poly-remainder, allows the addition of GFM to a standard architecture without the need to introduce a special purpose register for the GFM. Further, both of these instructions can be implemented using the same hardware block.

We have shown that, for some applications, it is not necessary to execute both a poly-multiply and a poly-remainder for each GFM. In the cases where the results of several GFM are added together, the products of the corresponding poly-multiplies are summed and then a single poly-remainder is used. In one specific case, only 25% of the poly-multiplies required a poly-remainder. Our simulation results indicate a speedup of 11.5x of the extended processor versus the standard processor.

References

- [1] Texas Instruments, Inc, "TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide," February 2008.
- [2] M. Moudgill, J. Glossner, S. Agrawal, and G. Nacer, "The sandblaster 2.0 architecture and SB3500 implementation," in *Proceedings of the Software Defined Radio Technical Forum (SDR Forum '08)*, Washington, DC, USA, October 2008.
- [3] D. Iancu, J. Glossner, and M. Moudgill, "A Method of Reed-Solomon Encoding and Decoding," European Patent EP1704647.
- [4] S. G. Wilson, *Digital Modulation and Coding*, Prentice-Hall, Upper Saddle River, NJ, USA, 1996.
- [5] S. B. Wicker, *Error Control Systems for Digital Communication and Storage*, Prentice-Hall, Englewood Cliffs, NJ, USA, 1995.
- [6] D. Iancu, M. Moudgill, J. Glossner, and J. Takala, "Efficient Reed-Solomon iterative decoder using galois field instruction set," in *Proceedings of the 8th Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS '08)*, vol. 5114 of *Lecture Notes in Computer Science*, pp. 126–135, Samos, Greece, July 2008.
- [7] D. Iancu, H. Ye, J. Glossner, M. J. Schulte, S. Mamidi, and J. Takala, "Improved spectral efficiency through iterative concatenated convolutional reed-solomon software decoding," in *Proceedings of the Joint 1st Workshop on Sensor Networks & Symposium on Trends in Communications (SymptoTIC '06)*, pp. 1–5, Bratislava, Slovakia, June 2006.
- [8] J. L. Massey, "Shift register synthesis and BCH decoding," *IEEE Transactions on Information Theory*, vol. 15, no. 1, pp. 122–127, 1969.
- [9] G. Forney Jr., "On decoding BCH codes," *IEEE Transaction on Information Theory*, vol. 11, no. 4, pp. 549–557, 1965.