Freescale Semiconductor
Application Note

Document Number: AN2926
Rev. 0, 01/2006

AltiVec™ Solutions to Sequential Problems: Calculating CRC with Scalable Congruent Equivalence Compression

by *Bo Lin*
Digital Systems Division
Freescale Semiconductor, Inc.
East Kilbride, Scotland

1 Introduction

Conventional parallel CRC (cyclic redundancy check) calculation methods are based on looking up multiple bits at a time. The lookup result is XORed to the successive input bit-string to generate a new index for the next lookup. That is, an index generation relies on the previous lookup result. [1][3][4].

For modern, high-performance processors, this data dependency causes bottlenecks in the CPU. For example, if a lookup (a load instruction) takes 3 cycles, then 4 data-dependent lookups will cost 12 cycles on any CPU architecture, but the cost for 4 data-independent lookups is much less. A pipelined super-scalar CPU can complete 4 data-independent table lookups in 7 cycles if one table lookup costs 3 cycles. For some other architectures supporting parallel multiple memory bank accesses, such as StarCore or TI C6xx, only 3 cycles are needed for the 4 data-independent table lookups by duplicating the table in 4 different memory banks under the assumption that one table lookup costs 3 cycles.

Although data-independency is desirable, data-dependency is inevitable because the process of problem solving needs

Contents

freescale™ semiconductor

related data. Instead of eliminating data-dependency, one way to ease this bottleneck is to postpone the occurrence of the dependency to make local operations data-independent, although the overall result is data-dependent.

This application note proposes an innovative method of postponing data dependency to calculate CRC. Although this method does not reduce the total number of operations involved in calculating CRC, it greatly improves the CPU's IPC (instructions per cycle) and, as a result, achieves higher performance.

2 Calculating CRC with Scalable Congruent Compression

2.1 Overview

Let $B(x) = b_0x^{N-1} + b_1x^{N-2} + \ldots + b_{N-2}x + b_{N-1}$ denote an N-bit binary string B to be processed; that is, $B = b_0\, b_1 \ldots b_{N-1}$. Let $G(x) = g_0x^M + g_1x^{M-1} + \ldots + g_{M-1}x + g_M$ denote a fixed $(M + 1)$-bit value. B's cyclic redundancy check (CRC) with respect to G is an M-bit value $CRC_M = c_0c_1 \ldots c_{M-1}$ where the c_i ($i = 0, 1, 2, \ldots, M - 1$) is the coefficients of the polynomial $CRC_M(x) = c_0x^{M-1} + \ldots + c_{M-2}x + c_{M-1} = B(x) \times x^M \bmod G(x)$. That is, the CRC_M is the remainder of left-shifting string $B(x)$ M bits and then divided by $G(x)$.

An L-bit parallel approach to calculate CRC over B can be achieved if B can be accessed in L bits as a unit (L-bit unit) and if it is feasible to build a 2^L entry table to store 2^L L-bit units' CRC_M values; that is, pre-calculating $\mathrm{LTU}(z) = CRC_M(z)$ for $z = 0, 1, 2, \ldots, 2^L - 1$. For example, if $L = 8$, then there is a byte-wise table lookup approach to calculate CRC.

This conventional approach can be illustrated in Figure 1 where each B_i is L-bit wide—that is, an L-bit unit—and the final remainder is the desired CRC_M. Also shown in Figure 1, the second lookup $\mathrm{LTU}(B_1')$ relies on the first lookup result $\mathrm{LTU}(B_0)$, the third lookup relies on the second lookup result, and so on.

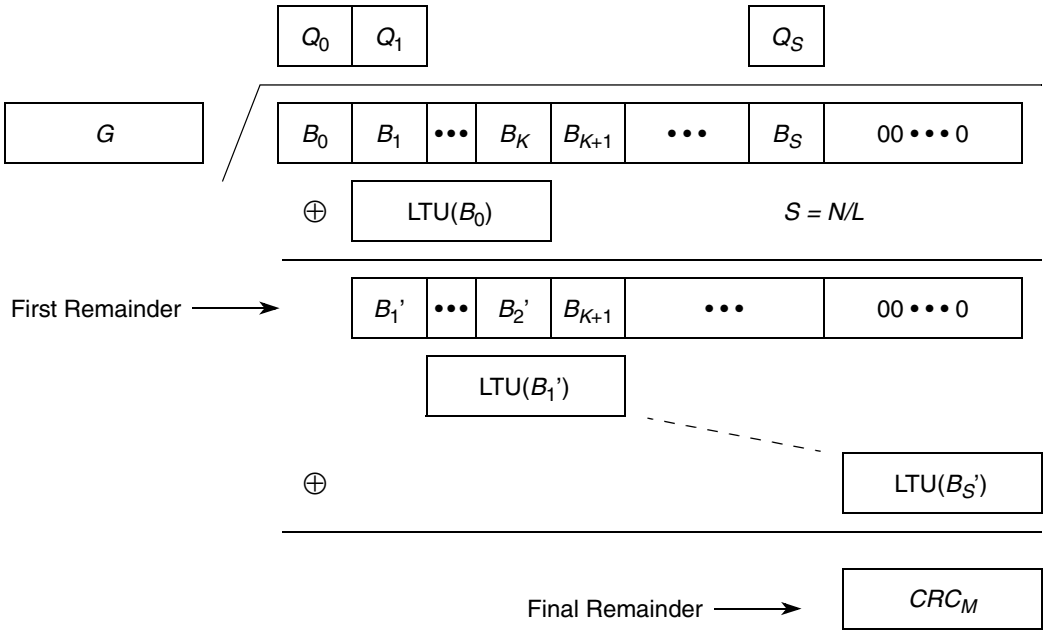

Figure 1. Calculating CRC with Table Lookups

The new method described in this document postpones the table lookup results for several (say, *K*) *L*-bit units to allow *K* data-independent lookups. This approach is illustrated in Figure 2. By building a compression table, *K* data-independent table lookups can be launched independently. Although this approach does not reduce required operations, it "squeezes" more instructions into a given number of CPU cycles and, as a result, achieves higher performance with increased IPC (instructions per cycle).

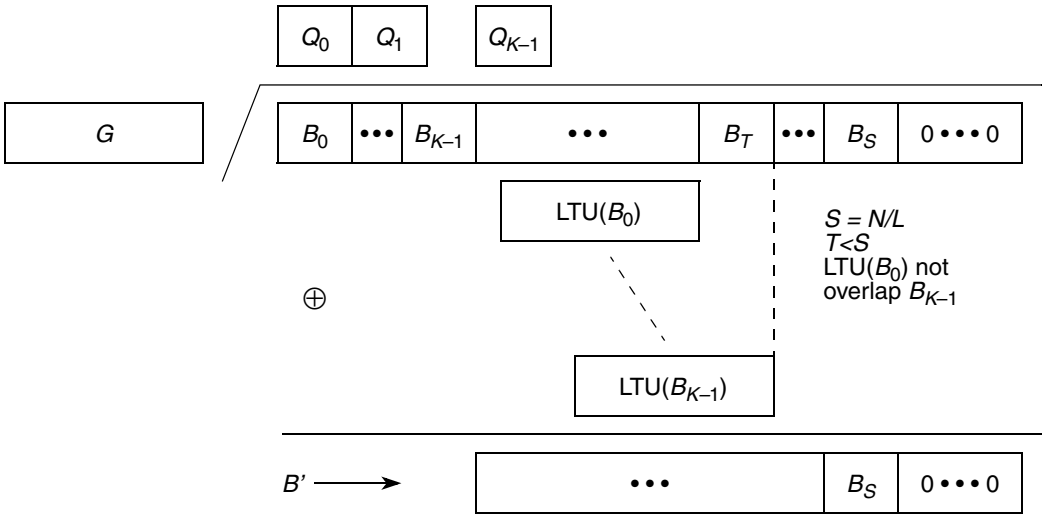

Figure 2. Calculating CRC with a Compression Table Lookup

The reason why the table lookup results can be postponed is that there is no carry propagation in an addition operation on polynomial-based $GF(2^n)$; that is, there is no carry propagation for ⊕ operations on binary strings. What is achieved is that *B*' has the same CRC_M as *B*. By recursively using the method illustrated in Figure 2, a long binary string *B* can be efficiently compressed into a short binary string *B*' with the same CRC_M value.

2.2 Congruent Equivalence

Definition 1

A(x) is congruent to *B(x)* modulo *G(x)* if and only if there exists a *Q(x)* such that *A(x)* – *B(x)* = *Q(x)G(x)*.

Corollary 1

If $A(x)$ mod $G(x) = R(x)$ and $B(x)$ mod $G(x) = R(x)$, then $A(x)$ is congruent to $B(x)$ modulo $G(x)$.

Proof:

$A(x)$ mod $G(x) = R(x)$ implies that there exists a polynomial $Q_1(x)$ such that $A(x) = Q_1(x)G(x) + R(x)$, and $B(x)$ mod $G(x) = R(x)$ implies that there exists a polynomial $Q_2(x)$ such that $B(x) = Q_2(x)G(x) + R(x)$. Hence, $A(x) - B(x) = (Q_1(x) - Q_2(x))G(x)$. This implies that there exists $Q(x) = Q_1(x) - Q_2(x)$ such that $A(x) - B(x) = Q(x)G(x)$. Hence, $A(x)$ is congruent to $B(x)$ modulo $G(x)$.

QED.

2.3 Scalable Congruent Compression

2.3.1 Assumptions

Assume a vector consisting of K L-bit units. $v_i(x) = B_{K*i}(x)\ B_{(K*i+1)}(x)\ B_{(K*i+2)}(x)\ \ldots\ B_{(K*i+K-1)}(x)$ can be used to represent the ith vector ($i = 0, 1, \ldots, m-1$), which is a concatenation of K L-bit units as shown in Figure 3, where each B_i is an L-bit unit. For example, if $L = 8$ and $K = 16$, an L-unit is a byte and a vector is an AltiVec™ vector. From Figure 3, the data frame $B(x)$ can be denoted as

$$\begin{aligned} B(x) &= b_0 x^{n-1} + b_1 x^{n-2} + \ldots b_{n-1} \\ &= B_0 x^{(S-1)L} + B_1 x^{(S-2)L} + \ldots B_{S-1} \\ &= (v_0(x)\, x^{(m-2)KL} + v_1(x)\, x^{(m-3)KL} + \ldots v_{m-2}(x))x^T + t(x) \\ &= W(x)\, x^T + t(x) \end{aligned}$$

where the $W(x)$ is the "integral part," and the $t(x)$ is a T-bit trailing "left-over part," as shown in Figure 3. That is, B needs m vectors to cover, t is shorter than a vector (a complete KL-bit unit), and W is the integral part occupied by the first $(m-1)$ complete vectors.

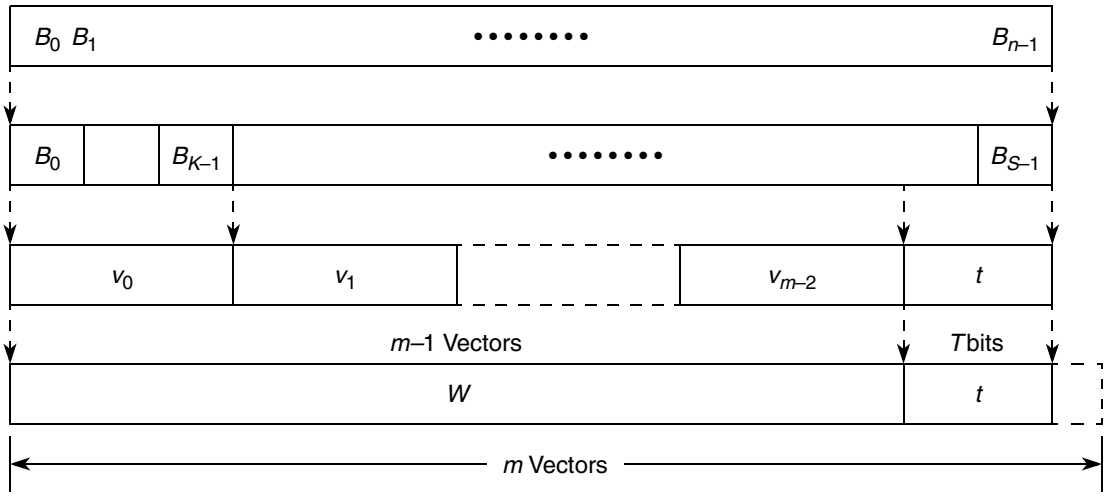

Figure 3. Alignment Convention

2.3.2 The Main Results

Theorem 1

$(W(x) - Q(x)G(x))x^T + t(x)$ is congruent to $B(x)$ modulo $G(x)$ where $Q(x)$ is arbitrary. That is, the congruence holds by subtracting $Q(x)G(x)$ from $B(x)$'s "integral part" $W(x)$.

Proof:

$$((W(x) - Q(x)G(x))\,x^T + t(x)) \bmod G(x)$$
$$= ((W(x) \bmod G(x) - \mathbf{Q(x)G(x) \bmod G(x)})\,x^T \bmod G(x) + t(x) \bmod G(x)) \bmod G(x)$$
$$= (W(x)x^T + t(x)) \bmod G(x)$$
$$= B(x) \bmod G(x).$$

By Corollary 1, $B(x)$ is congruent to $(W(x) - Q(x)G(x))x^T + t(x)$ modulo $G(x)$.

QED.

Theorem 1 states that, after subtracting a multiple of $G(x)$ in the "integral part" of the data frame, the same CRC value can still be calculated.

On the other hand, as shown in Figure 3, the integral part can be expressed as

$$W(x) = v_0(x)x^{(m-2)KL} + v_1(x)x^{(m-3)KL} + v_2(x)x^{(m-4)KL} + \ldots + v_{m-2}(x).$$

Consider the first item:

$$v_0(x)x^{(m-2)KL} = (B_0x^{(K-1)L} + B_1x^{(K-2)L} + \ldots B_{K-2}x^L + B_{K-1})x^{(m-2)KL}\ .$$

The goal here is to replace $v_0(x)x^{(m-2)KL}$ with one of its congruent polynomials with a lower degree. As a result, the integral part $W(x)$ will be reduced to $W'(x)$ with one less vector. In this way, the same CRC can be worked out over the resultant frame.

By introducing an integer C $(0< C < (m-2))$, the item is equivalent to

$$v_0(x)x^{(m-2)KL} = (B_0x^{CKL+(K-1)L} + B_1x^{CKL+(K-2)L} + \ldots + B_{K-2}x^{CKL+L} + B_{K-1}x^{CKL})x^{(m-2-C)KL}\ .$$

For each B_ix^{CKL}, $i = 0, 1, \ldots, K\text{-}1$, there exists a $q_i(x)$ such that $B_ix^{CKL} = q_i(x)G(x) + r_i(x)$ with $r_i(x)$'s degree less than $G(x)$'s degree; that is, $deg\ r_i(x) < M$.

This results in the following:

$$v_0(x)x^{(m-2)KL} - (q_0(x)\,G(x)x^{(K-1)L} + q_1(x)\,G(x)x^{(K-2)L} + \ldots + q_{K-1}(x)\,G(x))x^{(m-2-C)KL}$$
$$= (B_0x^{CKL+(K-1)L} + B_1x^{CKL+(K-2)L} + \ldots + B_{K-2}x^{CKL+L} + B_{K-1}x^{CKL})x^{(m-2-C)KL}$$
$$- (q_0(x)\,G(x)x^{(K-1)L} + q_1(x)\,G(x)x^{(K-2)L} + \ldots + q_{K-1}(x)\,G(x))x^{(m-2-C)KL}$$
$$= (r_0x^{(K-1)L} + r_1x^{(K-2)L} + \ldots r_{K-2}x^L + r_{K-1})x^{KL(m-2-C)}$$

It is equivalent to

$$v_0(x)x^{(m-2)KL} = (r_0x^{(K-1)L} + r_1x^{(K-2)L} + \ldots r_{K-2}x^L + r_{K-1})x^{(m-2-C)KL} \bmod G(x).$$

$r_i = B_ix^{CKL} \bmod G(x)$ can be pre-calculated and stored in a table LTU(B_i) for $B_i = 0, 1, \ldots 2^L - 1$, which applies to any L-bit unit. r_i = LTU(B_i) and r_{i+1} = LTU(B_{i+1}) may be overlapped as shown in Figure 2, but they use the same lookup table.

At this stage, v_0 is eliminated, and v_1, v_2 , …, v_C are updated as v_1', v_2' , …, v_C' with

$$W' = v_1'v_2' \ldots v_C'v_{c+1} \ldots v_{m-2} =$$
$$((v_1v_2 \ldots v_C) + (r_0x^{(K-1)L} + r_1x^{(K-2)L} + \ldots r_{K-2}x^L + r_{K-1})\, x^{(m-2-C)KL})$$
$$\| (v_{c+1} \ldots v_{m-2})$$

and

$$B(x) = W(x)x^T + t(x) = W'(x)x^T + t(x) \text{ modulo } G(x)$$

Theorem 2

$W(x) = W'(x)$ mod $G(x)$, where $W'(x)$ is the resultant binary string after eliminating the leading vector v_0 and shifting and XORing r0, r_1, … r_{K-1}.

Proof

See the above elaboration.

QED.

By recursively applying the above approach to the leading vector, an arbitrary long binary string is reduced into a binary string with C vectors as the integral part. The reduced binary string is congruent to the original string modulo $G(x)$.

In practice, C is chosen as 2. This makes the last table lookup result, LUT(B_{K-1}), right-aligned with v_2 and only v_1 and v_2 need updating. However, other values can be chosen to postpone the data-dependency further and they may bring advantages over C = 2.

3 Application to the PowerPC Core (Word-Wise Compression)

By setting L = 8, K = 4, C = 2 (CKL = 64) at the 32-bit word level, we compress an arbitrary binary string into a string of 8 to 11 bytes, where LUT(*byte*) = $byte*x^{64}$ *mod G*. Then the conventional *CRC* calculation is applied on the compressed binary string to achieve the final *CRC* result.

As illustrated in Figure 4, instead of calculating and accumulating (xoring) each byte's *CRC* value as described in [3] and [4], the word compression method compresses a 12-byte string P into an 8-byte string P' by using a pre-calculated table $wCcrc = byte*x^{32}$ mod G for *byte* = 0, 1, …, 255. The string *P'* has the same *CRC* value.

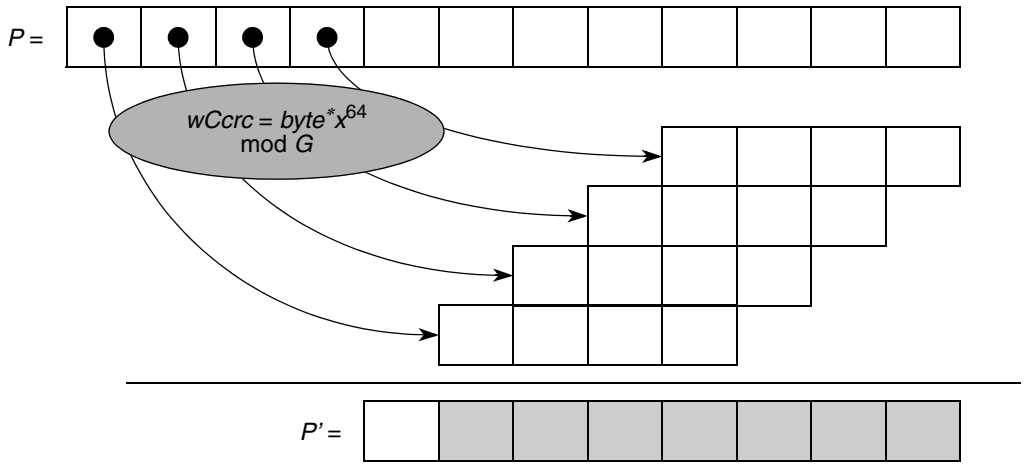

Figure 4. Word Compression

4 Application to AltiVec

By setting L = 8, K = 16, C = 2 (CKL = 256, and LUT(*byte*) = *byte**x^{256} *mod G*) at the vector level, a whole binary string is compressed into a shorter binary string such that v_{m-3}', v_{m-2}', v_{m-1} if v_{m-1} is not fully occupied by the input bit string.

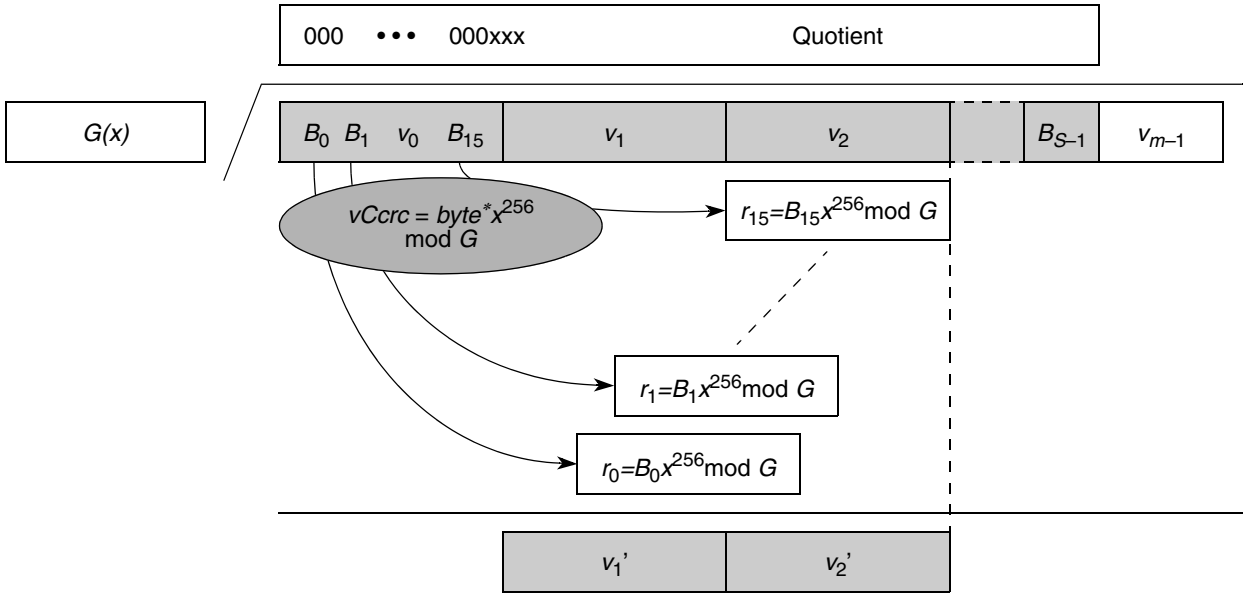

Figure 5. Long Division

The r_i (i = 0, 1, …, 15) in Figure 5 is the remainder of byte i, B_i, of vector j, v_j, times x^{256} with respect to $G(x)$; that is, $B_i * x^{256}$ mod $G(x)$. For the $G(x)$ with degree of M, each r_i occupies M bits.

AltiVec can carry out the 16 $B_i * x^{256} \bmod G(x)$ for $i = 0, 1, \ldots, 15$ in parallel by using

$$B_i * x^{256} = (H_i * x^4 + L_i) * x^{256} = H_i * x^{260} + L_i * x^{256} \bmod G$$

where H_i and L_i are B_i 's high-nibble and low-nibble respectively. This can be implemented by two 16-entry tables:

$$\text{LUTH}(H) = H * x^{260} \bmod G$$

and

$$\text{LUTL}(L) = L * x^{256} \bmod G$$

That is LUT(*byte*) = LTU(*HL*) = LUTH(*H*) + LUTL(*L*)

Although these remainders are scattered in the long division, they can be grouped together as shown in Figure 6 (*M* = 32, or *CRC*32, is used as an example).

As indicated in Figure 6, by pre-calculating $byte * x^{256} \bmod G(x)$—that is, the remainder of the byte at the second next (the next of next) vector—an arbitrarily long binary string can be compressed into a binary string with two complete vectors.

The resultant vectors and the remaining bytes form a new string with length between 32 and 47 bytes and it can be further compressed by using “word compression.”

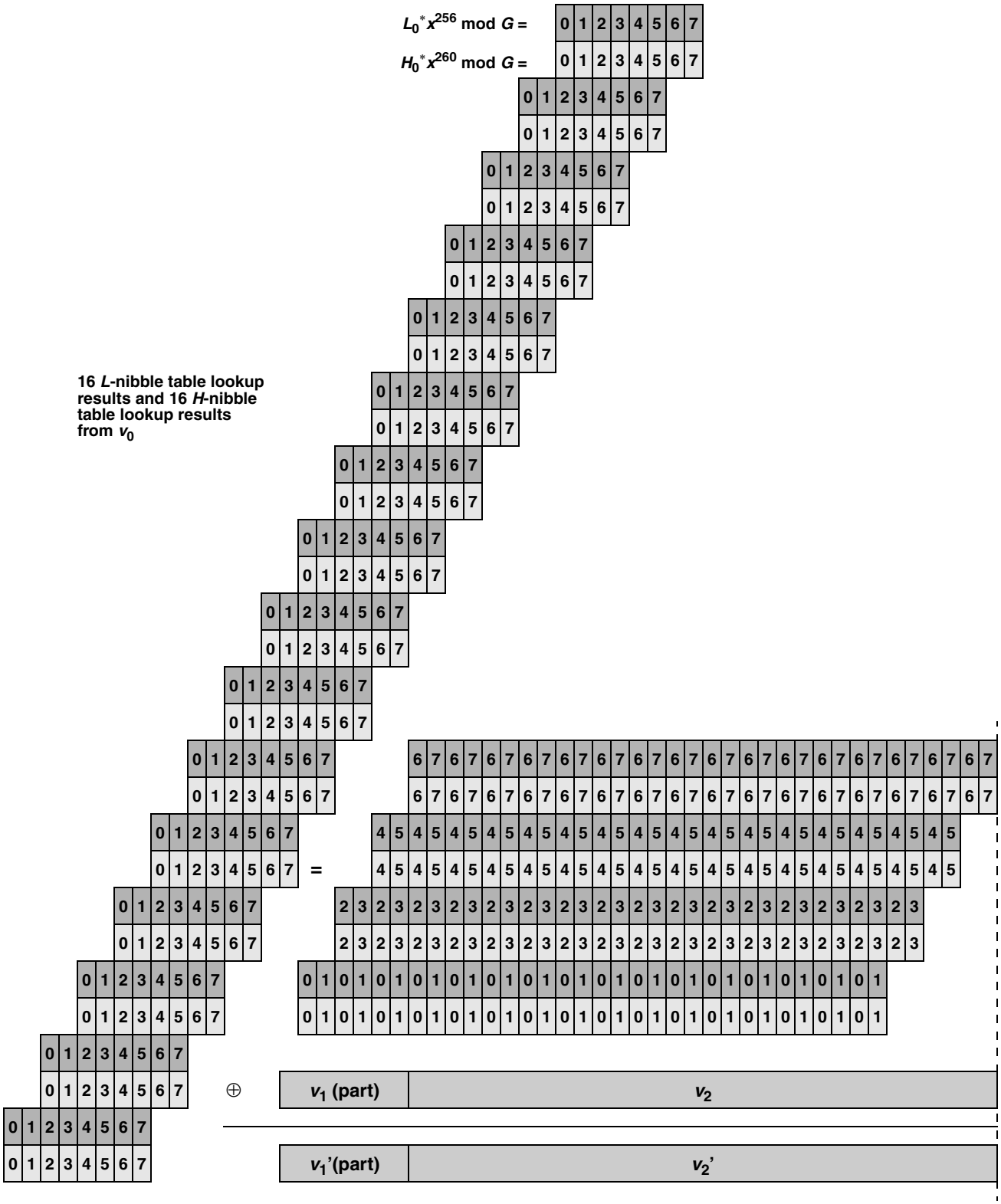

Figure 6. Vector Equivalence

5 Other Applications

The reduction approach described in Figure 2 has many applications in communications systems, for example, to calculate IEEE 802.3 FCS over a packet *B*. The following procedure can be used:

1. Replace the leading 32 bits of *B* with their one's complements.
2. Calculate $FCS = \sim(B(x)x^{32} \bmod G(x)) = \sim((W(x)\,x^R + t(x))\,x^{32} \bmod G(x))$. By replacing $W(x)$ with its two-vector-long reduction, $W'(x)$, we can calculate *CRC*32 of *B* by calculating *CRC*32 over $(W'(x)\,x^R + t(x))$, because $B(x)x^{32} \bmod G(x) = (W'(x)\,x^R + t(x))\,x^{32} \bmod G(x)$, where $W'(x)$ is two vectors long. The one's complement of the *CRC*32 is the FCS value.
3. Replace the leading 32 bits of *B* with their one's complements to recover its original value.

The above procedure can also be used to calculate iSCSI's CRC (RFC 3385) and SCTP's checksum (RFC 3309).

More applications can be found in BCH code or other algebraic codes for syndrome calculation.

6 Summary of CRC Calculation with Congruent Equivalence Compression

To calculate a CRC over an S-byte frame B0B1 … BS with a PowerPC, 32-bit, scalar core and AltiVec, a vector-level compression is carried out first by breaking the frame into vecs vectors and (S – 16·vecs) trailing bytes, because an AltiVec vector consists of 16 bytes. Then, the first (vecs – 2) vectors are eliminated using the method described in Section 4, "Application to AltiVec." The result is a frame with U = 32 + (S – 16·vecs) bytes B0'B1'… BU'.

The U bytes are broken into words and (U – 4 × m) trailing bytes. By applying the method described in Section 3, "Application to the PowerPC Core (Word-Wise Compression)," the w words are compressed into two or three words, that is, the U-byte string should be compressed into a range for which the conventional table lookup method is most efficient. Then, the conventional method is used to complete the final CRC calculation. This procedure is described in Figure 7.

A numerical example for CRC12 calculation is given in Section 8, "A Numerical Example for CRC12," where the parameter *c* is chosen as 3. As a result, the final frame is 13 bytes long.

Section 7, "Benchmark Results," provides a benchmark result for FCS that is a variant of textbook CRC32.

All the tables for CRC12 and CRC32 are listed in Appendices A and B. Other tables for CRC8, CRC16, CRC24 are available upon request.

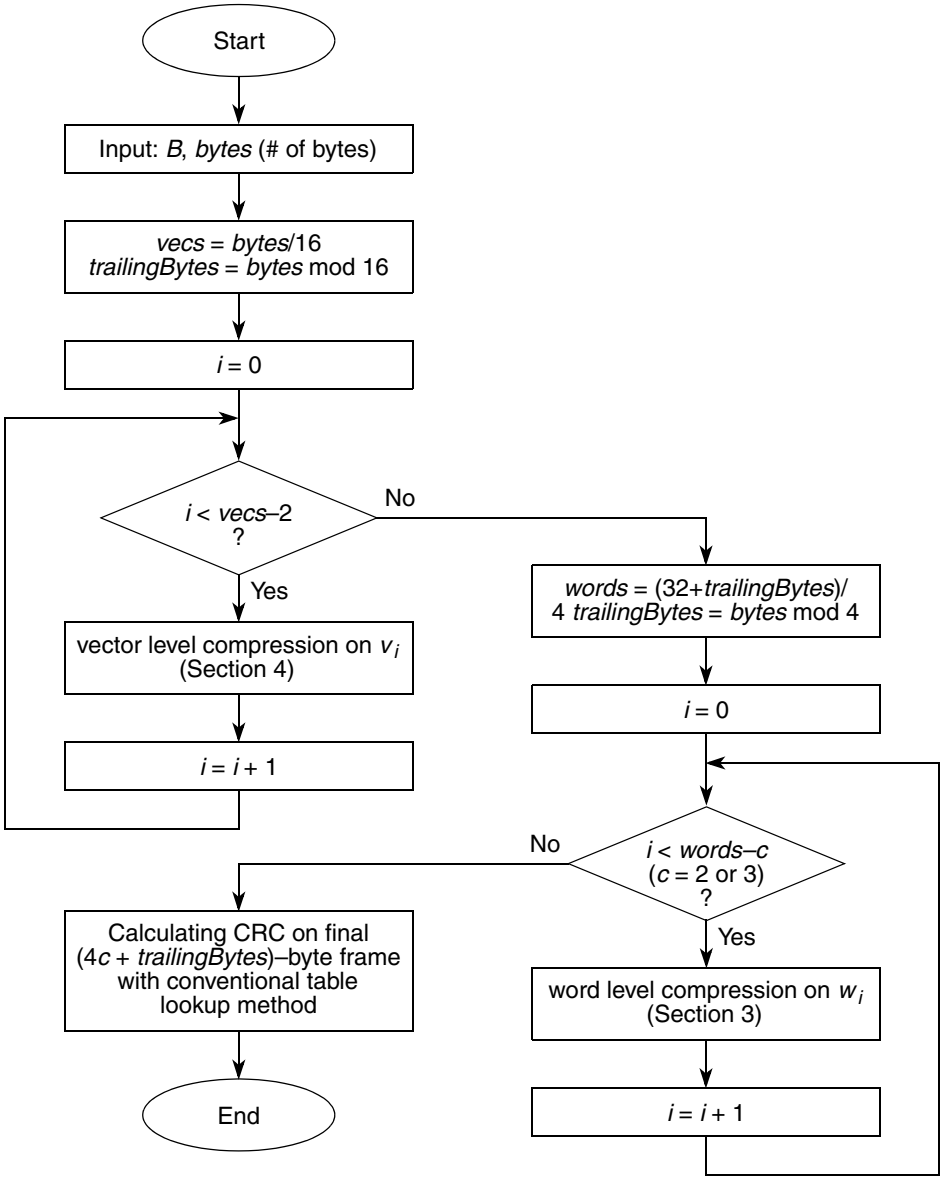

Figure 7. CRC Calculation with Congruent Equivalence Compression

7 Benchmark Results

An AltiVec-enabled implementation on CRC32 (FCS defined in [2]) is as shown in Table 1:

Table 1. 7457 Benchmark Results

Bytes	Cycles	Instructions	IPC	Bits/Cycle (v2)
8	56	52	0.93	1.14
11	78	83	1.06	1.13
12	82	110	1.34	1.17

Table 1. 7457 Benchmark Results (continued)

Bytes	Cycles	Instructions	IPC	Bits/Cycle (v2)
16	97	136	1.40	1.32
32	153	240	1.57	1.67
40	181	292	1.61	1.77
42	196	313	1.60	1.71
48	209	344	1.65	1.84
64	238	360	1.51	2.15
82	270	410	1.52	2.43
96	276	416	1.51	2.78
122	350	508	1.45	2.79
128	314	472	1.50	3.26
256	466	696	1.49	439
512	770	1144	1.49	5.32
768	1074	1592	1.48	5.72
1024	1378	2040	1.48	5.94
1536	1987	2936	1.48	6.18
2048	2594	3832	1.48	6.32
4096	5026	7416	1.48	6.52

The benchmark setup is

CPU: 7457 without L3 @1GHz

Language: ANSI C with AltiVec enabled

Compiler: gcc 3.3 on Linux

The shaded part is in the range specified by [2].

8 A Numerical Example for CRC12

The compression method can be illustrated as follows where **bold** style is for updated results. All the tables can be found in Appendix A.

```
P = v0:v1:v2:w0:B0
= 99F1E2D3 C4B5A697 08796A5B 4C3D2E1F:    (v0)
10111213 14151617 18191A1B 1C1D1E1F:    (v1)
20212223 24252627 28292A2B 2C2D2E2F:    (v2)
30313233:                               (w0)
84                                      (B0)
```

By using the traditional procedure, named crc12B, to calculate CRC12 on P, crc12B(P) = 0xF19 is obtained. However, the following steps achieve the same value:

1. Break v0 into two nibble vectors, vH and vL, such that

   vH = 9FED CBA9 0765 4321

   vL = 9123 4567 89AB CDEF

2. Launch table lookup with vH and vL to get the MS bytes

   vCcrc12_MSB(vH) = 0901060e09060109000f080007080f07

   vCcrc12_MSB(vL) = 0a090a030d04070e030a09000e07040d

3. vCcrc12_MSB(v0) = vCcrc12_MSB(vH) ⊕ vCcrc12_MSB(vL)

   = 03080c0d040206070305010090f0b0a

4. Launch table lookup with vH and vL to get the LS bytes

   vCcrc12_LSB(vH) = 8437fda36910da840079b3ed275e94ca

   vCcrc12_LSB(vL) = 9c79fd84f58c0871e59c18611069ed94

5. Sum up table lookup results as follows:

   (vCcrc12_MSB(v0)<<8) ⊕ (vCcrc12_LSB(vH) ⊕ (vCcrc12_LSB(vL))

   = 03:10420d239e9ad5f6e0e4ab85383c735e

6. Xor the above result to v1:v2, and P' is achieved with elimination of v0,

   P' = 10111213 14151617 18191a1b 1c1d1e**1c: (v1')**

   30632f00 babff3d1 c8cd81ae 14115d71: (v2')

   30313233: (w0)

   84 (B0)

   = w1:w2:w3:w4:

   w5:w6:w7:w8:

   w0:

   B0

7. Switch to word compression since less than three vectors are available to calculate CRC12 with (v1':v2') = (w1:w2:w3:w4:w5:w6:w7:w8)

8. Lookup w1 as:

   wCcrc12(10) = 46C

   wCcrc12(11) = 229

   wCcrc12(12) = 8E6

   wCcrc12(13) = EA3

9. Xor the above four table lookup results to w2:w3 to get w2':w3'

   w2':w3' = 14151617 18191a1b

   ⊕ 4 6C

   ⊕ 229

```
                ⊕              8E6
                ⊕                EA3
                 14151613 7638F2B8
```

and get a reduced string (w1 is eliminated)

```
P'' = 14151613:7638F2B8:18191a1b:1c1d1e1c:30632f00:
          babff3d1:c8cd81ae:14115d71:30313233:84
         = w2':w3':w4:w5:w6:w7:w8:w0:B0
```

10. Repeat 8 ~ 9 and the following string is achieved:

```
P''' = 7a1747a5: 0xaf156735:30313233:84
```

String `P'''` and string `P` have the same *CRC*12 value.

11. By applying conventional *CRC*12 on `P'''`, crc12B(`P'''`) = `0xF19`

As illustrated, the table lookups in Step 8 are data-independent. It is possible to push the table lookup result further to increase the data-independent lookup chain.

9 References

[1] G. Griffiths and G. C. Stones, “The Tea-Leaf Reader Algorithm: An Efficient Implementation of CRC-16 and CRC-32”, *Communications of the ACM*, vol. 30, No. 7, July 1987.

[2] IEEE Std 802.11-1997, “Wireless Medium Access Control (MAC) and Physical (PHY) Specifications.”

[3] T. V. Ramabadran, S. S. Gaitonde, “A Tutorial on CRC Computations”, *IEEE Micro*, August 1988.

[4] D. V. Sarwate, “Computation of Cyclic Redundancy Checks via Table Look-up,” *Communications of the ACM*, vol. 31, No. 8, August 1988.

10 Revision History

Table 2 provides a revision history for this application note.

Table 2. Document Revision History

Rev. Number	Date	Editor/ Writer	Substantive Change(s)
0	01/13/2006	MC/BL	Initial release.

Appendix A Tables for CRC12 Calculation

A.1 crc12

This table is used for conventional CRC12 calculation. It is used after a long packet is vector-compressed and word-compressed.

Each entry is four bytes (32 bits) wide, padded with 20 binary trailing zeros to facilitate left alignment. This arrangement can be changed to two bytes wide to save memory.

```
0x00000000, 0x80F00000, 0x81100000, 0x01E00000,
0x82D00000, 0x02200000, 0x03C00000, 0x83300000,
0x85500000, 0x05A00000, 0x04400000, 0x84B00000,
0x07800000, 0x87700000, 0x86900000, 0x06600000,
0x8A500000, 0x0AA00000, 0x0B400000, 0x8BB00000,
0x08800000, 0x88700000, 0x89900000, 0x09600000,
0x0F000000, 0x8FF00000, 0x8E100000, 0x0EE00000,
0x8DD00000, 0x0D200000, 0x0CC00000, 0x8C300000,
0x94500000, 0x14A00000, 0x15400000, 0x95B00000,
0x16800000, 0x96700000, 0x97900000, 0x17600000,
0x11000000, 0x91F00000, 0x90100000, 0x10E00000,
0x93D00000, 0x13200000, 0x12C00000, 0x92300000,
0x1E000000, 0x9EF00000, 0x9F100000, 0x1FE00000,
0x9CD00000, 0x1C200000, 0x1DC00000, 0x9D300000,
0x9B500000, 0x1BA00000, 0x1A400000, 0x9AB00000,
0x19800000, 0x99700000, 0x98900000, 0x18600000,
0xA8500000, 0x28A00000, 0x29400000, 0xA9B00000,
0x2A800000, 0xAA700000, 0xAB900000, 0x2B600000,
0x2D000000, 0xADF00000, 0xAC100000, 0x2CE00000,
0xAFD00000, 0x2F200000, 0x2EC00000, 0xAE300000,
0x22000000, 0xA2F00000, 0xA3100000, 0x23E00000,
0xA0D00000, 0x20200000, 0x21C00000, 0xA1300000,
0xA7500000, 0x27A00000, 0x26400000, 0xA6B00000,
0x25800000, 0xA5700000, 0xA4900000, 0x24600000,
0x3C000000, 0xBCF00000, 0xBD100000, 0x3DE00000,
0xBED00000, 0x3E200000, 0x3FC00000, 0xBF300000,
0xB9500000, 0x39A00000, 0x38400000, 0xB8B00000,
0x3B800000, 0xBB700000, 0xBA900000, 0x3A600000,
0xB6500000, 0x36A00000, 0x37400000, 0xB7B00000,
```

```
0x34800000, 0xB4700000, 0xB5900000, 0x35600000,
0x33000000, 0xB3F00000, 0xB2100000, 0x32E00000,
0xB1D00000, 0x31200000, 0x30C00000, 0xB0300000,
0xD0500000, 0x50A00000, 0x51400000, 0xD1B00000,
0x52800000, 0xD2700000, 0xD3900000, 0x53600000,
0x55000000, 0xD5F00000, 0xD4100000, 0x54E00000,
0xD7D00000, 0x57200000, 0x56C00000, 0xD6300000,
0x5A000000, 0xDAF00000, 0xDB100000, 0x5BE00000,
0xD8D00000, 0x58200000, 0x59C00000, 0xD9300000,
0xDF500000, 0x5FA00000, 0x5E400000, 0xDEB00000,
0x5D800000, 0xDD700000, 0xDC900000, 0x5C600000,
0x44000000, 0xC4F00000, 0xC5100000, 0x45E00000,
0xC6D00000, 0x46200000, 0x47C00000, 0xC7300000,
0xC1500000, 0x41A00000, 0x40400000, 0xC0B00000,
0x43800000, 0xC3700000, 0xC2900000, 0x42600000,
0xCE500000, 0x4EA00000, 0x4F400000, 0xCFB00000,
0x4C800000, 0xCC700000, 0xCD900000, 0x4D600000,
0x4B000000, 0xCBF00000, 0xCA100000, 0x4AE00000,
0xC9D00000, 0x49200000, 0x48C00000, 0xC8300000,
0x78000000, 0xF8F00000, 0xF9100000, 0x79E00000,
0xFAD00000, 0x7A200000, 0x7BC00000, 0xFB300000,
0xFD500000, 0x7DA00000, 0x7C400000, 0xFCB00000,
0x7F800000, 0xFF700000, 0xFE900000, 0x7E600000,
0xF2500000, 0x72A00000, 0x73400000, 0xF3B00000,
0x70800000, 0xF0700000, 0xF1900000, 0x71600000,
0x77000000, 0xF7F00000, 0xF6100000, 0x76E00000,
0xF5D00000, 0x75200000, 0x74C00000, 0xF4300000,
0xEC500000, 0x6CA00000, 0x6D400000, 0xEDB00000,
0x6E800000, 0xEE700000, 0xEF900000, 0x6F600000,
0x69000000, 0xE9F00000, 0xE8100000, 0x68E00000,
0xEBD00000, 0x6B200000, 0x6AC00000, 0xEA300000,
0x66000000, 0xE6F00000, 0xE7100000, 0x67E00000,
0xE4D00000, 0x64200000, 0x65C00000, 0xE5300000,
0xE3500000, 0x63A00000, 0x62400000, 0xE2B00000,
0x61800000, 0xE1700000, 0xE0900000, 0x60600000
```

A.2 wCcrc12

This table is used for word (4-byte) compression. It is used after a long packet is vector-compressed.

Each entry is two bytes (16 bits) wide, padded with four binary leading zeros to facilitate right alignment.

```
0x0000, 0x0645, 0x0C8A, 0x0ACF, 0x011B, 0x075E, 0x0D91, 0x0BD4,
0x0236, 0x0473, 0x0EBC, 0x08F9, 0x032D, 0x0568, 0x0FA7, 0x09E2,
0x046C, 0x0229, 0x08E6, 0x0EA3, 0x0577, 0x0332, 0x09FD, 0x0FB8,
0x065A, 0x001F, 0x0AD0, 0x0C95, 0x0741, 0x0104, 0x0BCB, 0x0D8E,
0x08D8, 0x0E9D, 0x0452, 0x0217, 0x09C3, 0x0F86, 0x0549, 0x030C,
0x0AEE, 0x0CAB, 0x0664, 0x0021, 0x0BF5, 0x0DB0, 0x077F, 0x013A,
0x0CB4, 0x0AF1, 0x003E, 0x067B, 0x0DAF, 0x0BEA, 0x0125, 0x0760,
0x0E82, 0x08C7, 0x0208, 0x044D, 0x0F99, 0x09DC, 0x0313, 0x0556,
0x09BF, 0x0FFA, 0x0535, 0x0370, 0x08A4, 0x0EE1, 0x042E, 0x026B,
0x0B89, 0x0DCC, 0x0703, 0x0146, 0x0A92, 0x0CD7, 0x0618, 0x005D,
0x0DD3, 0x0B96, 0x0159, 0x071C, 0x0CC8, 0x0A8D, 0x0042, 0x0607,
0x0FE5, 0x09A0, 0x036F, 0x052A, 0x0EFE, 0x08BB, 0x0274, 0x0431,
0x0167, 0x0722, 0x0DED, 0x0BA8, 0x007C, 0x0639, 0x0CF6, 0x0AB3,
0x0351, 0x0514, 0x0FDB, 0x099E, 0x024A, 0x040F, 0x0EC0, 0x0885,
0x050B, 0x034E, 0x0981, 0x0FC4, 0x0410, 0x0255, 0x089A, 0x0EDF,
0x073D, 0x0178, 0x0BB7, 0x0DF2, 0x0626, 0x0063, 0x0AAC, 0x0CE9,
0x0B71, 0x0D34, 0x07FB, 0x01BE, 0x0A6A, 0x0C2F, 0x06E0, 0x00A5,
0x0947, 0x0F02, 0x05CD, 0x0388, 0x085C, 0x0E19, 0x04D6, 0x0293,
0x0F1D, 0x0958, 0x0397, 0x05D2, 0x0E06, 0x0843, 0x028C, 0x04C9,
0x0D2B, 0x0B6E, 0x01A1, 0x07E4, 0x0C30, 0x0A75, 0x00BA, 0x06FF,
0x03A9, 0x05EC, 0x0F23, 0x0966, 0x02B2, 0x04F7, 0x0E38, 0x087D,
0x019F, 0x07DA, 0x0D15, 0x0B50, 0x0084, 0x06C1, 0x0C0E, 0x0A4B,
0x07C5, 0x0180, 0x0B4F, 0x0D0A, 0x06DE, 0x009B, 0x0A54, 0x0C11,
0x05F3, 0x03B6, 0x0979, 0x0F3C, 0x04E8, 0x02AD, 0x0862, 0x0E27,
0x02CE, 0x048B, 0x0E44, 0x0801, 0x03D5, 0x0590, 0x0F5F, 0x091A,
0x00F8, 0x06BD, 0x0C72, 0x0A37, 0x01E3, 0x07A6, 0x0D69, 0x0B2C,
0x06A2, 0x00E7, 0x0A28, 0x0C6D, 0x07B9, 0x01FC, 0x0B33, 0x0D76,
0x0494, 0x02D1, 0x081E, 0x0E5B, 0x058F, 0x03CA, 0x0905, 0x0F40,
0x0A16, 0x0C53, 0x069C, 0x00D9, 0x0B0D, 0x0D48, 0x0787, 0x01C2,
0x0820, 0x0E65, 0x04AA, 0x02EF, 0x093B, 0x0F7E, 0x05B1, 0x03F4,
0x0E7A, 0x083F, 0x02F0, 0x04B5, 0x0F61, 0x0924, 0x03EB, 0x05AE,
0x0C4C, 0x0A09, 0x00C6, 0x0683, 0x0D57, 0x0B12, 0x01DD, 0x0798
```

A.3 vCcrc12

Four vectors are needed to store the vector compression table to facilitate AltiVec's vperm instruction. There are two logical tables, one for L-nibbles and the other for H-nibbles. The two-byte result of a parallel nibble lookup needs two vectors.

```
vCcrc12 for L-nibbles
MS byte (byte 0) vector =
0x00, 0x09, 0x0A, 0x03, 0x0D, 0x04, 0x07, 0x0E,
0x03, 0x0A, 0x09, 0x00, 0x0E, 0x07, 0x04, 0x0D
LS byte (byte 1) vector =
0x00, 0x79, 0xFD, 0x84, 0xF5, 0x8C, 0x08, 0x71,
0xE5, 0x9C, 0x18, 0x61, 0x10, 0x69, 0xED, 0x94
vCcrc12 for H-nibbles
MB byte (byte 0) vector =
0x00, 0x07, 0x0F, 0x08, 0x07, 0x00, 0x08, 0x0F,
0x0E, 0x09, 0x01, 0x06, 0x09, 0x0E, 0x06, 0x01


LS byte (byte 1) vector =
0x00, 0xCA, 0x94, 0x5E, 0x27, 0xED, 0xB3, 0x79,
0x4E, 0x84, 0xDA, 0x10, 0x69, 0xA3, 0xFD, 0x37
```

Appendix B Tables for CRC32 Calculation

B.1 Tcrc32

This table is used for conventional CRC12 calculation. It is used after a long packet is vector-compressed and word-compressed. Each entry is four bytes (32 bits) wide.

```
0x00000000, 0x04C11DB7, 0x09823B6E, 0x0D4326D9,
0x130476DC, 0x17C56B6B, 0x1A864DB2, 0x1E475005,
0x2608EDB8, 0x22C9F00F, 0x2F8AD6D6, 0x2B4BCB61,
0x350C9B64, 0x31CD86D3, 0x3C8EA00A, 0x384FBDBD,
0x4C11DB70, 0x48D0C6C7, 0x4593E01E, 0x4152FDA9,
0x5F15ADAC, 0x5BD4B01B, 0x569796C2, 0x52568B75,
0x6A1936C8, 0x6ED82B7F, 0x639B0DA6, 0x675A1011,
0x791D4014, 0x7DDC5DA3, 0x709F7B7A, 0x745E66CD,
0x9823B6E0, 0x9CE2AB57, 0x91A18D8E, 0x95609039,
0x8B27C03C, 0x8FE6DD8B, 0x82A5FB52, 0x8664E6E5,
```

```
0xBE2B5B58, 0xBAEA46EF, 0xB7A96036, 0xB3687D81,
0xAD2F2D84, 0xA9EE3033, 0xA4AD16EA, 0xA06C0B5D,
0xD4326D90, 0xD0F37027, 0xDDB056FE, 0xD9714B49,
0xC7361B4C, 0xC3F706FB, 0xCEB42022, 0xCA753D95,
0xF23A8028, 0xF6FB9D9F, 0xFBB8BB46, 0xFF79A6F1,
0xE13EF6F4, 0xE5FFEB43, 0xE8BCCD9A, 0xEC7DD02D,
0x34867077, 0x30476DC0, 0x3D044B19, 0x39C556AE,
0x278206AB, 0x23431B1C, 0x2E003DC5, 0x2AC12072,
0x128E9DCF, 0x164F8078, 0x1B0CA6A1, 0x1FCDBB16,
0x018AEB13, 0x054BF6A4, 0x0808D07D, 0x0CC9CDCA,
0x7897AB07, 0x7C56B6B0, 0x71159069, 0x75D48DDE,
0x6B93DDDB, 0x6F52C06C, 0x6211E6B5, 0x66D0FB02,
0x5E9F46BF, 0x5A5E5B08, 0x571D7DD1, 0x53DC6066,
0x4D9B3063, 0x495A2DD4, 0x44190B0D, 0x40D816BA,
0xACA5C697, 0xA864DB20, 0xA527FDF9, 0xA1E6E04E,
0xBFA1B04B, 0xBB60ADFC, 0xB6238B25, 0xB2E29692,
0x8AAD2B2F, 0x8E6C3698, 0x832F1041, 0x87EE0DF6,
0x99A95DF3, 0x9D684044, 0x902B669D, 0x94EA7B2A,
0xE0B41DE7, 0xE4750050, 0xE9362689, 0xEDF73B3E,
0xF3B06B3B, 0xF771768C, 0xFA325055, 0xFEF34DE2,
0xC6BCF05F, 0xC27DEDE8, 0xCF3ECB31, 0xCBFFD686,
0xD5B88683, 0xD1799B34, 0xDC3ABDED, 0xD8FBA05A,
0x690CE0EE, 0x6DCDFD59, 0x608EDB80, 0x644FC637,
0x7A089632, 0x7EC98B85, 0x738AAD5C, 0x774BB0EB,
0x4F040D56, 0x4BC510E1, 0x46863638, 0x42472B8F,
0x5C007B8A, 0x58C1663D, 0x558240E4, 0x51435D53,
0x251D3B9E, 0x21DC2629, 0x2C9F00F0, 0x285E1D47,
0x36194D42, 0x32D850F5, 0x3F9B762C, 0x3B5A6B9B,
0x0315D626, 0x07D4CB91, 0x0A97ED48, 0x0E56F0FF,
0x1011A0FA, 0x14D0BD4D, 0x19939B94, 0x1D528623,
0xF12F560E, 0xF5EE4BB9, 0xF8AD6D60, 0xFC6C70D7,
0xE22B20D2, 0xE6EA3D65, 0xEBA91BBC, 0xEF68060B,
0xD727BBB6, 0xD3E6A601, 0xDEA580D8, 0xDA649D6F,
0xC423CD6A, 0xC0E2D0DD, 0xCDA1F604, 0xC960EBB3,
0xBD3E8D7E, 0xB9FF90C9, 0xB4BCB610, 0xB07DABA7,
```

```
0xAE3AFBA2, 0xAAFBE615, 0xA7B8C0CC, 0xA379DD7B,
0x9B3660C6, 0x9FF77D71, 0x92B45BA8, 0x9675461F,
0x8832161A, 0x8CF30BAD, 0x81B02D74, 0x857130C3,
0x5D8A9099, 0x594B8D2E, 0x5408ABF7, 0x50C9B640,
0x4E8EE645, 0x4A4FFBF2, 0x470CDD2B, 0x43CDC09C,
0x7B827D21, 0x7F436096, 0x7200464F, 0x76C15BF8,
0x68860BFD, 0x6C47164A, 0x61043093, 0x65C52D24,
0x119B4BE9, 0x155A565E, 0x18197087, 0x1CD86D30,
0x029F3D35, 0x065E2082, 0x0B1D065B, 0x0FDC1BEC,
0x3793A651, 0x3352BBE6, 0x3E119D3F, 0x3AD08088,
0x2497D08D, 0x2056CD3A, 0x2D15EBE3, 0x29D4F654,
0xC5A92679, 0xC1683BCE, 0xCC2B1D17, 0xC8EA00A0,
0xD6AD50A5, 0xD26C4D12, 0xDF2F6BCB, 0xDBEE767C,
0xE3A1CBC1, 0xE760D676, 0xEA23F0AF, 0xEEE2ED18,
0xF0A5BD1D, 0xF464A0AA, 0xF9278673, 0xFDE69BC4,
0x89B8FD09, 0x8D79E0BE, 0x803AC667, 0x84FBDBD0,
0x9ABC8BD5, 0x9E7D9662, 0x933EB0BB, 0x97FFAD0C,
0xAFB010B1, 0xAB710D06, 0xA6322BDF, 0xA2F33668,
0xBCB4666D, 0xB8757BDA, 0xB5365D03, 0xB1F740B4
```

B.2 wCcrc32

This table is used for word (4-byte) compression. It is used after a long packet is vector-compressed. Each entry is 4 bytes (32 bits) wide.

```
0x00000000, 0x490D678D, 0x921ACF1A, 0xDB17A897,
0x20F48383, 0x69F9E40E, 0xB2EE4C99, 0xFBE32B14,
0x41E90706, 0x08E4608B, 0xD3F3C81C, 0x9AFEAF91,
0x611D8485, 0x2810E308, 0xF3074B9F, 0xBA0A2C12,
0x83D20E0C, 0xCADF6981, 0x11C8C116, 0x58C5A69B,
0xA3268D8F, 0xEA2BEA02, 0x313C4295, 0x78312518,
0xC23B090A, 0x8B366E87, 0x5021C610, 0x192CA19D,
0xE2CF8A89, 0xABC2ED04, 0x70D54593, 0x39D8221E,
0x036501AF, 0x4A686622, 0x917FCEB5, 0xD872A938,
0x2391822C, 0x6A9CE5A1, 0xB18B4D36, 0xF8862ABB,
0x428C06A9, 0x0B816124, 0xD096C9B3, 0x999BAE3E,
0x6278852A, 0x2B75E2A7, 0xF0624A30, 0xB96F2DBD,
0x80B70FA3, 0xC9BA682E, 0x12ADC0B9, 0x5BA0A734,
```

```
0xA0438C20, 0xE94EEBAD, 0x3259433A, 0x7B5424B7,
0xC15E08A5, 0x88536F28, 0x5344C7BF, 0x1A49A032,
0xE1AA8B26, 0xA8A7ECAB, 0x73B0443C, 0x3ABD23B1,
0x06CA035E, 0x4FC764D3, 0x94D0CC44, 0xDDDDABC9,
0x263E80DD, 0x6F33E750, 0xB4244FC7, 0xFD29284A,
0x47230458, 0x0E2E63D5, 0xD539CB42, 0x9C34ACCF,
0x67D787DB, 0x2EDAE056, 0xF5CD48C1, 0xBCC02F4C,
0x85180D52, 0xCC156ADF, 0x1702C248, 0x5E0FA5C5,
0xA5EC8ED1, 0xECE1E95C, 0x37F641CB, 0x7EFB2646,
0xC4F10A54, 0x8DFC6DD9, 0x56EBC54E, 0x1FE6A2C3,
0xE40589D7, 0xAD08EE5A, 0x761F46CD, 0x3F122140,
0x05AF02F1, 0x4CA2657C, 0x97B5CDEB, 0xDEB8AA66,
0x255B8172, 0x6C56E6FF, 0xB7414E68, 0xFE4C29E5,
0x444605F7, 0x0D4B627A, 0xD65CCAED, 0x9F51AD60,
0x64B28674, 0x2DBFE1F9, 0xF6A8496E, 0xBFA52EE3,
0x867D0CFD, 0xCF706B70, 0x1467C3E7, 0x5D6AA46A,
0xA6898F7E, 0xEF84E8F3, 0x34934064, 0x7D9E27E9,
0xC7940BFB, 0x8E996C76, 0x558EC4E1, 0x1C83A36C,
0xE7608878, 0xAE6DEFF5, 0x757A4762, 0x3C7720EF,
0x0D9406BC, 0x44996131, 0x9F8EC9A6, 0xD683AE2B,
0x2D60853F, 0x646DE2B2, 0xBF7A4A25, 0xF6772DA8,
0x4C7D01BA, 0x05706637, 0xDE67CEA0, 0x976AA92D,
0x6C898239, 0x2584E5B4, 0xFE934D23, 0xB79E2AAE,
0x8E4608B0, 0xC74B6F3D, 0x1C5CC7AA, 0x5551A027,
0xAEB28B33, 0xE7BFECBE, 0x3CA84429, 0x75A523A4,
0xCFAF0FB6, 0x86A2683B, 0x5DB5C0AC, 0x14B8A721,
0xEF5B8C35, 0xA656EBB8, 0x7D41432F, 0x344C24A2,
0x0EF10713, 0x47FC609E, 0x9CEBC809, 0xD5E6AF84,
0x2E058490, 0x6708E31D, 0xBC1F4B8A, 0xF5122C07,
0x4F180015, 0x06156798, 0xDD02CF0F, 0x940FA882,
0x6FEC8396, 0x26E1E41B, 0xFDF64C8C, 0xB4FB2B01,
0x8D23091F, 0xC42E6E92, 0x1F39C605, 0x5634A188,
0xADD78A9C, 0xE4DAED11, 0x3FCD4586, 0x76C0220B,
0xCCCA0E19, 0x85C76994, 0x5ED0C103, 0x17DDA68E,
0xEC3E8D9A, 0xA533EA17, 0x7E244280, 0x3729250D,
```

```
0x0B5E05E2, 0x4253626F, 0x9944CAF8, 0xD049AD75,
0x2BAA8661, 0x62A7E1EC, 0xB9B0497B, 0xF0BD2EF6,
0x4AB702E4, 0x03BA6569, 0xD8ADCDFE, 0x91A0AA73,
0x6A438167, 0x234EE6EA, 0xF8594E7D, 0xB15429F0,
0x888C0BEE, 0xC1816C63, 0x1A96C4F4, 0x539BA379,
0xA878886D, 0xE175EFE0, 0x3A624777, 0x736F20FA,
0xC9650CE8, 0x80686B65, 0x5B7FC3F2, 0x1272A47F,
0xE9918F6B, 0xA09CE8E6, 0x7B8B4071, 0x328627FC,
0x083B044D, 0x413663C0, 0x9A21CB57, 0xD32CACDA,
0x28CF87CE, 0x61C2E043, 0xBAD548D4, 0xF3D82F59,
0x49D2034B, 0x00DF64C6, 0xDBC8CC51, 0x92C5ABDC,
0x692680C8, 0x202BE745, 0xFB3C4FD2, 0xB231285F,
0x8BE90A41, 0xC2E46DCC, 0x19F3C55B, 0x50FEA2D6,
0xAB1D89C2, 0xE210EE4F, 0x390746D8, 0x700A2155,
0xCA000D47, 0x830D6ACA, 0x581AC25D, 0x1117A5D0,
0xEAF48EC4, 0xA3F9E949, 0x78EE41DE, 0x31E32653
```

B.3 vCcrc32

Eight vectors are needed to store the vector compression table to facilitate AltiVec's vperm instruction. There are two logical tables, one for L-nibbles and the other for H-nibbles. The four-byte result of a parallel nibble lookup needs four vectors.

```
vCcrc32 for L-nibbles
byte 0 vector =
0x00, 0x75, 0xEB, 0x9E, 0xD2, 0xA7, 0x39, 0x4C,
0xA0, 0xD5, 0x4B, 0x3E, 0x72, 0x07, 0x99, 0xEC,
byte 1 vector =

0x00, 0xBE, 0x7C, 0xC2, 0x38, 0x86, 0x44, 0xFA,
0xB1, 0x0F, 0xCD, 0x73, 0x89, 0x37, 0xF5, 0x4B,

byte 2 vector =
0x00, 0x46, 0x8D, 0xCB, 0x07, 0x41, 0x8A, 0xCC,
0x13, 0x55, 0x9E, 0xD8, 0x14, 0x52, 0x99, 0xDF,
```

```
byte 3 vector =
0x00, 0xB7, 0x6E, 0xD9, 0x6B, 0xDC, 0x05, 0xB2,
0x61, 0xD6, 0x0F, 0xB8, 0x0A, 0xBD, 0x64, 0xD3

Similarly, vCcrc32 for L-nibbles
byte 0 vector =
0x00, 0x45, 0x8B, 0xCE, 0x12, 0x57, 0x99, 0xDC,
0x24, 0x61, 0xAF, 0xEA, 0x36, 0x73, 0xBD, 0xF8,

byte 1 vector =
0x00, 0xA3, 0x46, 0xE5, 0x4D, 0xEE, 0x0B, 0xA8,
0x9B, 0x38, 0xDD, 0x7E, 0xD6, 0x75, 0x90, 0x33,

byte 0 vector =
0x00, 0x3B, 0x76, 0x4D, 0xF0, 0xCB, 0x86, 0xBD,
0xE0, 0xDB, 0x96, 0xAD, 0x10, 0x2B, 0x66, 0x5D,

byte 0 vector =
0x00, 0x75, 0xEA, 0x9F, 0x63, 0x16, 0x89, 0xFC,
0xC6, 0xB3, 0x2C, 0x59, 0xA5, 0xD0, 0x4F, 0x3A
```

How to Reach Us:

Home Page:
www.freescale.com

email:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
(800) 521-6274
480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku
Tokyo 153-0064, Japan
0120 191014
+81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate,
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor
Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
(800) 441-2447
303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor
@hibbertgroup.com

Document Number: AN2926
Rev. 0
01/2006

freescale™
semiconductor