Application Report
SPRA530

Cyclic Redundancy Check Computation: An Implementation Using the TMS320C54x

Patrick Geremia *C5000*

Abstract

Cyclic redundancy check (CRC) code provides a simple, yet powerful, method for the detection of burst errors during digital data transmission and storage. CRC implementation can use either hardware or software methods. This application report presents different software algorithms and compares them in terms of memory and speed using the Texas Instruments (TI™) TMS320C54x digital signal processor (DSP). Various common CRC codes will be used.

Contents

Figures

Tables

Introduction

Error correction codes provide a means to detect and correct errors introduced by a transmission channel. Two main categories of code exist: block codes and convolutional codes. They both introduce redundancy by adding parity symbols to the message data.

Cyclic redundancy check (CRC) codes are a subset of cyclic codes that are also a subset of linear block codes. The theory behind block coding and more specifically CRC coding is briefly discussed in this application report as well as most common CRC codes.

CRC implementation can use either hardware or software methods. In the traditional hardware implementation, a simple shift register circuit performs the computations by handling the data one bit at a time. In software implementations, handling data as bytes or words becomes more convenient and faster. You choose a particular algorithm depending on which memory and speed constraints are required. Different types of algorithms and the results are presented later in this application report.

Coding Theory Behind CRC

Fundamentals of Block Coding

A block code consists of a set of fixed-length vectors called code words. The length of a code word is the number of elements in the vector and is denoted by *n*. The elements of a code word are selected from an alphabet of *q* elements. When the alphabet consists of two elements, 0 and 1, the code is binary; otherwise, it is nonbinary code. In the binary case, the elements are bits. The following discussion focuses on binary codes.

There are 2^n possible code words in a binary block code of length *n*. From these 2^n code words you may select $M = 2^k$ code words *(k < n)* to form a code. Thus a block of *k* information bits is mapped into a code word of length *n* selected from the set of $M = 2^k$ code words. This is called an (*n*,*k*) code and the ratio *k/n* is defined to be the rate of the code.

The encoding and decoding functions involve the arithmetic operations of addition and multiplication performed on code words. These arithmetic operations are performed according to the conventions of the algebraic field that has, as its elements, the symbols contained in the alphabet. For binary codes, the field is finite and has 2 elements, 0 and 1, and is called *GF(2)* (Galois Field).

A code is linear if the addition of any two-code vectors forms another code word. Code linearity simplifies implementation of coding operations.

The minimum distance d_{min} is the smallest hamming distance between two code words. The hamming distance is the number of symbols (bits in the binary case) in which they differ. The minimum distance is closely linked to the capacity of the code to detect and correct errors and is a function of the code characteristics. An (*n*,*k*) block code is capable of detecting $d_{min} - 1$ errors and correcting $\frac{1}{2}(d_{min} - 1)$ errors.

Suppose m_i is the k-bits information word, the output of the encoder will result in an n-bits code word c_i, defined as $c_i = m_i G \ \ (i = 0,1,....,2^k - 1)$ where *G* is called the generator matrix of dimension $k \times n$. Every linear block code is equivalent to a systematic code; therefore, it is always possible to find a matrix G generating code words formed by the *k* information bits followed by the *n-k* parity check bits, for example, $G = [I_k, P]$ where I_k is the $k \times k$ identity matrix and *P* is a $k \times (n-k)$ matrix of parity checks. The parity check matrix *H* is defined as a $(n-k) \times k$ matrix so that $GH^T = 0$. If the code is systematic, then $H = \left\lfloor P^T, I_{n-k} \right\rfloor$. Since all code words are linear sums of the rows in *G*, we have $c_i H^T = 0$ for all *I,* $(i = 0,1,....,2^k - 1)$. If a code word *c* is corrupted during transmission so that the receive word is $c' = c + e$, where *e* is a non-zero error pattern, then we call syndrome *s* the result of following multiplication, $s = c' H^T = (c+e)H^T = cH^T + eH^T = eH^T$, where *s* is $(n-k)$ - dimensional vector. The syndrome *s* is dependent on the error pattern. If the error pattern is a code vector, the errors go undetected. For all other error patterns, however, the syndrome is nonzero. Decoding then uses standard array decoders that are based on lookup tables to associate each syndrome with an error pattern. This method becomes impractical for many interesting and powerful codes as $(n-k)$ increases.

Cyclic codes are a subclass of linear block codes with an algebraic structure that enables encoding to be implemented with a linear feedback shift register and decoding to be implemented without using standard array decoders. Therefore, most block codes in use today are cyclic or are closely related to cyclic codes. These codes are best described if vectors are interpreted as polynomials. In a cyclic code, all code word polynomials are multiples of a generator polynomial $g(x)$ of degree $n-k$. This polynomial is chosen to be a divisor of $x^n + 1$ so that a cyclic shift of a code vector yields another code vector. A message polynomial $m_i(x)$ can be mapped to a code word polynomial $c_i(x) = m_i(x)x^{n-k} - r_i(x) \ \ (i = 0,1,....,2^k - 1)$ (systematic form), where $r_i(x)$ is the remainder of the division of $m_i(x)x^{n-k}$ by $g(x)$.

The first step in decoding is to determine if the receive word is a multiple of $g(x)$. This is done by dividing it by $g(x)$ and examining the remainder. Since polynomial division is a linear operation, the resulting syndrome $g(x)$ depends only on the error pattern. If $g(x)$ is the all-zero polynomial, transmission is errorless or an undetectable error has occurred. If $g(x)$ is nonzero, at least one error has occurred. This is the principle of CRC. More powerful codes attempt to correct the error and use the syndrome to determine the locations and values of multiple errors.

CRC Coding

CRC codes are a subset of cyclic codes and use a binary alphabet, 0 and 1. Arithmetic is based on *GF(2)*, for example, modulo-2 addition (logical XOR) and modulo-2 multiplication (logical AND).

In a typical coding scheme, systematic codes are used. Assume the convention that the leftmost bit represents the highest degree in the polynomial. Suppose $m(x)$ to be the message polynomial, $c(x)$ the code word polynomial and $g(x)$ the generator polynomial, we have $c(x) = m(x)g(x)$ which can also be written using the systematic form $c(x) = m(x)x^{n-k} + r(x)$, where $r(x)$ is the remainder of the division of $m(x)x^{n-k}$ by $g(x)$ and $r(x)$ represents the CRC bits. The transmitted message $c(x)$ contains k-information bits followed by $n-k$ CRC bits, for example, $c(x) = m_{k-1}x^{n-1} + ... + m_0 x^{n-k} + r_{n-k-1}x^{n-k-1} + r_0$. So encoding is straightforward: multiply $m(x)$ by x^{n-k}, that is, append $n-k$ bits to the message, calculate the CRC bits by dividing $m(x)x^{n-k}$ by $g(x)$, and append the resulting $n-k$ CRC bits to the message. For the decoding part, the same algorithm can be used. If $c'(x)$ is the received message, then no error or undetectable errors have occurred if $c'(x)$ is a multiple of $g(x)$, which is equivalent to determining that if $c'(x)x^{n-k}$ is a multiple of $g(x)$, that is, if the remainder of the division from $c'(x)x^{n-k}$ by $g(x)$ is 0.

CRC Code Examples

The performance of a CRC code is dependent on its generator polynomial. The theory behind its generation and selection is beyond the scope of this application report. This application report will only consider the most common used ones (see Table 1).

Table 1. Common CRC Codes and Associated Generator Polynomial

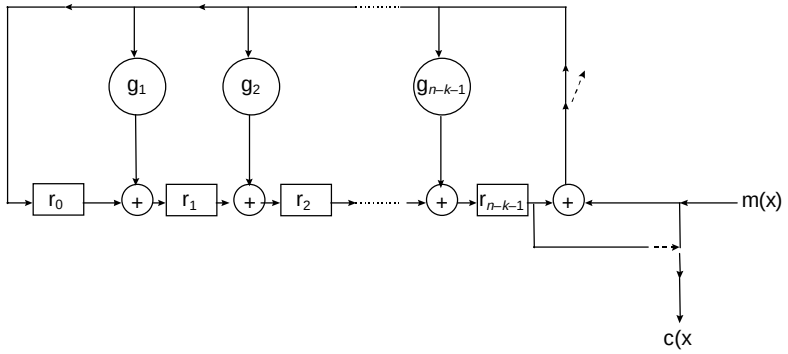
CRC Code	Generator Polynomial
CRC-CCITT (X25)	$x^{16} + x^{12} + x^5 + 1$
CRC-32 (Ethernet)	$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$
GSM TCH/FS-HS-EFS (Channel coding for speech traffic channels)	$x^3 + x + 1$
GSM TCH/EFS pre-coding (Preliminary channel coding for Enhanced Full Rate)	$x^8 + x^4 + x^3 + x^2 + 1$
GSM control channels – FIRE code (Channel coding for control channels)	$x^{40} + x^{26} + x^{23} + x^{17} + x^3 + 1$

Algorithms for CRC Computation

Bitwise Algorithm

The bitwise algorithm (CRCB) is simply a software implementation of what would be done in hardware using a linear feedback shift register (LFSR). Figure 1 illustrates a generic hardware implementation. The shift register is driven by a clock. At every clock pulse, the input data is shifted into the register in addition to transmitting the data. When all input bits have been processed, the shift register contains the CRC bits, which are then shifted out on the data line.

Figure 1. CRC Generation Using a Linear Feedback Shift Register (LFSR)



In the software implementation, the following algorithm can be used:

Assume now that the check bits are stored in a register referred as the CRC register, a software implementation would be:

1) CRC ← 0
2) if the CRC left-most bit is equal to 1, shift in the next message bit, and XOR the CRC register with the generator polynomial; otherwise, only shift in the next message bit
3) Repeat step 2 until all bits of the augmented message have been shifted in

Faster implementations can be achieved by handling the data as larger units than bits, as long as the size does not exceed the degree of the generator polynomial. However, the speed gain corresponds to a memory increase, since precomputed values (lookup tables) will be used.

Lookup Table Algorithms

A code word can be written as $c(x) = x^{n-k}m(x) + r(x) = a(x)g(x) + r(x)$, where $r(x)$ is the CRC polynomial of the input message $m(x)$. Let us augment the message by α bits. We therefore have $m'(x) = x^{\alpha}c(x) + b(x)$, where $b(x) = b_{\alpha-1}x^{a-1} + ... + b_0$ is the new added α -bit word. The CRC of the augmented message is the remainder of $x^{n-k}m'(x)$ divided by $g(x)$, let us call it $r'(x)$. We have $r'(x) = R_{g(x)}\left[x^{n-k}m'(x)\right]$. Expanding the dividend, we obtain:

$x^{n-k}m'(x) = x^{n-k}\left[x^{\alpha}c(x) + b(x)\right] = x^{n-k}b(x) + x^{\alpha}a(x)g(x) + x^{\alpha}r(x)$, so that,
$r'(x) = R_{g(x)}\left[x^{n-k}b(x) + x^{\alpha}r(x)\right]$. Expanding the result, we have
$r'(x) = R_{g(x)}\left[(b_{\alpha-1} + r_{n-k-1})x^{n-k-1+\alpha} + ... + (b_0 + r_{n-k-\alpha})x^{n-k}\right] + r_{n-k-\alpha-1}x^{n-k-1} + \cdots + r_0x^{\alpha}$

The last equation relates the check bits of the augmented message with the check bits of the original message. Note that if $n-k=\alpha$, then
$r'(x) = R_{g(x)}\left[(b_{\alpha-1} + r_{n-k-1})x^{n-k-1+\alpha} + ... + (b_0 + r_{n-k-\alpha})x^{n-k}\right]$. Different algorithms for CRC computation may be viewed as methods to compute $r'(x)$ from $r(x)$ and $b(x)$. Practical values for α are 8 or 16, since it corresponds to many machine word lengths.

Standard Lookup Table Algorithm

The idea behind the standard lookup table algorithm is to precompute the CRC values of all augmented bits combinations. Therefore, 2^{α} $(n-k)$ -bit word values are necessary, which limits practical implementations to small α values.

Assume now that the check bits are stored in a register named CRC, the algorithm will be (if $\alpha < n-k$):

1) CRC ← 0, that is, set the $(r_{n-k-1},...,r_0)$ bits to 0
2) XOR the α input bits with the CRC register content shifted right by $n-k-\alpha$ bits, that is, XOR with the $(r_{n-k-1},...,r_{n-k-\alpha})$ bits
3) find the corresponding value in the lookup table and XOR the CRC register content shifted left by α bits, that is, XOR with the $(r_{\alpha-1},...,r_0)$ bits. This is the new CRC value.
4) Repeat steps 2 to 3 until you reach the end of the message.

In the case where $\alpha = n-k$, steps 2) and 3) will be slightly different:

2) XOR the α input bits with the CRC register, that is, XOR with the $(r_{n-k-1},...,r_0)$ bits
3) Find the corresponding value in the lookup table. This is the new CRC value.

Reduced Lookup Table Algorithm

This algorithm is a variant of the standard lookup table algorithm to fit applications where memory space is of primary importance. The amount of required memory to store the lookup table is significantly reduced and equal to α $(n-k)$ -bit words. The basic idea is to split the expression $R_{g(x)}\left[(b_{\alpha-1}+r_{n-k-1})x^{n-k-1+\alpha}+...+(b_0+r_{n-k-\alpha})x^{n-k}\right]$ into the sum $R_{g(x)}\left[(b_{\alpha-1}+r_{n-k-1})x^{n-k-1+\alpha}\right]+...+R_{g(x)}\left[(b_0+r_{n-k-\alpha})x^{n-k}\right]$. Each term of the sum can be either 0 or 1. If $(b_{\alpha-1-i}+r_{n-k-1-i})$ is 0, then $R_{g(x)}\left[(b_{\alpha-1-i}+r_{n-k-1-i})x^{n-k-1+\alpha-i}\right]$ is equal to 0. So you only need to precompute α values corresponding to $R_{g(x)}\left[x^{n-k-1+\alpha-i}\right]$ with $i=0,...,(\alpha-1)$. The algorithm will be (if $\alpha<n-k$):

1) CRC ← 0, that is, set the $(r_{n-k-1},...,r_0)$ bits to 0
2) XOR the α input bits with the CRC register content shifted right by α bits, XOR with the $(r_{n-k-1},...,r_{n-k-\alpha})$ bits
3) on each bit of this value (α bits), if it is equal to 1 then find the corresponding value in the lookup table and XOR the CRC register content with it
4) XOR this value with the CRC shifted left by α bits, that is, XOR with the $(r_{\alpha-1},...,r_0)$ bits. This is the new CRC value.
5) Repeat steps 2 to 4 until you reach the end of the message.

In the case where $\alpha=n-k$, steps 2) and 4) will be slightly different:

2) XOR the α input bits with the CRC e.g. XOR with the $(r_{n-k-1},...,r_0)$ bits,
4) The value computed in step 3) is the new CRC value

Note that only step 3 differs from the standard lookup table algorithm. As expected, there is an increase of processing due to the fact that each bit must be tested and if it is equal to 1 then a XOR operation must be performed.

TMS320C54x Implementation

General Considerations

The TMS320C54x DSP is a 16-bit fixed-point DSP. Interesting for the implementation of the examples given:

- 40-bit Arithmetical and Logical Unit (ALU)
- Two 40-bit accumulators (A and B)
- Efficient memory addressing modes
- Multiple bus structure
- Barrel shifter

CRC codes with a size smaller than 40 bits are relatively easy to implement. Depending on the type of algorithm used, 8-, 16-, or up to 32-bit data is used.

Results

Tables 2 to 6 illustrate the results obtained from simulation done on 256 input words, which were randomly generated by a C-program using the *rand()* function. *CRCB*, *CRCT*, and *CRCR* are the function names corresponding, respectively, to the bit-wise algorithm, standard lookup table algorithm, and reduced lookup table algorithm. You can refer to the assembly code in the appendixes.

As expected from the theory, we have a trade-off between memory requirement and processing speed between the different algorithms. One trade-off is that the reduced look-up table algorithm does not offer any advantage over the two others; this is understandable, since in both cases, testing of all input bits must be performed.

CRC-CCITT

The standard lookup table algorithm is *2.7 times faster* than the bit-wise implementation, but requires *16.5 times more memory*, due to the lookup table essentially (see Table 2).

Table 2. Benchmarks for CRC-CCITT

Algorithm	Cycles	Program†	Data†	Tables†
CRCB	91‡ (25343§)	10 (17§)	–	–
CRCT	12¶ (9472§)	10 (23§)	1	256
CRCR	91‡ (25602§)	15 (26§)	–	16

† Number of 16-bit words.
‡ To process one word. Includes return instruction.
§ To process array of 256 words. Includes function calls, returns, main loop, and specific initialization.
¶ To process one byte. Includes return instruction.

CRC-32

The standard lookup table algorithm is *3.8 times faster* than the bit-wise implementation, but requires *10.7 times more memory*, due to the lookup table essentially (see Table 3).

Table 3. Benchmarks for CRC-32

Algorithm	Cycles	Program†	Data†	Tables†
CRCB	139‡ (37693§)	12 (34§)	16	–
CRCT	13¶ (9984§)	11 (24§)	–	512
CRCR	16‡ (42750§)	22 (28§)	2	32

† Number of 16-bit words
‡ To process one word. Includes return instruction.
§ To process array of 256 words. Includes function calls, returns, main loop, and specific initialization.
¶ To process one byte. Includes return instruction.

CRC for GSM/TCH

Only one algorithm has been implemented (see Table 4) due to the small CRC size (3 bits). The lookup table oriented algorithms would have required to work on 3-bit entities that are not really supported efficiently by the TMS320C54x architecture.

Table 4. Benchmarks for GSM/TCH

Algorithm	Cycles	Program†	Data†	Tables†
CRCB	89‡ (24859§)	10 (21§)	–	–
CRCT¶	–	–	–	–
CRCR¶	–	–	–	–

† Number of 16-bit words
‡ To process one word. Includes return instruction.
§ To process array of 256 words. Includes function calls, returns, main loop, and specific initialization.
¶ To process one byte. Includes return instruction.

CRC for GSM/TCH/EFS Precoder

The standard lookup table algorithm is *3 times faster* than the bit-wise implementation, but requires *16.2 times more memory*, due to the lookup table essentially (see Table 5).

Table 5. Benchmarks for GSM/TCH/EFS Precoder

Algorithm	Cycles	Program†	Data†	Tables†
CRCB	91‡ (25343§)	10 (17§)	–	–
CRCT	8¶ (8448§)	6 (19§)	1	256
CRCR	53¶ (31486§)	14 (25§)	–	8

† Number of 16-bit words
‡ To process one word. Includes return instruction.
§ To process array of 256 words. Includes function calls, returns, main loop, and specific initialization.
¶ To process one byte. Includes return instruction.

GSM FIRE Code

The reduced lookup table algorithm is not implemented, because it would not have had any advantages over the bit-wise algorithm. This is even more obvious than for the previous codes, since the FIRE code is 40 bits long and fetching 40 bits from memory requires extra overhead.

The standard lookup table algorithm *is 2.7 times faster* than the bit-wise implementation, but requires *14.3 times more memory*, due to the lookup table essentially (see Table 6).

Table 6. Benchmarks for GSM FIRE Code

Algorithm	Cycles	Program†	Data†	Tables†
CRCB	137‡ (37258§)	12 (40§)	–	–
CRCT	20¶ (13570§)	18 (33§)	–	768
CRCR¶	–	–	–	–

† Number of 16-bit words
‡ To process one word. Includes return instruction.
§ To process array of 256 words. Includes function calls, returns, main loop, and specific initialization.
¶ To process one byte. Includes return instruction.

Summary

Three CRC computation algorithms and their implementation using the TMS320C54x DSP have been considered in this application report, as well as various commonly used CRC codes.

The bit-wise algorithm and the standard lookup table algorithm appear to be the most appropriate depending on the application requirements, that is, memory usage or computation time optimization. The results of the simulation based on five CRC codes show that the standard lookup table algorithm is about 3 times faster, but requires about 14 times more memory than the bit-wise algorithms.

The code provided in this application report is easily adaptable to other CRC codes and should provide you a good startup point, when considering implementation of cyclic redundancy check.

Code Availability

The associated program files are available from Texas Instruments TMS320 Bulletin Board System (BBS). Internet users can access the BBS via anonymous ftp.

References

1. Ramabadran T.V., Gaitonde S.S., “A tutorial on CRC computations”, IEEE Micro, Aug 1988
2. ETSI GSM 05.03 Specification, “Channel Coding” , August 1996, Version 5.2.0.
3. John G. Proakis, “Digital Communications”, 3rd edition.

Appendix A CRC-CCITT Listing†

```
*************************************************************
*
* (C) Copyright 1997, Texas Instruments Incorporated
*
*************************************************************

        .mmregs
        .def Entry, Reset
        .title  "CRC CCITT"

genCRC  .set    1021h   ; Generator CRC Polynomial value
                        ; g(x) = x16 + x12 + x5 + 1
inport  .set    0h      ; IO address

        .asg    NOP,PIPELINE

*************************************************************

*       Stack setup
*       Reset vector

*************************************************************
BOS     .usect  "stack",0fh     ; setup stack
TOS     .usect  "stack",1       ; Top of stack at reset

        .sect   "vectors"
Reset:
        bd      Entry           ; reset vector
        stm     #TOS,SP         ; Setup stack


*************************************************************

*       Main Program

*************************************************************
        .sect   "program"
Entry
        .include "c5xx.inc"

        ld      #crc,DP         ; scratch pad mem -> DP=0

        portr   #inport,@nbDataIn       ; nb words
        addm    #-1,@nbDataIn           ; for repeat
```

† Part of the tables are truncated.

```
*************************************************************
*       CRCB (word wise)

*************************************************************
        stm     #input,AR2      ; copy inputwords in RAM
        rpt     @nbDataIn       ;
         portr  #inport,*AR2+   ; read them from IO space

        ld      #genCRC,16,B    ; BH = CRC gen. poly.
        mvdm    @nbDataIn,AR1   ; AR1 = length of message
        stm     #input,AR2      ; AR2 points to input word

        ld      *AR2+,16,A      ; initialize CRC register
next    calld    CRCB           ; perform CRC computation
        or      *AR2+,A
        nop
        banz    next,*AR1-      ; process all input words

        sth     A,@crc          ; store result in memory

*************************************************************

*       CRCT (byte wise)

*************************************************************

        stm     #input,AR2      ; AR2 points to inputword
        mvdm    @nbDataIn,AR1   ; AR1 = length of message
        ld      #0,A            ; clear Acc A
        rsbx    SXM             ; no sign extension

        stm     #t_start,AR3    ; AR3 = LTU start address
next1   calld   CRCT            ; process LSByte
         ld     *AR2,-8,B       ; BL = LSByte
        ld      *AR2+,B
        calld   CRCT            ; process MSByte
         and    #0FFh,B         ; BL = MSByte
        banz    next1,*AR1-

        stl     A,@crc          ; store result


*************************************************************

*       CRCR (word wise)

*************************************************************

        stm     #input,AR2      ; AR2 points to inputword
        mvdm    @nbDataIn,AR1   ; AR1 = length of message
        ld      #0,A            ; clear Acc A

        stm     #rtw_start-1,AR3 ; AR3=LTU start address-1
next2   xor     *AR2+,A          ; AL = CRC  XOR inputWord
        calld    CRCRW           ; process input word
         and     #0FFFFh,A
        banzd    next2,*AR1-
```

```
          stm     #rtw_start-1,AR3 ; AR3=LTU start address-1

         stl     A,@crc            ; store result

done     b       done


*************************************************************

* CRCB routine : bit-wise CRC calculation

* input : 1) 16-bit input words in AL
*         2) CRC generator polynomial in BH
*         3) OVM = 0
* ouput : CRC value in AH

* BRC, REA, RSA, A, C, SP modified

* Code size = 10 words
* Cycles    = 11 + 5*N = 91 for N = 16 bits

* A = |A31.............A16 A15.............A0|
*      <------ CRC ------> <---- input bits---->

* B = |B31.............B16 B15.............B0|
*      <-- polynomial --->

*************************************************************
CRCB     stm             #16-1,BRC       ; BRC = 16-1
         rptb            CRC_end-1       ; repeat block
           sftl          A,1,A           ; A = A << 1, C=MSB
           PIPELINE
           PIPELINE
           xc            1,C             ;
           xor           B,A             ; if C=1,
                                         ; AH = new CRC
                                         ;    = AH XOR gen.
CRC_end ret                              ; end of repeat



*************************************************************

* CRCT routine : standard lookup table CRC calculation
*                with bytes as input

* input : 1) 8-bit words in AL
*         2) AR3 = LTU start address
*         3) DP = 0 = DP(temp)
*         4) SXM = 0
* ouput : CRC value in AL

* A, B, AR4, SP modified

* code size =  10 words
* cycles = 12 cycles
```

```
* A = |A31.............A16 A15.................A0|
*                          <------- CRC ------->

* B = |B31.......................B7........B0|
*                               <-- input-->


************************************************************
CRCT                    ; AL (low part of Acc A) = CRC bits
        stl     A,-8,@AR4       ; AR4 = CRC >> 8
        stl     A,8,@temp       ; temp = CRC << 8
        xor     @AR4,B  ; B = (inputByte) XOR (CRC>>8)
                        ;   = Offset in LTU
        adds    @AR3,B  ; B = Offset + LTU start address
        stlm    B,AR4   ; AR4 = absolute address in LTU
        retd
         ld      @temp,A        ; AL = CRC << 8
         xor     *AR4,A         ; AL = new CRC




************************************************************

* CRCRW  routine : reduced lookup table CRC calculation
*                  with words as input

* input : 1) 16-bit words in AL
*         2) AR3 = LTU start address - 1
* ouput : CRC value in AL

* A, B, C, AR3, BRC, RSA, REA, SP modified

* Code size = 15 words
* Cycles    = 11 + 5*N = 91 for N = 16 bits

* A = |A31.............A16 A15.................A0|
*                          <------- CRC ------->

************************************************************
CRCRW
        stm     #16-1,BRC       ; BRC = 16-1
        rptbd   #loop-1         ; repeat block
         ld     #0,B            ; reset B
         nop
          ror   A               ; get MSBit in Carry
          mar   *AR3+           ; increment index in LTU
          nop
          xc    1,C             ; test Carry
          xor   *AR3,B  ; if 1 then B = B XOR table[i++]
loop    retd                    ; end of repeat
        xor     B,A             ; AL = new CRC
        nop


        .bss    input,257       ; augmented message
                                ; (16 bits appended)
```

```
crc     .usect  "scratchPad",32 ;scratch pad memory
temp    .set    crc+1
nbDataIn .set   temp+1


        .sect   "reducedTablew" ; reduced LUT
                                ; (based on words)
rtw_start
        .word   01021h
        .word   02042h
        .word   04084h
        .word   08108h
        .word   01231h
        .word   02462h
        .word   048C4h
        .word   09188h
        .word   03331h
        .word   06662h
        .word   0CCC4h
        .word   089A9h
        .word   0373h
        .word   06E6h
        .word   0DCCh
        .word   01B98h
rtw_len .set    16

        .sect   "table" ; LTU (based on bytes)
t_start
        .word   00h
        .word   01021h
        .word   02042h
        .word   03063h
……
        .word   0ED1h
        .word   01EF0h
t_len   .set    256
```

Appendix B CRC-32 Listing†

```
*************************************************************
*
* (C) Copyright 1997, Texas Instruments Incorporated
*
*************************************************************

        .mmregs
        .global Entry, Reset
        .title  "CRC 32"

hgenCRC .set    004C1h  ; CRC Gen. Poly. MSB's
lgenCRC .set    01DB7h  ; CRC Gen. Poly. LSB's
      ; g(x) = x32 + x26 + x23 + x22 +x16 +x12 + x11 +
      ;        x10 + x8 + x7 + x5 + x4 + x2 + x + 1

inport  .set    0h                      ; IO address

        .asg    NOP,PIPELINE


*************************************************************
*       Stack setup
*       Reset vector

*************************************************************
BOS     .usect  "stack",0fh        ; setup stack
TOS     .usect  "stack",1          ; Top of stack at reset

        .sect   "vectors"
Reset:
        bd      Entry                   ; reset vector
        stm     #TOS,SP                 ; Setup stack

*************************************************************

*       Main Program

*************************************************************
        .sect   "program"
Entry
        .include "c5xx.inc"

        ld      #crc,DP         ;scratch pad mem -> DP=0

        portr   #inport,@nbDataIn    ; reads in nb words
        addm    #-1,@nbDataIn        ; for repeat


        stm     #input,AR2      ; copy inputwords in RAM
        rpt     @nbDataIn
         portr  #inport,*AR2+   ; read them from IO space
```

† Part of the tables are truncated.

```
*************************************************************

*       CRCB (word wise)

*************************************************************
        ld      #hgenCRC,16,B   ; BH = CRC gen. poly. MSB's
        or      #lgenCRC,B      ; BL = CRC gen. poly. LSB's
        mvdm    @nbDataIn,AR1   ; AR1 = length of message
        stm     #input,AR2      ; AR2 points to input word


        stm     #16,BK          ; circular buffer size = 16
        ld      #0,A            ; init Accumulator A
        stm     #16-1,BRC       ; BRC = 16-1
        rptbd   initbitpos-1    ; repeat block
         stm    #bitpos,AR3     ; AR3 -> bitpos
          stl   A,*AR3+%        ; initialize bit positions
          add   #1,A
initbitpos                      ; end of repeat
        mar     *AR3+%          ; Begin with bit #0
        stm     #0,T            ; init T register
        dld     *AR2+,A         ; CRC register intialized
next    call    CRCB            ; perform CRC computation
        banzd   next,*AR1-      ; process all input words
        mar     *AR2+
        nop
        dst     A,@crc          ; store result in memory

*************************************************************

*       CRCT (byte wise)

*************************************************************
        stm     #input,AR2      ; AR2 points to inputword
        mvdm    @nbDataIn,AR1   ; AR1 = length of message
        ld      #0,A            ; clear Acc A
        rsbx    SXM             ; no sign extend

        stm     #t_start,AR3    ; AR3 = LTU start address
next1   calld   CRCT            ; process LSByte
         ld     *AR2,-8,B       ; BL = LSByte
        ld      *AR2+,B
        calld   CRCT            ; process MSByte
         and    #0FFh,B         ; BL = MSByte
        banz    next1,*AR1-

        dst     A,@crc          ; store result


*************************************************************

*       CRCR (word wise)

*************************************************************
        stm     #2,AR0          ; index = 2
        stm     #input,AR2      ; AR2 points to inputword
        mvdm    @nbDataIn,AR1   ; AR1 = length of message
        ld      #0,A            ; clear Acc A
```

```
next2   calld   CRCRW           ;process input word
         ld     *AR2+,B
         nop
        banz    next2,*AR1-

        dst     A,@crc          ;store result



done    b       done


************************************************************

* CRCB routine : bit-wise CRC calculation

* input : 1) AR2 points to 16-bit word input value
*         2) CRC generator polynomial in B (32 bits)
*         3) OVM = 0
* ouput : CRC value in A (32 bits)

* BRC, REA, RSA, A, C, T, TC, AR3, SP modified

* Code size = 12 words
* Cycles    = 11 + 8*N = 139 for N = 16 bits

* A = |A31.............A16 A15.................A0|
*      <---------------- CRC ------------------>

* B = |B31.............B16 B15.................B0|
*      <--------------polynomial -------------->

************************************************************
CRCB    stm     #16-1,BRC       ; BRC = 16 - 1
        rptb    CRCB_end-1      ; repeat block
         bitt   *AR2            ; Test bit #(15-T)
         sftl   A,1,A           ; A = A << 1
         ld     *AR3+%,T        ; T = next bit position
         xc     1,TC
         or     #1,A            ; if TC=1, A(LSB) = 1
         xc     1,C
         xor    B,A             ; if C=1, A = new CRC
CRCB_end
        ret


************************************************************

* CRCT routine : standard lookup table CRC calculation
*                with bytes as input

* input : 1) 8-bit words in B
*         2) AR3 = LTU start address
*         3) DP = 0
*         4) SXM = 0
* ouput : CRC value in A (32 bits)
```

```
* A, B, AR4, SP modified

* code size =  11 words
* cycles = 13 cycles

* A = |A31.............A16 A15.................A0|
*      <---------------- CRC ------------------>

* B = |B31.............B16..........B7........B0|
*                                   <---input-->

************************************************************
CRCT         ; AL (low part of Acc A) = low part of CRC
             ; AG (high part of Acc A) = high part of CRC
        sth     A,-8,@AR4 ; AR4 = CRC >> 24
        xor     @AR4,B    ; B = (inputByte) XOR (CRC>>24)
                          ;   = Offset in LTU
        sftl    B,1,B     ; multiply by 2 because 32 bits
        add     @AR3,B    ; B = Offset + LTU start address
        stlm    B,AR4     ; AR4 = absolute address in LTU
        sftl    A,8,A     ; A = A << 8
        retd
        dld     *AR4,B    ; B = LTU[inputbyte XOR (CRC>>24)]
        xor     B,A       ; A = new CRC


************************************************************

* CRCRW  routine : reduced lookup table CRC calculation
*                  with words as input

* input : 1) input word in B
*         2) DP = DP(temp) = DP(temp1)
*         3) SXM = 0
* ouput : CRC value in A (32 bits)

* A, B, C, AR3, BRC, RSA, REA, SP modified

* Code size = 22 words
* Cycles    = 16 + 11*N = 192 for N = 16 bits (worst case)
*           = 16 +  7*N = 128 for N = 16 bits (best case)
*           = 160 average

* A = |A31.............A16 A15.................A0|
*      <---------------- CRC ------------------>

************************************************************
CRCRW
        stm     #16-1,BRC
        sth     A,@temp1        ; temp1 = CRC >> 16
        stl     A,@temp         ; temp = A = CRC << 16
        ld      @temp1,A        ; A = CRC >> 16
        xor     B,A         ; A = (CRC >> 16) XOR inputByte
        ld      #0,B            ; clear Acc B
        rptbd   #loop-1
        stm     #rtw_start,AR3  ; AR3 = LTU start address
         ror    A               ; get A(LSB) in Carry bit
```

```
          bc     noCarry,NC
           dst    A,@temp1
           dld    *AR3,A  ;if C=1 then B = B XOR table[i++]
           xor    A,B
           dld    @temp1,A
noCarry   mar    *AR3+0

loop     retd
         ld      @temp,16,A      ; A = temp = CRC << 16
         xor     B,A             ; new CRC in A




         .bss    input,258 ; augmented message
                           ; (32 bits appended)
bitpos .usect "circ",16   ; contain bit number

         .def    crc
crc      .usect  "scratchPad",32 ; scratch pad memory
temp     .set    crc+2
temp1    .set    temp+2
nbDataIn .set    temp1+2

         .sect   "table"         ; LTU (byte wise)
t_start
         .long   00h
         .long   04C11DB7h
         .long   09823B6Eh
         .long   0D4326D9h
……
         .long   0B8757BDAh
         .long   0B5365D03h
         .long   0B1F740B4h
t_len    .set    512


         .sect   "reducedTablew" ; reduced LTU (word wise)
rtw_start
         .long   04C11DB7h
         .long   09823B6Eh
         .long   0130476DCh
         .long   02608EDB8h
         .long   04C11DB70h
         .long   09823B6E0h
         .long   034867077h
         .long   0690CE0EEh
         .long   0D219C1DCh
         .long   0A0F29E0Fh
         .long   0452421A9h
         .long   08A484352h
         .long   010519B13h
         .long   020A33626h
         .long   041466C4Ch
         .long   0828CD898h
rtw_len  .set    32
```

Appendix C GSM/TCH Listing

```
*************************************************************
*
* (C) Copyright 1997, Texas Instruments Incorporated
*
*************************************************************

        .mmregs
        .global Entry, Reset
        .title  "CRC - GSM/all TCH"

genCRC  .set    6000h   ; Generator CRC Polynomial value
                        ; (left aligned x3 + x + 1)
inport  .set    0h
        .asg    NOP,PIPELINE

*************************************************************

*       Stack setup
*       Reset vector

*************************************************************
BOS     .usect  "stack",0fh          ; setup stack
TOS     .usect  "stack",1            ; Top of stack at reset

        .sect   "vectors"
Reset:
        bd      Entry                ; reset vector
        stm     #TOS,SP              ; Setup stack

*************************************************************

*       Main Program

*************************************************************
        .sect   "program"
Entry
        .include "c5xx.inc"

        ld      #crc,DP         ;scratch pad mem -> DP=0

        portr   #inport,@nbDataIn    ;reads in nb words
        addm    #-1,@nbDataIn    ;for repeat


*************************************************************

*       CRCB (word wise)

*************************************************************
        stm     #input,AR2      ; copy inputwords in RAM
        rpt     @nbDataIn       ;
         portr  #inport,*AR2+   ; read them from IO space

        ld      #genCRC,16,B    ; BH = CRC gen. poly.
```

```
        mvdm    @nbDataIn,AR1   ; AR1 = length of message
        stm     #input,AR2      ; AR2 points to input word

        ld      *AR2+,16,A      ; initialize CRC register
next    calld   CRCB            ; perform CRC computation
         stm     #16-1,BRC      ; BRC = 16 - 1
        banz    next,*AR1-      ; process all input words
        calld   CRCB
         stm    #3-1,BRC        ; trailing bits

        sth     A,@crc          ; store result in memory



done    b       done


*************************************************************

* CRCB routine : bit-wise CRC calculation

* input : 1) 16-bit words in AL
*         2) CRC generator polynomial in BH (left justified)
*         3) OVM = 1
*         4) number of bits to test in BRC
* ouput : CRC value in AH (left justified)

* BRC, REA, RSA, AR2, A, C, SP modified

* Code size = 10 words
* Cycles    = 9 + 5*N = 89 for N = 16

* A = |A31..A29........A16 A15.................A0|
*      <-CRC-->            <---- input bits---->

* B = |B31..B29.............................B0|
*      <-poly->

*************************************************************
CRCB    rptbd   CRC_end-1       ; repeat block
         or      *AR2+,A        ; get next input word
         nop
          sftl    A,1,A         ; A = A << 1
          PIPELINE
          PIPELINE
          xc      1,C           ; test Carry bit
          xor     B,A           ; if C=1 then A = A XOR B
                                ;               = new CRC
CRC_end ret



        .bss    input,257       ; augmented message
                                ; (3 bits appended)

crc     .usect  "scratchPad",32 ; scratch pad memory
nbDataIn .set    crc+1
```

Appendix D GSM/TCH/EFS Precoder Listing†

```
*************************************************************
*
* (C) Copyright 1997, Texas Instruments Incorporated
*
*************************************************************

        .mmregs
        .global Entry, Reset
        .title  "CRC - GSM/EFR pre coding"

genCRC  .set    1D00h   ; Generator CRC Polynomial value
                        ;(left aligned)
                        ; g(x) = x8 + x4 + x3 + x2 + 1

inport  .set    0h      ; IO space address

        .asg    NOP,PIPELINE

*************************************************************

*       Stack setup
*       Reset vector

*************************************************************
BOS     .usect  "stack",0fh        ; setup stack
TOS     .usect  "stack",1          ; Top of stack at reset

        .sect   "vectors"
Reset:
        bd      Entry                   ; reset vector
        stm     #TOS,SP                 ; Setup stack

*************************************************************

*       Main Program

*************************************************************
        .sect   "program"
Entry
        .include "c5xx.inc"

        ld      #crc,DP         ; scratch pad mem -> DP=0

        portr   #inport,@nbDataIn       ; reads in nb words
        addm    #-1,@nbDataIn           ; for repeat


        stm     #input,AR2      ; copy inputwords in RAM
        rpt     @nbDataIn       ;
         portr  #inport,*AR2+   ; read them from IO space
```

† Part of the tables are truncated.

```
*************************************************************

*       CRCB (word wise)

*************************************************************
        ld      #genCRC,16,B    ; BH = CRC gen. poly.
        mvdm    @nbDataIn,AR1   ; AR1 = length of message.
        stm     #input,AR2      ; AR2 points to input word

        ld      *AR2+,16,A      ; initialize CRC register
next    calld   CRCB            ; perform CRC computation.
         or      *AR2+,A        ; store input word in AL
         nop
        banz    next,*AR1-      ; process all input words

        sth     A,-8,@crc       ; store result in memory


*************************************************************

*       CRCT (byte wise)

*************************************************************
        stm     #input,AR2      ; AR2 points to inputword
        mvdm    @nbDataIn,AR1   ; AR1 = length of message
        ld      #0,A            ; clear Acc A
        ld      #genCRC,B       ; init Acc B with gen. poly.
        sftl    B,-8,B          ; right justify it
        rsbx    SXM             ; no sign extension

        stm     #t_start,AR3    ; AR3 = LTU start address
next1   ld      *AR2,-8,B       ; BL = MSByte
        call    CRCT            ; process MSByte
        ld      *AR2+,B         ;
        and     #00FFh,B        ; BL = LSByte
        call    CRCT            ; process LSByte
        banz    next1,*AR1-

        stl     A,@crc          ;store result

*************************************************************

*       CRCR (byte wise)

*************************************************************
        stm     #input,AR2      ; AR2 points to inputword
        mvdm    @nbDataIn,AR1   ; AR1 = length of message
        ld      #0,A            ; clear Acc A

next2   ld      *AR2,-8,B       ; BL = MSByte
        call    CRCR            ; process MSByte
        ld      *AR2+,B         ;
        and     #00FFh,B        ; BL = LSByte
        call    CRCR
        banz    next2,*AR1-

        stl     A,@crc          ;store result
done    b       done
```

```
*************************************************************

* CRCB routine : bit-wise CRC calculation

* input : 1) 16-bit words in AL
*         2) CRC generator polynomial in BH (left justified)
* ouput : CRC value in AH (left justified)

* BRC, REA, RSA, A, C, SP modified

* Code size = 10 words
* Cycles    = 11 + 5*N = 91 for N = 16

* A = |A31......A24....A16 A15.............A0|
*      <-- CRC --->        <---- input bits---->

* B = |B31......B24.........................B0|
*      <-- poly -->

*************************************************************
CRCB    stm     #16-1,BRC   ; BRC = 16-1
        rptb    CRC_end-1   ; repeat block
          sftl  A,1,A       ; A = A << 1, get MSB in carry
          PIPELINE
          PIPELINE
          xc    1,C         ; test on Carry
          xor   B,A      ; if C=1 then A = old_CRC XOR g(x)
CRC_end ret

*************************************************************

* CRCT routine : standard lookup table CRC calculation
*                with bytes as input

* input : 1) 8-bit words in AL
*         2) AR3 = LTU start address
*         3) DP = 0 = DP(temp)
* ouput : CRC value in AL

* A, B, AR4, SP modified

* code size =  6 words
* cycles = 8 cycles

* A = |A31.............A16 A15...... A7.......A0|
*                                   <--- CRC-->

*************************************************************
CRCT                    ; AL (low part of Acc A) = CRC bits
        xor     A,B     ; B = (inputByte) XOR (CRC)
                        ;   = Offset in LTU
        adds    @AR3,B  ; B = Offset + LTU start address
        stlm    B,AR4   ; AR4 = absolute address in LTU
        retd
        nop
        ldu     *AR4,A  ; AL = new CRC
                        ;    = LTU[inputbyte XOR CRC]
```

```
*************************************************************

* CRCR routine : reduced lookup table CRC calculation
*                with bytes as input

* input : 1) 8-bit words in AL
*         4) SXM = 0
* ouput : CRC value in AL

* A, B, AR3, C, SP modified

* code size =  14 words
* Cycles    = 13 + 5*N = 53 for N = 8 bits

* A = |A31.............A16 A15...... A7.......A0|
*                                   <--- CRC-->

*************************************************************
CRCR
        xor     A,B  ; B = (inputByte) XOR (CRC)
                     ;   = Offset in LTU
        ld      #0h,A
        stm     #8-1,BRC
        rptbd   #loop-1
         stm    #rt_start-1,AR3 ; AR3=LTU start address-1
          ror   B
          mar   *AR3+
          nop
          xc    1,C
          xor   *AR3,A  ; if C=1 then B = B XOR table[i++]
loop    ret

        .bss    input,257 ; augmented message
                          ; (8 bits appended)

crc     .usect  "scratchPad",32 ;scratch pad memory
temp    .set    crc+1
nbDataIn .set temp+1


        .sect   "table"
t_start
        .word   00h
        .word   01Dh
        .word   03Ah
……
        .word   0FEh
        .word   0D9h
        .word   0C4h
t_len   .set    256


        .sect   "reducedTable"
rt_start
        .word   01Dh
        .word   03Ah
        .word   074h
        .word   0E8h
```

```
            .word   0CDh
            .word   087h
            .word   013h
            .word   026h
rt_len    .set   8
```

Appendix E GSM FIRE Code Listing†

```
*************************************************************
*
* (C) Copyright 1997, Texas Instruments Incorporated
*
*************************************************************

        .mmregs
        .global Entry, Reset
        .title  "GSM Fire code"


hgenCRC .set   0482h    ;generator polynomial is equal to
lgenCRC .set   0009h    ;x40 + x26 + x23 + x17 + x3 + 1

inport  .set    0h

        .asg    NOP,PIPELINE
*************************************************************

*       Stack setup
*       Reset vector

*************************************************************
BOS     .usect  "stack",0fh     ; setup stack
TOS     .usect  "stack",1       ; Top of stack at reset

        .sect   "vectors"
Reset:
        bd      Entry           ; reset vector
        stm     #TOS,SP         ; Setup stack

*************************************************************

*       Main Program

*************************************************************
        .sect   "program"
Entry
        .include "c5xx.inc"

        rsbx    ovm
        rsbx    sxm
        ld      #crc,DP ; scratch pad mem -> DP=0

        portr   #inport,@nbDataIn       ; nb words

        addm    #-1,@nbDataIn           ; for repeat
```

† Part of the tables are truncated.

```
*************************************************************

*       CRCB (word wise)

*************************************************************
        stm     #input,AR2      ; copy inputwords in RAM
        rpt     @nbDataIn       ;
         portr  #inport,*AR2+   ; read them from IO space

        ld      #hgenCRC,16,B   ; BH = CRC gen. poly. MSBs
        or      #lgenCRC,B      ; BL = CRC gen. poly. LSBs
        mvdm    @nbDataIn,AR1   ; AR1 = length of message
        stm     #input,AR2      ; AR2 points to input word

        stm     #16,BK          ; Circular buffer size is 16
        ld      #0,A            ; clear Acc A
        stm     #16-1,BRC       ; BRC = 16-1
        rptbd   initbitpos-1     ; repeat block
         stm    #bitpos,AR3     ; AR3 -> bitpos
          stl   A,*AR3+%        ; initialize bitpos
                                ;with bit positions
          add   #1,A
initbitpos                      ; end of repeat
        mar     *AR3+%          ;
        stm     #0,T            ; init T reg
        dld     *AR2+,A         ; CRC register intialized
next    calld   CRCB            ; perform CRC computation
         stm    #16-1,BRC       ; BRC = 16 - 1
        banzd   next,*AR1-      ; process all input words
         mar    *AR2+           ; AR2 points to next input
         nop
        calld   CRCB            ; last 8 bits
         stm    #8-1,BRC        ; BRC = 8-1

        mvdk    @AG,@crc        ; store result in memory
        mvdk    @AH,@crc+1
        mvdk    @AL,@crc+2


*************************************************************

*       CRCT (byte wise)

*************************************************************
        stm     #input,AR2      ; AR2 points to inputword
        mvdm    @nbDataIn,AR1   ; AR1 = length of message
        ld      #0,A            ; clear ACC A

        stm     #t_start,AR3    ; AR3 = LTU start address
        stm     #3,T            ; multiplication operand
                                ; (for calculating index
                                ;  in LTU table)
next1   calld   CRCT            ; process LSByte
         ld     *AR2,-8,B       ; BL = LSByte
        ld      *AR2+,B
        calld   CRCT            ; process MSByte
         and     #0FFh,B        ; BL = MSByte
        banz    next1,*AR1-
```

```
        mvdk    @AG,@crc        ; store result in memory
        mvdk    @AH,@crc+1
        mvdk    @AL,@crc+2

done    b       done

*************************************************************

* CRCB routine : bit-wise CRC calculation

* input : 1) 16-bit input words in AL
*         2) CRC generator polynomial in BH
*         3) OVM = 0
* ouput : CRC value in A (40 bits)

* BRC, REA, RSA, A, C, T, TC, AR3, SP modified

* Code size = 12 words
* Cycles    = 9 + 11*N = 137 for N = 16 bits

* A = |A39..A32 A31.....A16 A15.............A0|
*      <----------------- CRC ------------------>

* B = |B39..B32 B31....B16 B15.............B0|
*      <--------------polynomial -------------->
*************************************************************

CRCB
        rptb     CRCB_end-1      ; repeat block
          bitt   *AR2           ; Test bit #(15-T)
          sfta   A,1,A          ; A = A << 1
          ld     *AR3+%,T       ; T = next bit position
          xc     1,TC
          or     #1,A           ; if TC=1, A(LSB) = 1
          xc     1,C
          xor    B,A            ; if C=1, A = new CRC
CRCB_end
        ret



*************************************************************

* CRCT routine : standard lookup table CRC calculation
*                with bytes as input

* input : 1) 8-bit words in AL
*         2) AR3 = LTU start address
*         3) DP = 0 = DP(temp)
*         4) SXM = 0
*         4) FRCT = 0
* ouput : CRC value in AL

* A, B, AR4, SP modified

* code size =  18 words
* cycles = 20 cycles
```

```
* A = |A31.............A16 A15................A0|
*                          <------- CRC ------->

* B = |B31.......................B7........B0|
*                                <-- input-->


*************************************************************
CRCT
        mvmd    AG,@AR4 ; AR4 = CRC >> 32
        xor     @AR4,B  ; B = (inputByte) XOR (CRC>>32)
                        ;   = Offset in LTU
        sfta    A,8,A   ; A = A << 8
        mpy     @BL,B
        add     @AR3,B  ; B = Offset + LTU start address
        stlm    B,AR4   ; AR4 = absolute address in LTU
        PIPELINE
        PIPELINE
        mvdk    *AR4+,@BG
        mvdk    *AR4+,@BH
        mvdk    *AR4+,@BL
        retd
        xor     B,A     ; A = new CRC
        nop

.bss    input,259       ; augmented message
                                ; (40 bits appended)
bitpos  .usect "circ",16        ; contain bit position

crc     .usect  "scratchPad",32 ;scratch pad memory
nbDataIn .set   crc+3
temp    .set    crc+4   ;must be aligned to even boundary


        .sect   "table"
t_start .word   00h
        .word   00h
        .word   00h
        .word   00h
        .word   0482h
        .word   09h
……
        .word   03h
        .word   086FCh
        .word   070Eh
        .word   03h
        .word   0827Eh
        .word   0707h
t_len   .set    768
```

TI Contact Numbers

INTERNET

TI Semiconductor Home Page
www.ti.com/sc

TI Distributors
www.ti.com/sc/docs/distmenu.htm

PRODUCT INFORMATION CENTERS

Americas
Phone +1(972) 644-5580
Fax +1(972) 480-7800
Email sc-infomaster@ti.com

Europe, Middle East, and Africa
Phone
- Deutsch +49-(0) 8161 80 3311
- English +44-(0) 1604 66 3399
- Español +34-(0) 90 23 54 0 28
- Francais +33-(0) 1-30 70 11 64
- Italiano +33-(0) 1-30 70 11 67

Fax +44-(0) 1604 66 33 34
Email epic@ti.com

Japan
Phone
- International +81-3-3457-0972
- Domestic 0120-81-0026

Fax
- International +81-3-3457-1259
- Domestic 0120-81-0036

Email pic-japan@ti.com

Asia
Phone
- International +886-2-23786800
- Domestic
  - Australia 1-800-881-011
    - TI Number -800-800-1450
  - China 10810
    - TI Number -800-800-1450
  - Hong Kong 800-96-1111
    - TI Number -800-800-1450
  - India 000-117
    - TI Number -800-800-1450
  - Indonesia 001-801-10
    - TI Number -800-800-1450
  - Korea 080-551-2804
  - Malaysia 1-800-800-011
    - TI Number -800-800-1450
  - New Zealand 000-911
    - TI Number -800-800-1450
  - Philippines 105-11
    - TI Number -800-800-1450
  - Singapore 800-0111-111
    - TI Number -800-800-1450
  - Taiwan 080-006800
  - Thailand 0019-991-1111
    - TI Number -800-800-1450
  - Fax 886-2-2378-6808
  - Email tiasia@ti.com

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgement, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

CERTAIN APPLICATIONS USING SEMICONDUCTOR PRODUCTS MAY INVOLVE POTENTIAL RISKS OF DEATH, PERSONAL INJURY, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE (“CRITICAL APPLICATIONS"). TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. INCLUSION OF TI PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE FULLY AT THE CUSTOMER'S RISK.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty, or endorsement thereof.