Implementing CRCCs in Altera Devices

July 1995, ver. 1 Application Note 49

Introduction

Redundant encoding is a method of error detection that spreads the information across more bits than the original data. The more redundant bits you use, the greater the chance that you will detect transmission errors. For example, a 16-bit increase in data record length misses 1 error in 65,536, while a 32-bit increase misses only 1 error in over 4 billion.

Although these odds are favorable, undetectable errors can still occur. The nature of common errors varies depending on the transmission and storage medium, but undetectable errors can be caused by occasional isolated changed bits or short bursts of multiple changed bits. To minimize undetectable errors, you can distribute the data so that it is less likely that transmission errors will result in a valid encoding of an alternate set of data.

Cyclic redundancy codes (CRCs) are a favored type of redundant encoding. Cyclic redundancy code checkers (CRCCs) check for differences between transmitted data and the original data. Data transmission applications use CRCCs extensively. For example, the Asynchronous Transfer Mode (ATM) specification requires a CRCC that is implemented across the entire payload to ensure data integrity.

CRCCs are particularly effective for two reasons:

- They provide excellent protection against common errors, such as burst errors where consecutive bits in a data stream are corrupted during transmission.
- The original data is the first part of the transmission, which makes systems that use CRCCs easy to understand and implement.

When you use a CRCC to verify a frame of data, the frame is treated as one very large binary number, which is then divided by a *generator* number. This division produces a remainder, which is transmitted along with the data. At the receiving end, the data is divided by the same generator number and the remainder is compared to the one sent at the end of the data frame. If the two remainders are different, then an error occurred during the data transmission.

Creating CRCCs

In general, digital logic does not implement the division of very large numbers efficiently. Consequently, binary information must be converted into a more appropriate form before CRCCs are used. The string of bits to be verified is represented as the coefficients of a large polynomial, rather than as a large binary number, as shown in the following example:

$$1,0001,0000,0010,0001 = X^{16} + X^{12} + X^5 + 1$$

To make the calculations easier to implement, the arithmetic is cast in a binary algebraic field. The arithmetic is implemented modulo 2 with no carry, i.e., addition and subtraction are identical (implemented via XOR). Multiplication by 1 or 0 is performed with an AND function, and division by the binary field's single non-zero element, 1, leaves the dividend unchanged. In this system, any remainder for a polynomial of degree *n* is no more than *n*–1 bits long. Therefore, even though polynomials of order 16 have 17 terms (including X^0), any remainder is contained within 16 bits (X^{15} through X^0).

When implementing CRCCs, both the original data and the generator number must be represented as polynomials. The generator number is therefore called the *generator polynomial*. The polynomial that represents the original data is multiplied by X^n, where *n* is the degree of the generator polynomial (i.e., the length of the CRC). This operation shifts the data to the left by *n* bits. The resulting 0s at the end of the polynomial allow you to add the CRC to the data polynomial by replacing the last *n* bits (which have become 0) with the CRC.

Since addition and subtraction are equivalent, this operation also produces a polynomial that is evenly divisible by the generator polynomial. Therefore, when the data polynomial plus the CRC is divided by the generator polynomial at the receiving end of the system, the remainder for an error-free transmission is always 0.

In summary, the data D is multiplied by X^n and divided by the generator polynomial G. The quotient Q is then discarded, and the remainder R is added to the dividend $X^n \times D$. See the following equation:

$$(X^n \times D) + R = (Q \times G) + 0$$

At the receiving end, the first part of the transmitted information is the original data D; the second part (the last *n* bits) is the remainder R. This entire quantity is divided by the same generator polynomial G, and the quotient Q is discarded. The remainder of this division is always 0 if there are no errors in the transmitted data.

Error Detection

Generally, CRCCs detect the following types of errors:

- Single-bit errors
- Two-bit errors
- Three-bit and other odd-number-bit errors
- Burst errors that are less than or equal to the CRC length
- Most burst errors that are greater than the CRC length

The types of errors that a CRCC detects depends on the generator polynomial. Table 1 shows several common generator polynomials for various applications.

Table 1. Common Generator Polynomials

Generator Name	Polynomial
SDLC (CCITT)	$X^{16} + X^{12} + X^5 + X^0$
SDLC Reverse	$X^{16} + X^{11} + X^4 + X^0$
CRC-16	$X^{16} + X^{15} + X^2 + X^0$
CRC-16 Reverse	$X^{16} + X^{14} + X^1 + X^0$
CRC-12	$X^{12} + X^{11} + X^3 + X^2 + X^1 + X^0$
Ethernet	$X^{32} + X^{26} + X^{23} + X^{22} + X^{16} + X^{12} + X^{11} + X^{10} + X^8 + X^7 + X^5 + X^4 + X^2 + X^1 + X^0$

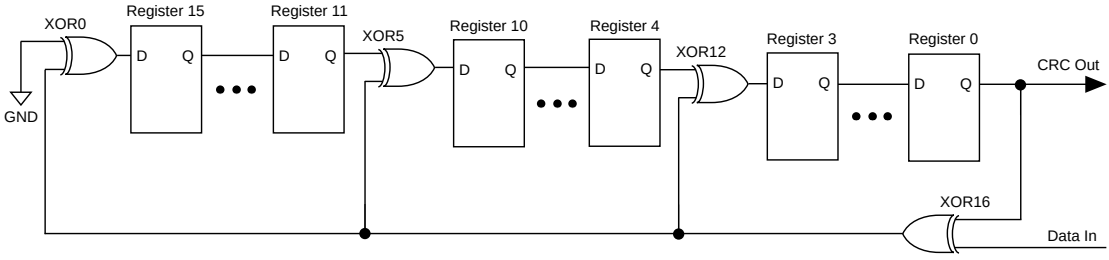
Different generator polynomials have different error-detection capabilities. To achieve optimum results, you must use a generator polynomial that effectively detects and corrects the expected transmission errors. Alternatively, you can use a standard generator polynomial that is appropriate for a particular transmission medium.

Serial CRC Computation

Typically, CRC calculations are implemented with linear-feedback shift registers (LFSRs). LFSRs use a method that yields the same results as the subtract and shift division process when the subtraction is performed without carry by the XOR function. To affect the subtract and shift division one bit at a time, you can shift through and examine each bit in the original frame of data (i.e., the dividend). For the first bit of value 1, the divisor high-order bit is subtracted (XORed) from the dividend. That dividend bit, which is unnecessary and is not generated, is set to zero by the subtraction. The lower order bits of the divisor cannot be subtracted yet, because the corresponding divisor bits have not been shifted in.

Figure 1 shows the Consultative Committee International Telegraph and Telephone (CCITT) CRC-16 generator computed serially. The quotient "1" bit (XOR16) is fed back and subtracted from the appropriate taps in the shift register (XOR12, XOR5, XOR0). These bits shift forward to appear at the end and are subtracted from the next data bit, unless they are eliminated by a coinciding second bit from an earlier or later subtraction that is fed back.

Figure 1. LFSR Configuration for CCITT CRC-16 Generator ($X^{16} + X^{12} + X^5 + X^0$)



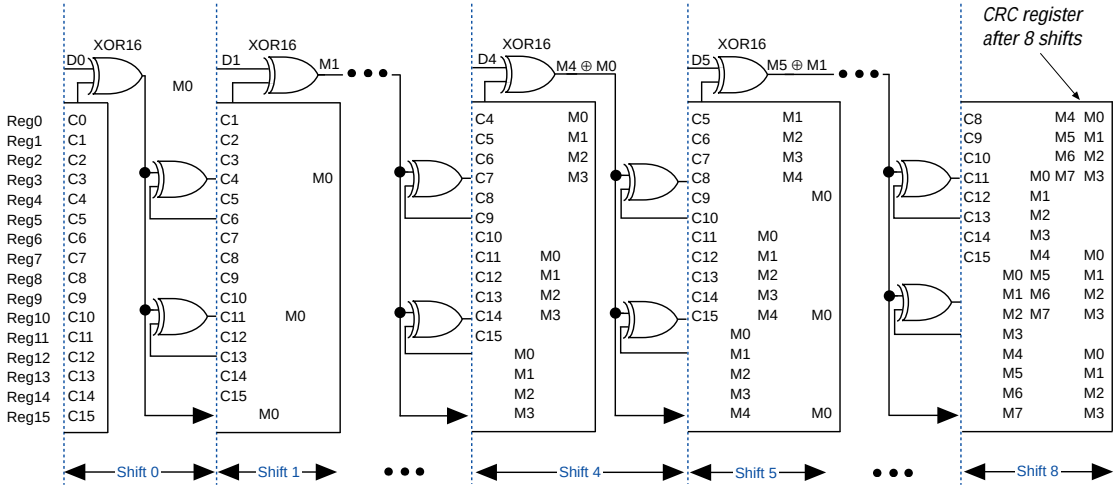
Parallel CRC Computation

The serial method works well when the data is available in bit-serial form. However, today's high-speed signal processing systems process data in byte, word, double-word (32-bit), or larger widths rather than serially. Even in telecommunications systems, which transmit data serially, received data is encapsulated within a VLSI device responsible for Clock recovery and byte framing. The data is presented to the board designer in 8-bit frames at a manageable speed. Therefore, designs for high-speed devices require the CRC to be calculated 8 bits at a time.

You can develop a parallel CRC algorithm with the LFSR approach. You can express the contents of the shift register after 8 shifts as a function of the initial contents of the shift register and the 8 data bits shifted in. This function can be created using only XOR operators. Figure 2 shows the function that uses the CCITT CRC-16 generator polynomial (X16 + X12 + X5 + X0). In this example, the register contents C[15..0] and the next 8 data bits D[7..0] are used to calculate the general contents of the register after 8 shifts. Because the register shifts once for each input data bit, the XOR16 operator produces the series of terms C0 ⊕ D0, C1 ⊕ D1, etc. These terms are assigned the name M*n* where M*n* = C*n* ⊕ D*n*.

Figure 2. CRC Register Shifts for CCITT CRC-16 Generator ($X^{16} + X^{12} + X^5 + X^0$)

All terms on a line are XORed together.



On the first four shifts, only M*n* terms are fed back. On the fifth shift, register 0 now contains C4 ⊕ M0 (from earlier shifts), which will be XORed with D4 and fed back, as shown in the following equation:

$$D4 \oplus (C4 \oplus M0) = (D4 \oplus C4)\ M0 = M4 \oplus M0$$

Later shifts all have multiple terms fed back. The CRC register contents after 8 shifts are shown in the box on the right.

AHDL CRC Macrofunction

Figure 3 shows an excerpt from an Altera Hardware Description Language (AHDL) Text Design File (**.tdf**) that includes the logic equations for the register values after 8 shifts (i.e., 1 byte later).

Figure 3. AHDL CRC Design Excerpt

```
ex0 = ( reg[0] $ dat[0] ) ;
ex1 = ( reg[1] $ dat[1] ) ;
ex2 = ( reg[2] $ dat[2] ) ;
ex3 = ( reg[3] $ dat[3] ) ;
ex4 = ( reg[4] $ dat[4] ) ;
ex5 = ( reg[5] $ dat[5] ) ;
ex6 = ( reg[6] $ dat[6] ) ;
ex7 = ( reg[7] $ dat[7] ) ;

reg[0].d   =   reg[8]            $ ex4   $ ex0 ;
reg[1].d   =   reg[9]            $ ex5   $ ex1 ;
reg[2].d   =   reg[10]           $ ex6   $ ex2 ;
reg[3].d   =   reg[11]     $ ex0 $ ex7   $ ex3 ;
reg[4].d   =   reg[12]     $ ex1               ;
reg[5].d   =   reg[13]     $ ex2               ;
reg[6].d   =   reg[14]     $ ex3               ;
reg[7].d   =   reg[15]     $ ex4         $ ex0 ;
reg[8].d   =          ex0  $ ex5         $ ex1 ;
reg[9].d   =          ex1  $ ex6         $ ex2 ;
reg[10].d  =          ex2  $ ex7         $ ex3 ;
reg[11].d  =          ex3                    ;
reg[12].d  =          ex4                $ ex0 ;
reg[13].d  =          ex5                $ ex1 ;
reg[14].d  =          ex6                $ ex2 ;
reg[15].d  =          ex7                $ ex3 ;
```

A complete AHDL description of the CCITT CRC-16 macrofunction is available in the self-extracting file **an_049.exe** from the Altera bulletin board service (BBS) at (408) 954-0104 or from Altera's FTP site.

A CRC is calculated before a frame of data is transmitted, then each data byte is clocked into the CRC macrofunction. Conversely, the CRC is clocked during each byte of transmitted data. After the last byte of data is clocked in, the 16-bit CRC is appended to the data. A multiplexer selects the low CRC byte (`REG[7..0]`), then selects the high byte for transmission.

To check the CRC after the data transmission, you should clear the shift register by asserting the Clear signal. The CRC macrofunction is enabled for one Clock cycle during each byte of received data, including the two CRC bytes calculated at transmission end and appended to the end of the data frame. After the last CRC byte is clocked in, the register contains the 16-bit CRC of the transmitted data plus the transmitted CRC, as shown in the following equation:

$$\frac{X^n \times D + R}{G}$$

If there are no errors in the transmission, the remainder is 0, and the `CRC00` signal is asserted.

Piecewise CRCC Computation

In packet-switching networks (such as ATM networks), a frame is broken into small packets for transmission, then reassembled into the original long frame after it is received. The frame can include a CRC at the end that allows the CRCC to verify the integrity of the entire frame once it is reassembled. It is easier to generate a CRC while the frame is still in its original format. For more complex transmission protocols, CRC generation may require a communications processor, which can easily become overburdened. To avoid the need for a communications processor entirely, you can compute the CRC in hardware at a lower level. However, his method can cause packets from different frames to intermingle, especially on the receiving end.

To prevent intermingling frames, you can compute the CRC for each packet separately, i.e., piecewise. Once the CRCC has been executed over a packet, the interim CRC is stored in a small RAM that is indexed by the channel number. For the next packet, the channel number is first determined from the header, and the interim CRC for the previous channel is extracted from the RAM and loaded into the CRC register. The packet's data is then clocked into the CRC. At the end of the packet, the interim CRC for the next packet is stored in the RAM in preparation for the next packet. Sometimes the data path needs to be delayed by a few bytes to allow cycles to acquire the channel number and load the previous interim CRC into the CRC macrofunction. When the last packet of a frame is transmitted, the CRC is appended to the data so that it can be verified as 0 when it is received.

The piecewise method adds RAM and control logic to the design. However, this method has little effect on the CRC engine; the register must simply be loaded with the interim CRC for the previous channel before proceeding with each packet's data.

A sample file of the CRC macrofunction with an added load function is available in the self-extracting file **an_049.exe** from the Altera bulletin board service (BBS) at (408) 954-0104 or from Altera's FTP site.

Each frame's packets must be processed in the same order at transmission and reception. If any of the packets are out of sequence when they are received, the CRCC indicates an error. Bit or burst errors are also detected and reported.

Conclusion

CRCCs are an efficient method for verifying data transmission, especially for ATM applications. CRCCs offer protection from common errors without adding extensive logic overhead. You can use either serial or parallel computation methods to implement CRCCs. For applications in which the data frame is segmented and mixed with data from other frames, a piecewise CRC calculation is most effective.

Bibliography

Hamming, R.W. ***Coding and Information Theory***. Prentice-Hall, 1980.

Lee, R. "Cyclic Code Redundancy." *Digital Design*, 7/81.

Perez, A. "Byte-wise CRC Calculations." *IEEE MICRO*, 6/83.

Peterson, W.W., and Brown, D.T. "Cyclic Codes for Error Detection." *Proc. IRE*, 1/61.

Peterson, W.W. and Weldon, E.J. ***Error-Correcting Codes***. MIT Press, 1961.

Pless, Vera. ***Introduction to the Theory of Error-Correcting Codes***. John Wiley & Sons, 1982.

Ramabadran, T.V. and Gaitonde, S.S. "A Tutorial on CRC Computations." *IEEE MICRO*, 8/88.

2610 Orchard Parkway
San Jose, CA 95134-2020
(408) 894-7000
Applications Hotline:
(800) 800-EPLD
Customer Marketing:
(408) 894-7104
Literature Services:
(408) 894-7144