Parallel Cyclic Redundancy Check (CRC) for HOTLink™

Introduction

This application note discusses using CRC codes to ensure data integrity over high-speed serial links, such as Fibre Channel, ESCON™ and other interfaces supported by Cypress's CY7B923/CY7B933 HOTLink™ devices. It also shows why parity and Hamming codes are not useful, and describes common CRC codes used in high-speed communications systems. Finally, algorithms for parallel calculation of CRC–16 and CRC–32 are presented.

Why Not Parity (or Why Some Parallel Interface Practices Don't Apply in the Serial World)?

Some systems go to great lengths to detect data errors. Parity is often used with parallel forms of data, on buses or memories, to detect some errors. It provides a small measure of robustness by detecting certain bit errors with minimal redundancy. However, while parity can detect single-bit errors, it can detect only half of all multiple-bit errors.

Other systems go further, employing Hamming codes to not only detect, but in many instances correct, bit errors. Both of these approaches are applied to data in its parallel form. Unfortunately, the use of a Hamming code requires many more bits of redundancy, per character or word, than parity.

For transmission of data on high-speed serial channels, the most prevalent errors are multi-bit bursts. These multi-bit errors make parity worthless, and severely limit the effectiveness of single-bit correcting Hamming codes.

The large amount of redundancy in a Hamming code (7 bits to protect a 32-bit word) also makes it a poor choice to protect data across a serial link. Transmission of the redundant bits in each word can easily consume a fifth of the available link bandwidth, or require operation of the link at a 20% faster transfer rate to carry the redundant bits.

In reality, bit errors of any type are quite rare in these links (<< 1 in 10^{12} bits). Since these errors cannot generally be corrected by a Hamming code or detected by character parity, the transmission overhead of these types of detection/correction bits becomes a poor use of link bandwidth. In systems where data is sent serially across a link, the data integrity of the link can be much better verified using Cyclic Redundancy Check (CRC) codes.

CRC Codes

CRC codes make use of a Linear Feedback Shift Register (LFSR) to generate a signature based on the contents of any data passed through it. This signature can be used to detect the modification or corruption of bits in a serial stream.

CRC–16 and CRC–32

In general CRC codes are able to detect:

- All single- and double-bit errors.
- All odd numbers of errors.
- All burst errors less than or equal to the degree of the polynomial used.
- Most burst errors greater than the degree of the polynomial used.

CRC codes have been used for years to detect data errors on interfaces, and their operation and capabilities are well understood. Two codes that have found wide use are CRC–16 and CRC–32. As the names imply, CRC–16 makes use of a 16-bit LFSR, while CRC–32 uses a 32-bit LFSR. Additional information on CRC codes can be found in the references at the end of this application note.

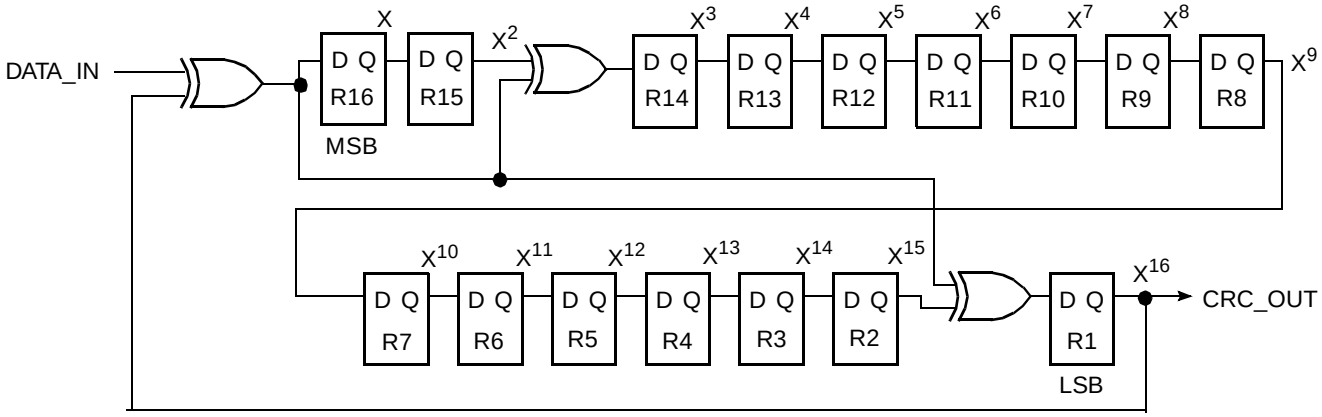

Figure 1. Linear Feedback Implementation of CRC–16

The generator polynomial for CRC–16 is listed in Equation 1, and the polynomial for CRC–32 is listed in Equation 2. These CRC codes are traditionally calculated on the serial data stream using a Linear Feedback Shift Registers (LFSR) built from flip-flops and XOR gates, as shown in *Figure 1*. The structure for the CRC–32 polynomial is similar to *Figure 1*, but with twice the number of flip-flops.

$$G(X) = X^{16} + X^{15} + X^2 + 1 \quad \text{Eq. 1}$$

$$G(X) = X^{32} + X^{26} + X^{23} + X^{22} + X^{16} + X^{12} + X^{11} + X^{10} + X^8 + X^7 + X^5 + X^4 + X^2 + X + 1 \quad \text{Eq. 2}$$

In these equations, the superscripts identify the tap location in the shift register. The order of the polynomial is identified by the highest order term, and specifies the number of flip-flops in the shift register. Since these polynomials are for modulo-2 arithmetic, each bit-shift is equivalent to a multiply by 2.

Development of a Parallel Implementation

When used with high-speed serial data, especially data which is encoded in the serial domain, it becomes quite difficult to implement the CRC calculation using a shift register. However, it is possible to convert a serial implementation into a parallel form that accumulates multiple bits in each clock cycle. The following paragraphs and tables describe how the CRC–16 polynomial is converted to calculate eight bits at a time (i.e., a byte basis). The CRC–32 polynomial is converted using a similar procedure, with the results calculated 16 bits at a time (on a half-word basis). The results for CRC–32 are presented in *Tables 5* and *6*, but without the intermediate calculations. The generation of these intermediate equations are left as an exercise for the reader.

Implementation

First, a few notes:

- R_i is the *i* th bit of the CRC register.
- C_i is the contents of *i* th bit of the *initial* CRC register, before any shifts have taken place.
- R1 is the least significant bit (LSB).
- The entries under each CRC register bit indicate the values to be XORed together to generate the content of that bit in the CRC register.
- D_i is the data input, with LSB input first.
- D8 is the MSB of the input byte, and D1 is the LSB.
- A substitution is made to reduce the table size, such that $X_i = D_i$ XOR C_i.

The results of the CRC are calculated one bit at a time and the resulting equations for each bit are examined. The CRC register prior to any shifts is shown in *Table 1*. The CRC register after a single bit shift is shown in *Table 2*. The CRC register after two shifts is shown in *Table 3*.

This process continues until eight shifts have occurred. *Table 4* lists the CRC register contents after eight shifts. X_i was substituted for the various D_i XOR C_i combinations. The following properties were used to simplify the equations:

- Commutativity (A XOR B = B XOR A).
- Associativity (A XOR B XOR C = A XOR C XOR B).
- Involution (A XOR A = 0).

A study of *Table 4* reveals two interesting facts:

- The most-significant byte (bits R16–R9) of the CRC register is only dependent on XOR combinations of the initial low-order byte of the CRC register and the input byte.

Table 1. CRC–16 Register prior to any shifts

R16	R15	R14	R13	R12	R11	R10	R9	R8	R7	R6	R5	R4	R3	R2	R1
C16	C15	C14	C13	C12	C11	C10	C9	C8	C7	C6	C5	C4	C3	C2	C1

Table 2. CRC–16 Register after One Shift

R16	R15	R14	R13	R12	R11	R10	R9	R8	R7	R6	R5	R4	R3	R2	R1
C1 D1	C16	C15 C1 D1	C14	C13	C12	C11	C10	C9	C8	C7	C6	C5	C4	C3	C2 C1 D1

Table 3. CRC–16 Register after Two Shifts

R16	R15	R14	R13	R12	R11	R10	R9	R8	R7	R6	R5	R4	R3	R2	R1
C2 D2 C1 D1	C1 D1	C16 C2 D2 C1 D1	C15 C1 D1	C14	C13	C12	C11	C10	C9	C8	C7	C6	C5	C4	C3 C2 D2 C1 D1

Table 4. CRC–16 Register after Eight Shifts

R16	R15	R14	R13	R12	R11	R10	R9	R8	R7	R6	R5	R4	R3	R2	R1
X8 X7 X6 X5 X4 X3 X2 X1	X7 X6 X5 X4 X3 X2 X1	X8 X7	X7 X6	X6 X5	X5 X4	X4 X3	X3 X2	C15 X2 X1	C15 X1	C14	C13	C12	C11	C10	C9 X8 X7 X6 X5 X4 X3 X2 X1

- The least-significant byte (bits R8–R1) of the CRC register is dependent on the XOR combination of the initial lower eight bits of the CRC register, the input data byte, and the initial contents of the high-order bits of the CRC register.

This allows the next value of the CRC register to be calculated as an XOR of the input data character bits, and a constant determined by the present contents of the CRC register. For example, calculating a new value for R9 is accomplished by calculating X3 and X2 and exclusive-ORing them together.

Implementation Issues for CRC–16 Parallel Algorithm

The most significant byte of the CRC register is based on eight data inputs and eight register values. By presenting these as address inputs to a 64Kx8 PROM, it is possible to directly output the next state of the CRC register. A 40-MHz maximum byte rate would dictate a total cycle time of 25 ns or less, which is available in the CY27H512.

The lower byte of the CRC register only contains three values that require any type of calculation (R8, R7, and R1). Of these, R1 is the equivalent of the input for R16 XORed with R9. This, and the inputs for R7 and R8, can be calculated in a small PLD like a PALCE22V10.

Both of these may also be implemented using a single level of XOR gates to calculate the X1 through X8 values, and a pair of 256x8 EPROMs.

Another approach is to calculate the XOR functions directly in logic, as one would do with a Field Programmable Gate Array (FPGA) or CPLD. From *Table 4,* the largest XOR to be calculated is that for R1, which contains 17 terms. Implementation of a large XOR structure consumes large numbers of product terms in a CPLD, however, many of the XOR terms are common across the various inputs of the CRC register.

At an XOR width of 17, it is not possible to implement this in a single level of logic within current Cypress CPLDs. However, XOR factoring makes it is possible to implement this in two levels of logic. With CPLD single-level delays of 10 ns (or less), the CRC may be implemented in a single CPLD. By ensuring that the input data is pipelined through an internal register, the timing is determined only by internal delays in the device. The CRC–16 parallel algorithm can be implemented using *Warp3*® (Cypress's VHDL-based CPLD design tool) in a number of FLASH370i CPLD devices. The design will run at the 40-MHz maximum parallel data rate supported by HOTLink.

Description of CRC–32 Parallel Algorithm

The parallel algorithm for CRC–32 is derived in the same manner as CRC–16. The differences here are that data is now handled 16 bits (a half-word) at a time, the CRC register is now 32 bits in length, and a different polynomial is used.

Table 5 contains the XOR information for the least-significant half-word (LSHW) of the CRC–32 register after 16 shifts, and *Table 6* contains the XOR information for the most-significant half-word (MSHW) of the CRC–32 register after 16 shifts. Again, note that the MSHW only depends on XOR combinations of the initial lower-order bits of the CRC–32 register and the input data. The LSHW depends on XOR combinations of the initial lower-order bits of the CRC–32 register, the input data, and the initial MSHW of the CRC–32 register.

Table 5. CRC–32 Register (LSW) after 16 Shifts with X_i Substitution

R16	R15	R14	R13	R12	R11	R10	R9	R8	R7	R6	R5	R4	R3	R2	R1
C32 X1 X3 X8 X10 X11 X12 X16	C31 X2 X7 X9 X10 X11 X15	C30 X1 X6 X8 X9 X10 X14	C29 X5 X7 X8 X9 X13	C28 X4 X6 X7 X8 X12	C27 X3 X5 X6 X7 X11	C26 X2 X5 X7 X16	C25 X1 X7 X10 X15 X16	C24 X6 X9 X14 X15	C23 X5 X8 X13 X14	C22 X6 X10 X12 X13 X16	C21 X5 X9 X11 X12 X15	C20 X4 X8 X10 X11 X14	C19 X3 X7 X9 X10 X13	C18 X2 X6 X8 X9 X12	C17 X1 X5 X7 X8 X11

Table 6. CRC–32 Register (MSW) after 16 Shifts

R32	R31	R30	R29	R28	R27	R26	R25	R24	R23	R22	R21	R20	R19	R18	R17
X4 X6 X7 X10 X16	X3 X4 X5 X7 X9 X10 X15 X16	X2 X3 X7 X8 X9 X10 X14 X15 X16	X1 X2 X6 X7 X8 X9 X13 X14 X15	X1 X4 X5 X8 X10 X12 X13 X14 X16	X3 X6 X8 X10 X11 X12 X13 X15 X16	X2 X5 X8 X9 X10 X11 X12 X14 X15	X1 X6 X8 X9 X11 X13 X14 X16	X4 X5 X6 X8 X12 X13 X15 X16	X3 X4 X5 X7 X11 X12 X14 X15	X2 X3 X4 X6 X10 X11 X13 X14	X1 X2 X3 X4 X5 X6 X7 X9 X12 X13 X16	X1 X2 X3 X5 X7 X8 X10 X11 X12 X15 X16	X1 X2 X4 X6 X7 X9 X10 X11 X14 X15	X1 X3 X5 X6 X8 X9 X10 X13 X14	X2 X4 X5 X7 X8 X9 X12 X13

Implementation Issues for CRC–32 Parallel Algorithm

The issues confronting a designer wishing to implement this algorithm are the same as those for the CRC–16 algorithm, except that the magnitude of the problem is increased. Implementation of the X_i calculation in a look-up table now requires 16 inputs and, since we are calculating 16 bits at a time, 16 outputs. This implies a 64K x 16 EPROM. The difference is that HOTLink takes parallel data at a maximum of 40 MBytes per second, and the CRC is calculated two bytes at a time. This gives approximately two clock cycles (50 ns) to perform the calculation. Using an EPROM look-up table requires an extra component (EPROM and XOR array), compared with implementing the entire design in a CPLD.

Tables 5 and *6* show that the largest XOR is a 22-term function feeding CRC registers R20 or R21. Within a CPLD, it is not possible to implement this in a single level of logic. However, using the slowest FLASH370i CPLDs available (66 MHz), it is possible to implement this in three levels of logic and still be under the 50 ns delay limit. This is one more than required for the CRC–16 implementation, but remember, there is twice as much time available to calculate the CRC–32.

As data rates increase, so do timing constraints. However, for the data rates supported by the HOTLink devices (150 to 400 MBaud), the FLASH370i devices can successfully implement a parallel calculation of CRC–32.

Figure 2 is a graphical representation of the logic delays in the XOR-tree portion of the CRC–32 design. Three layers of logic at 15 ns each (for a -66 CPLD) give a maximum internal delay of 45 ns. Note that there are no routing delays in FLASH370i devices. This is comfortably under the 50 ns figure mentioned earlier.

Conclusion

This application note shows how to calculate a parallel implementation of any CRC polynomial, and the equations for CRC–16 and CRC–32 are provided. Implementation and performance issues for the CRC–16 and CRC–32 polynomials was presented. Both designs easily operate at the fastest character rate supported by Cypress's HOTLink devices.

Additional information on usage of CRC polynomials may be found in the following references and in the Cypress application note titled "Drive ESCON With HOTLink."

Reference

1. A. Perez, "Byte-wise CRC Calculations," *IEEE MICRO*, June 1983, pp. 40-50.
2. A. K. Pandeya and T. J. Cassa, "Parallel CRC Lets Many Lines Use One Circuit," *Computer Design*, Sept. 1975, pp 87-91.
3. R. Swanson, "Understanding Cyclic Redundancy Codes," *Computer Design*, Nov. 1975, pp. 93-99.

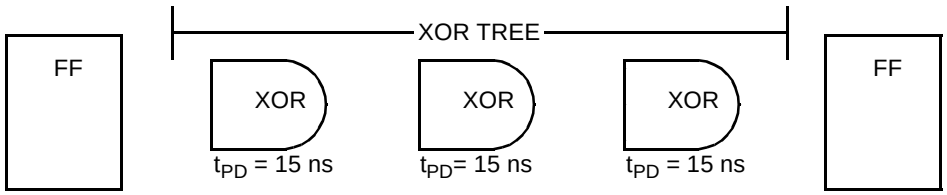

Figure 2. Critical Path Logic Delay Estimate for CRC-32

HOTLink is a trademark and *Warp3* is a registered trademark of Cypress Semiconductor.
ESCON is a trademark of IBM.