Parallel Cyclic Redundancy Check (CRC) for HOTLink™

Introduction

This note discusses using CRC codes to insure data integrity over high-speed serial links, such as Fibre Channel, ESCON™ and other standards supported by Cypress's CY7B923 and CY7B933 HOTLink™ devices. It also shows why parity is not useful and then describes the most common CRC codes (CRC−16 and CRC−32) used in high-speed communications systems. Finally, an algorithm for calculating CRC−16 one byte at a time and for calculating CRC−32 two bytes at a time is presented. These algorithms are necessary because of the impracticality of running programmable devices at the 155 to 330 MHz required to calculate the CRCs in a bit-serial fashion.

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 corruption. Parity is sometimes used to provide a small measure of robustness by detecting certain bit errors. But, while parity can detect single-bit errors in data, it can detect only half of all multiple-bit errors in the same data. Other systems go further, employing CRC codes to not only detect, but in many instances correct, single- or even multi-bit errors. (Error Detection and Correction (EDAC) circuits use CRC codes.) Both of these approaches are applied to data in its parallel form.

It turns out that double-bit errors are the most prevalent error mode for single-event (burst) errors in high-speed serial channels, so parity is not very useful. Since an error of any kind is rare in typical systems (1 in 10^{12} bits), errors of greater than two bits are only a small portion of this number, which leaves us with predominantly double-bit errors with which to deal.

CRC−16 and CRC−32

In general CRC codes will 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

So, CRC codes and their ability to detect multiple-bit errors are more suitable to systems employing high-speed serial links. Much work has been done with CRC codes and two suitable codes are CRC−16 and CRC−32. As the names imply, CRC−16 is 16 bits long and CRC−32 is 32 bits long. Additional information can be found in the references at the end of this note.

CRC−16 has a generator polynomial $G(x) = x^{16} + x^{15} + x^2 + 1$ and CRC−32 has a $G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$. The traditional way of calculating these CRCs is with Linear Feedback Shift Registers (LFSR) which are built from flip-flops and XOR gates, as shown in *Figure 1*. CRC−32 is similar. The superscripts over the flip-flops indicate the term in the CRC polynomial. For example, the 16 superscript indicates the flip-flop representing x^{16}. Be-

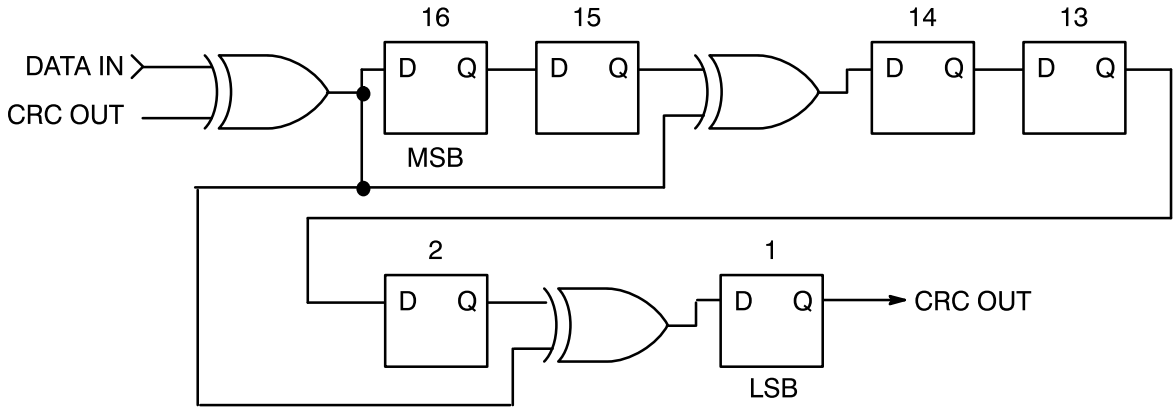

Figure 1. Linear Feedback Implementation of CRC−16

tween flip-flop 2 and 13 are nothing but flip-flops (no XOR gates).

Development of Parallel Algorithm

Following is a description of the algorithm and the results for calculating CRC−16 eight bits at a time (on a byte-by-byte basis). The procedure is similar for the CRC−32, except the results are calculated 16 bits at a time (on a word-by-word basis). The results for CRC−32 will be presented, but the actual calculations will not. This will be left as the proverbial "exercise for the reader". First, a few notes:

- R_i is the *i*th bit of the CRC register.
- C_i is the *i*th bit of the *initial* CRC register, before any shifts have taken place.
- Bit 1 is the least significant bit (LSB).
- The entries under each CRC register bit indicate the values to be XORed to generate the contents of that particular CRC register.
- A substitution will be made after eight shifts: $X_i = D_i$ XOR C_i

The results of the CRC are being calculated one bit at a time and the resulting equations for each bit are being examined. The CRC register after zero shifts is shown in *Table 1*. The CRC register after one 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:

Table 1. CRC−16 Register after Zero 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 C1 D1 D2	C1 D1	C16 C2 C1 D1 D2	C15 C1 D1	C13	C12	C11	C10	C9	C8	C7	C6	C5	C4	C3	C3 C2 C1 D1 D2

Table 4. CRC−16 Register after Eight Shifts with X_i Substitution

R16	R15	R14	R13	R12	R11	R10	R9	R8	R7	R6	R5	R4	R3	R2	R1
0 X8 X7 X6 X5 X4 X3 X2 X1	0 X7 X6 X5 X4 X3 X2 X1	0 X8 X7	0 X7 X6	0 X6 X5	0 X5 X4	0 X4 X3	0 X3 X2	C16 X2 X1	C15 X1	C14	C13	C12	C11	C10	C9 X8 X7 X6 X5 X4 X3 X2 X1

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

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

- the least-significant byte (bits 1..8) of the CRC register is dependent on 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.
- the most-significant byte (bits 9..16) of the CRC register is only dependent on XOR combinations of the initial low-order byte of the CRC register and the input byte. Note that zeros have been placed into the CRC register making use of the identity property.

A conclusion can be made that it is possible to shift the CRC register high-order byte into the low-order byte, throw away the low-order byte and XOR the CRC register with some 16-bit word to generate the new word. For example, calculating a new value for R9 is accomplished by calculating X3 and X2 and exclusive-ORing them together.

Description of CRC−16 Parallel Algorithm

Finally, the parallel algorithm for CRC−16 can be stated as follows:

1. Calculate the X_is by XORing the input byte with the least-significant byte of the CRC register.
2. Shift the CRC register eight bits to the right.
3. Use *Table 4* to calculate the 16-bit word to be XORed with the CRC register.
4. XOR the CRC register with the 16-bit word just calculated.

Repeat these four steps for every data byte in the data stream.

Implementation Issues for CRC−16 Parallel Algorithm

The key implementation issue for this algorithm is the calculation of the 16-bit register value defined in *Table 4*. Since the word is only dependent on eight inputs, it can be calculated and stored in a 256 x 8

EPROM look-up table. A 33-MHz maximum byte rate would dictate a total cycle time of 30 ns or less, which would then dictate a look-up table speed of significantly less than 30 ns. Such EPROMs, like the Cypress CY7C291A with a 15-ns access time, do exist.

Another approach is to calculate the XOR functions directly in logic, as one would do with a Field Programmable Gate Array (FPGA). Some quick calculations are in order. After the X_i values are calculated, one can see from *Table 4* that the largest XOR to be calculated is that for R1, which contains nine terms. A quick calculation of logic levels required in an FPGA can be made if the widest single-level XOR is known. The Cypress pASIC380 FPGAs can calculate a three-input XOR in a single logic cell. It can be shown that two levels of logic requiring a total of four logic cells would be required to calculate a nine-input XOR. Because a register would be feeding this XOR tree, and a register would be at the output of this XOR tree (as defined by the algorithm), the timing would be determined by internal delays in the device. The CRC−16 parallel algorithm can be implemented using *Warp3*™ (Cypress's VHDL-based FPGA design tool) in a CY7C384A FPGA. The design will run at the 33-MHz maximum parallel data rate supported by HOTLink. Design files can be found on the Cypress BBS.

Description of CRC−32 Parallel Algorithm

The parallel algorithm for CRC−32 is derived in the same manner as CRC−16; the only difference is the polynomial being used and the fact that data is now handled 16 bits (1 word) at a time, instead of 8 bits at a time. The CRC−16 algorithm can be modified into the CRC−32 and can be stated as follows:

1. Calculate the X_is by XORing the input byte with the least-significant word of the CRC register.
2. Shift the CRC register 16 bits to the right.
3. Use *Tables 5* and *6* to calculate the 32-bit word to be XORed with the CRC register.
4. XOR the CRC register with the 32-bit word just calculated.

Repeat these four steps for every data word in the data stream.

Table 5 contains the XOR information for the least-significant word (LSW) of the CRC−32 register after 16 shifts, and *Table 6* contains the XOR information for the most-significant word (MSW) of the CRC−32 register after 16 shifts. Again, note that the MSW only depends on XOR combinations of the initial lower-order bits of the CRC−32 register and input word. The LSW depends on XOR combinations of the initial lower-order bits of the CRC−32 register, the input word, and the initial MSW 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	C31	C30	C29	C28	C27	C26	C25	C24	C23	C22	C21	C20	C19	C18	C17
X1	X1	X2	X1	X1	X2	X1	X1	X2	X1	X1	X1	X4	X3	X2	X1
X2	X3	X3	X2	X3	X3	X2	X3	X3	X2	X2	X5	X10	X9	X8	X7
X4	X4	X5	X4	X4	X5	X4	X4	X4	X3	X6	X11	X11	X10	X9	X8
X5	X6	X6	X5	X6	X6	X5	X5	X8	X7	X12	X12	X14	X13	X12	X11
X7	X7	X8	X7	X7	X7	X6	X9	X14	X13	X13	X15				
X8	X9	X9	X8	X8	X11	X10	X15	X15	X14	X16					
X10	X10	X10	X9	X12		X16	X16								
X11	X11	X14	X13												
X12	X15														
X16															

Table 6. CRC−32 Register (MSW) after 16 Shifts with X_i Substitution

R32	R31	R30	R29	R28	R27	R26	R25	R24	R23	R22	R21	R20	R19	R18	R17
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
X6	X5	X4	X3	X2	X1	X1	X3	X2	X1	X6	X5	X4	X3	X2	X1
X7	X6	X5	X4	X3	X2	X4	X9	X8	X7	X7	X6	X5	X4	X3	X2
X10	X9	X8	X7	X6	X5	X10	X10	X9	X8	X8	X7	X6	X5	X4	X3
X16	X15	X14	X13	X12	X11	X11	X11	X10	X9	X10	X9	X8	X7	X6	X5
	X16	X15	X14	X13	X12	X12	X13	X12	X11	X11	X10	X9	X10	X7	X6
		X16	X15	X14	X13	X14	X14	X13	X12	X13	X12	X11	X11	X9	X8
				X16	X15	X15	X16	X15	X14	X14	X13	X12	X13	X10	X9
					X16			X16	X15	X16	X15	X14	X14	X12	X11
											X16	X15	X15	X13	X12
												X16		X14	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 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 33 Mbytes per second, and we are calculating the CRC on two bytes at a time. This gives us approximately 2 clock cycles (60 ns) to perform the calculation. Using an EPROM look-up table requires an extra component, compared with implementing the entire design in an FPGA. So, let's look at the critical path for a pASIC380-based design.

Looking at *Tables 5* and *6* shows that the largest XOR is an 11-term function feeding CRC register R16. Again, given single-logic cell capabilities of a three-input XOR the longest calculation now requires three levels of logic cells. This is one more than required for the CRC−16 implementation, but remember, we have twice as much time to calculate the CRC−32.

As data rates increase, so will timing constraints, but for the data rates supported by the HOTLink devices (160 to 330 MBits/sec), the pASIC380 devices can successfully implement parallel calculation of CRC−32. A conservative estimate of the critical path delay is determined in the discussion that follows. The actual design files can be found on the Cypress BBS.

Figure 2 is a graphical representation of the logic delays in the XOR-tree portion of the CRC−32 design.

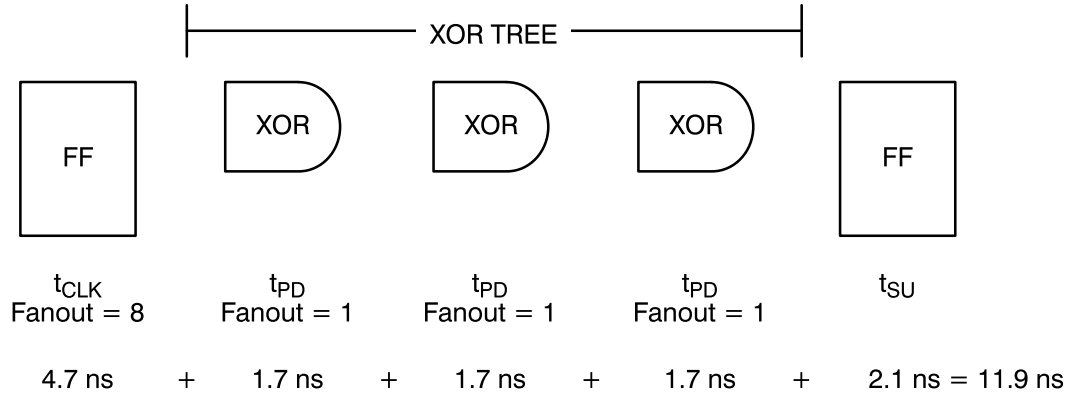

Figure 2. Critical Path Logic Delay Estimate for CRC−32

Note that routing delays are not included in this figure. Let's create a more realistic estimate:

1. round 11.9 ns to **12 ns**
2. add 50% of the logic delays for routing delays 12 ns + 0.5*12 ns = **18 ns**
3. find worst-case commercial timing by multiplying by the worst-case K factor for the −1 speed bin (18 ns x 1.33 = **24 ns**).

This is comfortably under the 60 ns figure mentioned earlier. All delay values, except for the 50% ratio mentioned above, are from the CY7C384A−1JC device datasheet.

Conclusion

This note has shown how to calculate a parallel implementation of any CRC polynomial and results for CRC−16 and CRC−32 were provided. Implementation issues were also discussed and performance data from implementing the byte-wise CRC−16 algorithm in a CY7C384A−1JC was presented. It easily ran at 33 MHz, the fastest parallel data rate supported by Cypress's HOTLink devices.

Estimated results for word-wise CRC−32 calculation in the CY7C384 were very close to the CRC−16 results reported. Since the speed requirements are halved, actual margins are greater than for the CRC−16.

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.

Document #: COM0009−0394