SINGLE BIT ERROR CORRECTION IMPLEMENTATION IN CRC-16 ON FPGA

Sunil Shukla, Neil W. Bergmann
School of ITEE, The University of Queensland, Australia
{Sunil, bergmann}@itee.uq.edu.au

Abstract

Framing protocols employ cyclic redundancy check (CRC) to detect errors incurred during transmission. Generally whole frame is protected using CRC and upon detection of error, re-transmission is requested. But certain protocols demand for single bit error correction capabilities for the header part of the frame, which often plays an important role in receiver synchronization. At a speed of 10 Gbps, header error correction implementation in hardware can be a bottleneck. This paper presents a hardware efficient way of implementing CRC-16 over 16 bits of data, multiple bit error detection and single bit error correction on FPGA device.

1. Introduction

The Internet is growing rapidly in terms of number of users and amount of bandwidth used. Besides the transmission and switching speeds, the per-packet operations necessary for Internet Protocol (IP) packet forwarding are the current limiting factors. As transmission speeds are continually increasing, IP packet processing overheads have become the main bottleneck [5]. Often, IP packets are encapsulated in frames protected by a cyclic redundancy check (CRC) code. CRC is the most preferred method of encoding because it provides very efficient protection against commonly occurring burst errors. CRC's can detect all one bit and two bit errors as well as all odd number of bits in error [2]. The most commonly used framing techniques are PPP, HDLC and GFP. Generic Framing Procedure (GFP) is a recently proposed technique for framing. The advantage of this technique is that it does not use any special code to indicate the beginning and end of frame. Frame delineation is based on packet length that is transmitted at the beginning of each frame. The 16 bit packet length is protected by CRC-16 and transmitted as core header. Single bit error correction capability is required from the receiver. Besides the packet length, GFP frame also has type header following core header which is also protected by CRC-16 and the receiver is expected to correct single bit error for type header also. This can be a bottleneck at a speed of 10Gbps, if the core is implemented on FPGA. A lot of work has been reported on hardware implementation of error detection using CRC but there is no published method for error correction in CRC in hardware. In this paper we have proposed a technique for CRC-16 error detection and single bit error correction which is hardware optimized and works at relatively higher frequency. This paper focuses on implementation of this method on FPGA. We are targeting FPGA because timing issues in FPGA arises more often and this technique utilizes the huge resources available in FPGA as Block RAM. Focus of the paper is on single bit error correction for header bits protected by CRC-16. The paper is organized into two parts viz.: CRC-16 implementation in hardware and the proposed technique for CRC error detection and single bit error correction.

2. CRC-16 Implementation in Hardware

The generator polynomial used for CRC-16 calculation is $X^{16} + X^{12} + X^{5} + 1$ in X25 standard and $X^{16} + X^{15} + X^{2} + 1$ in CCITT. In this paper, we will be referring to the polynomial defined in X25 standard but the results can be extended to any 16 bit generator polynomial with slight modification. CRC can be implemented in hardware via techniques such as serial implementation, parallel implementation or look up table-based implementation. Look up table approach involves storing CRC values for all possible input combinations. Thus for 16 bit input data, we need to store 2^{16} (65536) values i.e. a storage space of 1M bits. Serial implementation uses Linear Feedback Shift Registers (LFSR) in hardware. In LFSR, division is performed by left shifting and subtraction by XOR operation. Serial implementation is hardware efficient but is not feasible at higher frequencies. In case of parallel implementation, the division process is reducible to a set of equations involving XOR operation. Parallel implementation of CRC is fast because it involves two level of logic. The optimized equation of resulting checksum in CRC-16 is summarized in Fig. 1.

$$
\begin{aligned}
C(15) &= E(11) \oplus E(10) \oplus E(7) \oplus E(3) \\
C(14) &= E(10) \oplus E(9) \oplus E(6) \oplus E(2) \\
C(13) &= E(9) \oplus E(8) \oplus E(5) \oplus E(1) \\
C(12) &= E(15) \oplus E(8) \oplus E(7) \oplus E(4) \oplus E(0) \\
C(11) &= E(15) \oplus E(14) \oplus E(11) \oplus E(10) \oplus E(6) \\
C(10) &= E(14) \oplus E(13) \oplus E(10) \oplus E(9) \oplus E(5) \\
C(9) &= E(15) \oplus E(13) \oplus E(12) \oplus E(9) \\
&\quad \oplus E(8) \oplus E(4) \\
C(8) &= E(15) \oplus E(14) \oplus E(12) \oplus E(11) \\
&\quad \oplus E(8) \oplus E(7) \oplus E(3) \\
C(7) &= E(15) \oplus E(14) \oplus E(13) \oplus E(11) \\
&\quad \oplus E(10) \oplus E(7) \oplus E(6) \oplus E(2) \\
C(6) &= E(14) \oplus E(13) \oplus E(12) \oplus E(10) \\
&\quad \oplus E(9) \oplus E(6) \oplus E(5) \oplus E(1) \\
C(5) &= E(13) \oplus E(12) \oplus E(11) \oplus E(9) \\
&\quad \oplus E(8) \oplus E(5) \oplus E(4) \oplus E(0) \\
C(4) &= E(15) \oplus E(12) \oplus E(8) \oplus E(4) \\
C(3) &= E(15) \oplus E(14) \oplus E(11) \oplus E(7) \oplus E(3) \\
C(2) &= E(14) \oplus E(13) \oplus E(10) \oplus E(6) \oplus E(2) \\
C(1) &= E(13) \oplus E(12) \oplus E(9) \oplus E(5) \oplus E(1) \\
C(0) &= E(12) \oplus E(11) \oplus E(8) \oplus E(4) \oplus E(0)
\end{aligned}
$$

Figure 1 CRC-16 Equations

Where,
$E(i) = D(i)$ XOR $C_{prev}(i)$,
'$\oplus$' indicates XOR operator,
$D(i)$ is the i^{th} bit of input data,
$C_{prev}(i)$ is the i^{th} bit of previous CRC result. In our case, since data width is 16 bits, C_{prev} refers to the initial state of the CRC which may be either all zeros or all ones.

3. Proposed method for CRC-16 Error Detection and Correction

In this paper, we will be presenting a unique way of implementing multiple bit error detection and single bit error correction using CRC for a data width of 16 bits. Let F_{tr} be the frame transmitted in which checksum is appended after 16 bits of data. We can express F_{tr} as
$F_{tr} = D_{tr}$ & C_{tr}

Where,
& – Concatenation operator
D_{tr} – transmitted 16 bit data
C_{tr} – transmitted 16 bit checksum
At the receiver side, let F_{re} be the received frame such that
$F_{re} = D_{re}$ & C_{re}
Where, C_{re} indicates received checksum and D_{re} represents received data. Receiver again calculates CRC on the received data. Let C_{cal} indicates the CRC calculated over D_{re} at the receiver side. If no error has occurred during transmission then C_{re} and C_{cal} are equal. But if some bit(s) are in error, then C_{re} and C_{cal} will be in mismatch. Here we are concerned with just single bit error. There can be two cases, either single bit error can be in data, D_{re} or it can be in checksum, C_{re}. In case single bit error is there in one of the checksum bits, then we need to just detect it. So the real concern is to correct data in case one of the data bit is in error. If we refer to Fig 1, we will see that the checksum calculation involves XOR operations on a combination of data bits. If single bit of data flips then all the checksum bits in which that data bit has been used, will be inverted. For eg. Data bit 0 is used in checksum bit 0, 5 and 12. So if there is an error in data bit 0, then the calculated checksum and received checksum will differ in position 0, 5 and 12. Let $C_{xorpattern} = C_{cal}$ XOR C_{tr}. If we consider that only one bit in data is in error then we will have 16 unique patterns for $C_{xorpattern}$, each corresponding to individual data error bits. We have written a C program, and found the patterns for the XOR result.

Table 1. XOR pattern for data bit in error

Data Bit in Error	$C_{xorpattern}$	MSB 8 bits	LSB 8 bits
0	0001000000100001	16	33
1	0010000001000010	32	66
2	0100000010000100	64	132
3	1000000100001000	129	8
4	0001001000110001	18	49
5	0010010001100010	36	98
6	0100100011000100	72	196
7	1001000110001000	145	136
8	0011001100110001	51	49
9	0110011001100010	102	98
10	1100110011000100	204	196
11	1000100110101001	137	169
12	0000001101110011	3	115
13	0000011011100110	6	230
14	0000110111001100	13	204
15	0001101110011000	27	152

If there is single bit error in checksum bit, then we will obtain the following XOR patterns.

Table 2. XOR pattern for checksum bit in error

CRC Bit in Error	$C_{xorpattern}$	MSB 8 bits	LSB 8 bits
0	0000000000000001	0	1
1	0000000000000010	0	2
2	0000000000000100	0	4
3	0000000000001000	0	8
4	0000000000010000	0	16
5	0000000000100000	0	32
6	0000000001000000	0	64
7	0000000010000000	0	128
8	0000000100000000	1	0
9	0000001000000000	2	0
10	0000010000000000	4	0
11	0000100000000000	8	0
12	0001000000000000	16	0
13	0010000000000000	32	0
14	0100000000000000	64	0
15	1000000000000000	128	0

The XOR patterns are unique for single bit error occurring anywhere, either in data or in checksum. For correction purpose, we have to just find out the bit in error. If that bit is CRC bit we need not do anything but if that bit is data bit then we need to flip that bit. We can find the bit depending upon which we can have a bit sequence with which received data is XORed. For e.g. if bit two is in error, then the bit sequence is "0000000000000100". This pattern is XORed with received data, which is simply flipping of bit two. We have stored these bit sequences in memory.

4. Memory Design Considerations

In this section, memory design parameters and programming is discussed in detail. FPGA have abundant memory available in the form of Block RAM. We will show in this section that one block RAM is sufficient for whole processing. In fact two ports of single Block RAM can serve the purpose of two CRC correction engines simultaneously as the Block RAM is used as ROM with the configuration parameters initialized during its generation using Xilinx CORE Generator.

4.1. Memory Addressing

The memory is accessed using the XOR patterns. Each 16 bit XOR pattern is unique among the 32 cases. But using 16 bits of addressing implies 65536 locations, which is not desirable. To minimize the number of locations required, XOR pattern is divided into two parts of 8 bits each. If we observe the XOR patterns in Table 1, the lower 8 bits have a repetitive pattern hence upper 8 bits of XOR pattern is used for accessing memory, as there is no repetitive value. If we observe the XOR patterns in Table 2, for half of the cases, upper 8 bits are zero and for the remaining cases lower 8 bits are zero. Thus we can't use the upper 8 bits for addressing memory for the first 8 cases. In those cases, lower 8 bits are used for addressing memory. Thus there is a muxing of address. Whenever the upper 8 bits of XOR pattern are zero, lower 8 bits are used for accessing memory. In this way a maximum of 256 locations are required. If we observe the MSB 8 bit patterns for data and CRC, then we will find that all the 32 patterns are not unique. There are overlapping patterns of 16, 32 and 64. These patterns will come as address bits when there is single bit error in data at bit position 0, 1 and 2 respectively and will also appear when there is single bit error in CRC at bit position 5, 6, 7 respectively and again at 12, 13 and 14 respectively. These cases can be distinguished easily. If the lower or upper 8 bits of XOR pattern are all zeros, then it is a probable case of single bit error in CRC else it is a probable case of single bit error in data. We have protection for such overlapping cases in our data structure.

4.2. Memory Data Structure

The data structure for memory is shown in **Fig 2.**

No Error (1)	CRC bit error (1)	Data bit error (1)	XORing bit seq. (16)	Match Pattern (8)

Figure 2 Memory Data Structure

The lower 8 bits of XOR pattern are stored in memory location as "Match Pattern" for addresses 3, 6, 13, 16, 18, 27, 32, 36, 51, 64, 72, 102, 129, 137, 145 and 204. For address 1, 2, 4, 8 and 128 which represent 8 bit XOR pattern in case of CRC bit in error excluding 16, 32 and 64, "Match pattern" is all zeros. The 16 bit "XORing bit sequence" is stored in which only one bit that corresponds to the bit position in error is set to '1' and all other bits are set to '0'. For e.g. for location 72, which indicates a probable case of data bit error at bit position 6, match pattern will be "11000100" and XORing bit sequence will be "0000000000100000". Two additional bits, "CRC bit error" and "Data bit error", are used to indicate that the information is stored is related to single bit error in data or CRC. For locations 3, 6, 13, 16, 18, 27, 32, 36, 51, 64, 72, 102, 129, 137, 145 and 204 "Data bit error" is set to '1' and at all other locations it is set to '0'. For locations 1, 2, 4, 8, 16, 32, 64 and 128 "CRC bit error" is set to '1' and at all other locations it is set to '0'. At location 0, "No error" bit is set to '1' and at all other locations it is set to '0'. Thus last three bits in memory are used to decide the type of error. The data width of memory will be 26 bits.

5. Error Handling

In this section we will discuss how single and multi bit errors are detected and handled. The decision is based upon following algorithm.

```
BEGIN
 Reset flag no_error
 Reset flag single_bit_error
 Reset flag single_bit_crc_error
 Reset flag single_bit_data_error
 Reset flag multiple_bit_error

 IF bit_26 = '1' THEN

     Set flag no_error
     Transmit received data

 ELSIF bit_25 = '1' AND bit_24 = '0' THEN
  IF Cxorpattern (7:0) = "00000000" THEN

     Set flag single_bit_error
     Set flag single_bit_crc_error
     Transmit received data
  ELSE

     Set flag multiple_bit_error
     Transmit received data
  END IF

 ELSIF bit_25 = '0' AND bit_24 = '1' THEN
  IF Cxorpattern (7:0) = "Match Pattern" THEN

     Set flag single_bit_error
     Set flag single_bit_data_error
     XOR received data with 16 bit "XORing bit
     sequence"
  ELSE

     Set flag multiple_bit_error
     Transmit received data
  END IF

 ELSIF bit_25 = '1' AND bit_24 = '1' THEN
  IF Cxorpattern (7:0) = "00000000" THEN

     Set flag single_bit_error
     Set flag single_bit_crc_error
     Transmit received data
  ELSIF Cxorpattern (7:0) = "Match Pattern" THEN

     Set flag single_bit_error
     Set flag single_bit_data_error
     XOR received data with 16 bit "XORing bit
     sequence"
  ELSE

     Set multiple_bit_error
     Transmit received data
  END IF
 ELSE
  Set multiple_bit_error
  Transmit received data
 END IF
END
```

6. Hardware Implementation

The algorithm has been implemented and verified on Xilinx Virtex-II FPGA device. The code was written in VHDL and synthesized using Leonardo Spectrum. The device used for implementation is 2V40cs144 with speed grade 5 and wire load model xcv2-40-5_wc. The results obtained are summarized in **Table 3**.

Table 3 Hardware Implementation Results

Device	Area	Speed
Virtex II (2V40cs144)	52 slices	233 MHz

For a GFP IP core giving a throughput of 10Gbps, there can be 9 such CRC single bit error correction engines in receiver, assuming a data interface of 64 bits. Hence the effect of area saving in the design of this block is multiplied by 9.

7. Conclusion

In this paper, we have described how single bit error correction can be employed in case of CRC-16 in a very efficient way on FPGA. This approach is efficient both in terms of hardware and speed. The additional hardware required is very simple. This technique works efficiently in case of ASIC design also.

8. References

[1] Ross N. Williams, "Painless Guide to CRC Error Detection Algorithms".
[2] Norman Matloff, "Cyclic Redundancy Checking".
[3] Adrian Simionescu, "Computing CRC in Parallel for Ethernet".
[4] Giuseppe Campobello, Giuseppe Patane, Marco Russo, "Parallel CRC Realization".
[5] Florian Braun, Mmarcel Waldvogel, "Fast Incremental CRC Updates for IP over ATM Networks".