CRC Algorithm Implementation in FPGA by Xmodem protocol

T.V.A. Bhanuprakash[1] , K. Kameswar reddy[2] S. Mahaboob Basha [3]
[1] M.Tech student, AVR &SVR Engg College, Kurnool
[2] Assistant Professor, Dept of ECE, AVR &SVR Engg College, Kurnool
[3] Associate Professor and HOD, Dept of ECE, AVR &SVR Engg College, Kurnool

Abstract: - Xmodem protocol is a widely used asynchronous file transfer protocol. By using Xmodem protocol we can implement the CRC algorithm with 16-bit data. Here we studied the implementation of CRC algorithm by LFSR using FPGA. This paper studies of implementation of CRC algorithm of 16-bit and 32-bit by single -byte and multi-byte parallel circuit. We can achieve the synthesis design using Verilog language.

Keywords: Xmodem , CRC , Checksum , packet, LFSR

I. INTRODUCTION

XMODEM, like most file transfer protocols, breaks up the original data into a series of "packets" that are sent to the receiver, along with additional information allowing the receiver to determine whether that packet was correctly received. Xmodem is divided as Xmodem and 1k-Xmodem, Xmodem transmits the data as 128-bytes block form and 1k-Xmodem transmits as1024 bytes block form. These Xmodem supports the CRC and Checksum verification methods. The Xmodem CRC checksum requires the 128 bytes for the data packets. When the packet is received by the receiver, it sends conformation character if checksum is correct. It send the negative conformation character if error occurs. The Xmodem protocol of data is depends on efficiency of transmitted data check time. The Cyclic redundancy Check code (abbrivated as "CRC") is used to check the errors in the data. It is applicable to verify the data in Xmodem protocol, RFID protocol, USB communication protocol etc.

XMODEM-1K was an expanded version of XMODEM-CRC, which indicated the longer block size in the *sender* by starting a packet with the <STX> character instead of <SOH>. Like other backward-compatible XMODEM extensions, it was intended that a -1K transfer could be started with any implementation of XMODEM on the other end, backing off features as required.

XMODEM-1K was originally one of the many improvements to XMODEM introduced by

Chuck Forsberg in his YMODEM protocol. Forsberg suggested that the various improvements were optional, expecting software authors to implement as many of them as possible. Instead they generally implemented the bare minimum, leading to a profusion of semi-compatible implementations, and eventually, the splitting out of the name "YMODEM" into "XMODEM-1K" and a variety of YMODEMs. Thus XMODEM-1K actually post-dates YMODEM, but remained fairly common anyway.

A backwards compatible extensions of XMODEM with 32k and 64k block lengths was created by Adontec for better performance on high-speed error free connections like ISDN or TCP/IP networks.

CRC Xmodem is very similar to Checksum Xmodem. The protocol initiation has changed and the 8 bit checksum has been replaced by a 16 bit CRC. Only theses changes are presented.

One of the earliest and most persistent problems with Xmodem was transmission errors which were not caught by the checksum algorithm. Assuming that there is no bias in asynchronous communications errors, we would expect that 1 out of every 256 erroneous complete or oversized Xmodem packets to have a valid checksum. With the same assumption, if the checksum were 16 bits, we would expect 1 out of every 65,536 erroneous complete or oversized packets would have a valid checksum.

Considerable theoretical research has shown that a 16 bit cyclical redundancy check character (CRC/16) will detect a much higher percent of errors such that it would only allow 1 undetected bit in error for every 10^{14} bits transmitted. That's 1 undetected error per 30 years of constant transmission at 1 megabit per second. However, my personal experience indicates that something around 10^9 to 10^{10} is more realistic. Why such a vast improvement over the checksum algorithm? It is caused by the unique properties that prime numbers have when being divided into integers. Simply stated, if an integer is divided by a prime number, the remainder is unique. The CRC/16 algorithm treats all 1024 data bits in an Xmodem packet as an integer, multiples that integer by 2^16 and then divides that 1040 bit number by a 17 bit prime number. The low order 16 bits of the remainder becomes the 16 bit CRC.

The 17 bit prime number in CRC Xmodem is $2^{16} + 2^{12} + 2^5 + 1$ or $65536 + 4096 + 32 + 1 = 69665$. So calculating the CRC is simple, just multiply the 128 byte data number by 65536, divide by 69665 and the low order 16 bits of the remainder are the CRC. The only problem is, I've never seen a computer which has instructions to support 130 byte integer arithmetic! Fortunately for us, Seephan Satchell, Satchell Evaluations, published a specification a very efficient algorithm to calculate the CRC without either 130 byte arithmetic or bit manipulation. Appendix A contains the source code, in IBM/PC BASIC, for the calculation of a CRC. The initiation of CRC Xmodem was designed to provide for automatic fall back to Checksum Xmodem if the transmitter does not support the CRC version. The receiver requests CRC Xmodem by sending the letter C (decimal 67) instead of a NAK. If the transmitter supports CRC Xmodem, it will begin transmission of the first Xmodem packet upon receipt of the C. If the transmitter does not support CRC Xmodem, it will ignore the C. The receiver should timeout after 3 seconds and repeat sending the C. After 3 timeouts, the receiver should fall back to the checksum Xmodem protocol and send a NAK.

II ALGORITHM PRINCIPLES OF CRC:

The following is the specify CRC implementation principle:

The k-bit binary number is about using M(X) is

$M(X) = C_{k-1} X^{k-1} + C_{k-2} X^{k-2} + \ldots + C_i X^i + \ldots + C_1 X + C_0$ …… (1)

Let G (X) = Generated polynomial, r = highest power of sequence data then

$M(X) * x^r = C_{k-1} X^{k+r-1} + C_{k-2} X^{k+r-2} + \ldots + C_1 X^{k-1} + C_0 X^k$ ……… (2)

The result is divided by generated polynomial G(x)

$(M(X)..X^r) / G(x) = Q(X) + R(X)) / G(X)$ …(3)

Here R(X) is remainder and it is CRC.

R(X) is expressed as

$R(X) = d_{r-1}. x^{r-1} + \ldots + d_1 x^1 + r_0$ …………(.4)

The final data is transmitted as

$M^1 = (C_{K-1,\ldots\ldots\ldots} C_{0,} d_{r-1}, \ldots. d_{1,} d_0)$ …… (5)

Figure 1 is a typical circuit, if the generator polynomial G (x) is 1, the output of the D flip-flop connects to the output of the XOR gate; if it is 0, and the output of the D flip-flop connects to the output of the superior flip-flop. Therefore, this figure can be greatly simplified in the case of a fixed generator polynomial.

There are some standard generator polynomials in the practical applications, as follows:

CRC 8 : $X^8 + X^5 + X^4 + 1$ with number 0x131;
CRC 12: $X^{12} + X^{11} + X^3 + X^2 + 1$
with number 0x180D;
CRC 16: $X^{16} + X^{12} + X^5 + 1$ with number 0x11021;
ANSI CRC 16: $X^{16} + X^{15} + X^2 + 1$
with number 0x18005;
CRC 32: $X^{32} + X^{26} + X^{23} + X^{22} + X^{16} + X^{12} + X^{11} + X^{10} + X^8 + X^7 + X^5 + X^4 + X^2 + X^1 + 1$,
with number 0x104C11DB7.

To develop a hardware circuit for computing the CRC checksum, we reduce the polynomial division process to its essentials. The process employs a shift register, which we denote by CRC. This is of length r (the degree of G) bits, not as you might expect. When the subtractions (exclusive or's) are done, it is not necessary to represent the high-order bit, because the high-order bits of G and the quantity it is being subtracted from are both 1. The division process might be described informally as follows:

Initialize the CRC register to all 0-bits. Otherwise, just shift CRC and m left 1 position. If there are more message bits, go back to get the next one. It might seem that the subtraction should be done first, and then the shift. It would be done that way if the CRC register held the entire generator polynomials, which in bit form are bits. Instead, the CRC register holds only the low-order r bits of G, so the shift is done first, to align things properly.

Get first/next message bit m. If the high-order bit of CRC is 1, Shift CRC and m together left 1 position, and XOR the result with the low-order r bits of G.

Below is shown the contents of the CRC register for the generator $G = X^3 + X + 1$ and the message

$M = X^7 + X^6 + X^5 + X^2 + X$

Expressed in binary, G = 1011 and M = 11100110.

000 Initial CRC contents. High-order bit is 0, so just shift in first message bit.
001 High-order bit is 0, so just shift in second message bit, giving:

011 High-order bit is 0 again, so just shift in third message bit, giving:
111 High-order bit is 1, so shift and then XOR with 011, giving:
101 High-order bit is 1, so shift and then XOR with 011, giving:
001 High-order bit is 0, so just shift in fifth message bit, giving:
011 High-order bit is 0, so just shift in sixth message bit, giving:
111 High-order bit is 1, so shift and then XOR with 011, giving:
101 There are no more message bits, so this is the remainder.

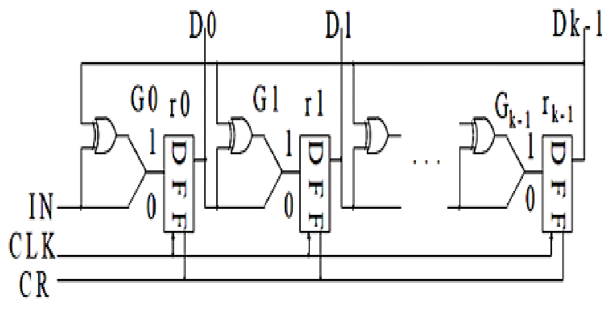

Figure 1 Linear feedback shift register LFSR

In the Xmodem protocol, we use the standard CCITT (Consultative Committee for International Telephony and Telegraphy) method, CRC16: $X^{16} + X^{12} + X^{5} + 1$

Implementation of the CRC hardware circuit:

1) We first clear the flip-flop by CR, and move the upper 16 bits (2 bytes) which need to be verified into 16 of the trigger. The upper 16 bits of the data stream will not be changed as the trigger has been cleared;

2) Then we continue flow the data into the trigger, it just need right 1-bit if the output of the r15 trigger is 0; and it need right 1-bit after the modulo 2 operation if the output is 1;

3) At last we need continuous move into 16-bit 0 after the data stream M (x) all into the trigger, and end the calculation of this group data CRC

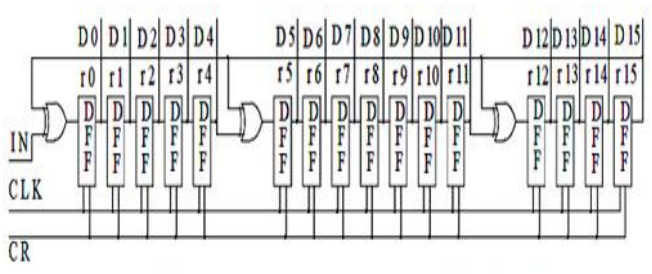
D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 D10 D11 D12 D13 D14 D15
r0 r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 r11 r12 r13 r14 r15
IN
CLK
CR
DFF
Figure 2 The circuit implementation of CCITT CRC16 LFSR

Figure 2 the process of the circuit is summarized as follows:

The above circuit using the general process of the modulo 2 division operation, its biggest drawback is need continuous input 16 "0" after the data stream M (x) and the CRC of the trigger need to more 16 times calculation.

III. THE IMPLEMENTATION OF CRC PARALLEL COMPUTING

In the Xmodem protocol, each packet is 128 bytes (1024 bits). We need 1040 (1024 + 6) cycles to figure out the CRC using the Figure 2. This design uses a parallel computing and hardware implementations in order to improve the real-time.

We main narrative the 8-bit CRC parallel computing. The state of the flip-flop is the remainder of the CRC as shown in Figure 1. The remainder of the CRC is just concerned with the former input and the remainder of the previous state when the serial operation. The calculation of 8-bit parallel operation as follows:

Supposed rji as the value of the trigger, i = 1, 2... n, as the input code sequence, j = 0, 1, ..., k-1, as the trigger coding,

$$r_j^i = G_j \cdot (r_{k-1}^{i-1} \oplus r_{j-1}^{i-1}) + \overline{G_j} \cdot r_{j-1}^{i-1} \quad \text{.........} \quad (6)$$

The input data is 8-bit, so the maximum of i is 8. We can transitive launch r_0^8, ..., r_{15}^8 by CCITT CRC16 (the polynomial is G(X) = $X^{16} + X^{12} + X^2 + 1$, that is, k = 16) and the equation (6).

We need 24-clock to calculate the CRC of the 8-bit data. In the first 16-clock, we move the 8-bit data into the high trigger and the low 8-bit is zeros. And this is the initial moment, we can get the CRC of the 8-bit data after 8 clocks and the input data is zeros. Then the initial moment of the trigger values are:

$r_0^0 \sim r_7^0 = 0$, r_8^0 , r_9^0 , … , r_{14}^0 , r_{15}^0 = M0 , M1 , … , M6 , M7。

If r_j^8 is represent the value after 8 clocks, we substituted the r_j^8 into equation (6) and get the final expression of r_j^8

$$r_0^8 = r_{15}^7 = r_{13}^5 = r_{12}^4 = r_{15}^3 \oplus r_{11}^3 = r_{14}^2 \oplus r_{10}^2 = r_{12}^0 \oplus r_8^0 = M0 \oplus M4 \quad (7)$$

$$r_1^8 = r_0^7 = r_{13}^4 = r_{12}^3 = r_{11}^2 \oplus r_{15}^2 = r_9^0 \oplus r_{13}^0 = M1 \oplus M5 \quad (8)$$

We can derive from other similar items:

$$r_2^8 = r_{10}^0 \oplus r_{14}^0 = M2 \oplus M6,$$

$$r_3^8 = r_{11}^0 \oplus r_{15}^0 = M3 \oplus M7,$$

$$r_4^8 = r_{12}^0 = M4,$$

$$r_5^8 = r_8^0 \oplus r_{12}^0 \oplus r_{13}^0 = M0 \oplus M4 \oplus M5,$$

$$r_6^8 = r_9^0 \oplus r_{13}^0 \oplus r_{14}^0 = M1 \oplus M5 \oplus M6,$$

$$r_7^8 = r_{10}^0 \oplus r_{14}^0 \oplus r_{15}^0 = M2 \oplus M6 \oplus M7,$$

$$r_8^8 = r_0^0 \oplus r_{11}^0 \oplus r_{15}^0 = M3 \oplus M7,$$

$$r_9^8 = r_1^0 \oplus r_{12}^0 = M4,$$

$$r_{10}^8 = r_2^0 \oplus r_{13}^0 = M5,$$

$$r_{11}^8 = r_3^0 \oplus r_{14}^0 = M6,$$

$$r_{12}^8 = r_4^0 \oplus r_8^0 \oplus r_{12}^0 \oplus r_{15}^0 = M0 \oplus M4 \oplus M7,$$

$$r_{13}^8 = r_5^0 \oplus r_9^0 \oplus r_{13}^0 = M1 \oplus M5,$$

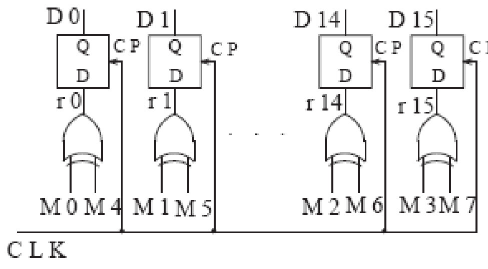

Fig3 8-bit parallel CCITT CRC16 hardware circuit

IV PACKETS

A method of transferring data by breaking it up into small chunks called packets. Packet data is how most data travels over the Internet, and, in recent years, over all cell phone networks as well. With packet-switched data, each user only consumes network resources when they are actually transferring data. This is often superior to circuit-switched data, where an open data connection must be maintained, which uses network resources even when idle. Packet-switched is the more modern type, and usually faster. In a mobile phone, data is used for functions involving the Internet, as well as most kinds of streaming video and audio.

There are many different types of packet data for mobile phones, with different maximum speeds. A packet is the unit of data that is routed between an origin and a destination on the Internet or any other packet-switched network.

When any file is sent from one place to another on the Internet, the Transmission Control Protocol (TCP) layer of TCP/IP divides the file into "chunks" of an efficient size for routing. Each of these packets is separately numbered and includes the Internet address of the destination. The individual packets for a given file may travel different routes through the Internet. When they have all arrived, they are reassembled into the original file

The packet was prefixed by a simple 3-byte header containing a `<SOH>` character, a "block number" from 0-255, and the "inverse" blocks number—255 minus the block number. Block numbering starts with 1 for the first block sent, not 0.

The packet was also suffixed with a single-byte checksum of the data bytes. The checksum was the sum of all bytes in the packet modulo 256. The modulo operation was easily computed by discarding all but the eight least significant bits of the result, or alternatively on an eight bit machine, ignoring arithmetic overflow which would produce the same effect automatically. In this way the checksum was restricted to an eight bit quantity which was able to be expressed using a single byte. For example, if this checksum method was used on a tiny data packet containing only two bytes carrying the values 130 and 130, the total of these codes is 260 and the resulting checksum is 4. The complete packet was thus 132 bytes long, containing 128 bytes of data, for a total channel efficiency of about 97%.

The file was marked "complete" with a <EOT> character sent after the last block. This character was not in a packet, but sent alone as a single byte. Since the file length was not sent as part of the protocol, the last packet was padded out with a "known character" that could be dropped. In the original specification this defaulted to <SUB> or 26 decimal, which CP/M used as the end-of-file marker inside its own disk format. The standard suggested any character could be used for padding, but there was no way for it to be changed *within the protocol* itself – if an implementation changed the padding character, only clients using the same implementation would correctly interpret the new padding character.

V. IMPLEMENTATION AND RESULTS

The proposed system is designed using Verilog hardware description language and structural form of coding. The basic blocks of scan based test structure are completely synthesized using Xilinx XST and implemented on device family Spartan 3e, device XC3S500E, and package FG320 with speed grade 4.

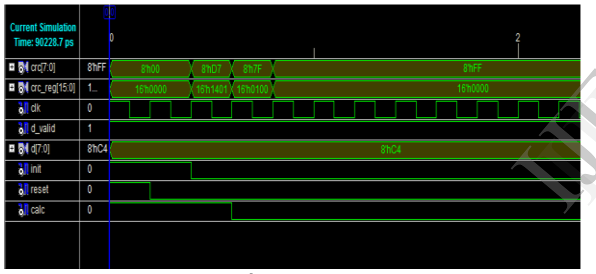

Fig 4 Simulation Results for CRC16

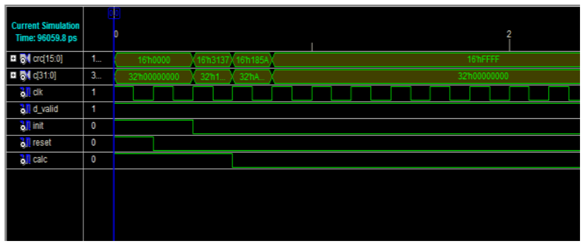

Fig 5 Simulation Results for CRC32

VI. CONCLUSION

This paper analysis the principle of the CRC calculation, redesign the per-byte parallel computing to the checksum of CCITT CRC16 and CRC 32, present a general method of parallel computing of CRC and the CRC algorithm solution of the data packet. We implement the Xmodem protocol with CRC check use FPGA based on the above method.

REFERENCES

[1] YU Xun The 32-bit cyclic redundancy check parallel algorithm and hardware implementation[J].Information Technology, 2007.

[2] LI You-zhong The generic parallel CRC calculation principle and its hardware implementation[J]. Northwest Minorities University (NaturalScience) 2002.

[3] LI You-mou, FANG Ding-yi. CRC coding algorithm research and Realization[J]. Journal of Northwest University(Natural Science Edition), 2006.

[4] JI Shang-man, LI Wei, SHEN Ke-jie, YAO Hui, TAO Zhi-jie. Improved CRC Arithmetic and Implementation by SCM[J]. Industrial Control Computer. 2009.

[5] ZHU Rong-hua. The Principle and Implementation of a Parallel CRC Computing[J]. Acta Electronica Sinica, 1997.

[6] ZHANG Shu-gang,ZHANG Sui-nan,HUANG Shi-tan. CRC Parallel Computation Implementation on FPGA[J]. Computer Technology And Development, 2007.