

Using CRC Hard Blocks in Virtex-5 FPGAs

CRC blocks provided as hard macros speed up the process of error detection.

by Sunita Jain

Associate Design Engineer
Xilinx India Technology Services Pvt. Ltd.
(Xilinx Hyderabad, XHD)
sunita.jain@xilinx.com

Guru Prasanna

Technical Lead
Xilinx India Technology Services Pvt. Ltd.
(Xilinx Hyderabad, XHD)
guru.prasanna@xilinx.com

Data corruption is the principal problem associated with data transmission and storage. Whenever data is transmitted over channels, there is always a finite probability that some errors will occur.

It is essential that the receiving module can differentiate between an error-free message and an erroneous one. There are various methods to detect errors; most error-checking methods do so by introducing redundant bits exclusively for this purpose. Commonly used methods for error detection in data communication include parity codes, Hamming codes, and cyclic redundancy check (CRC); of these, CRC is the most widely used.

CRC is computed on a given set of data bits and appended at the end of the data frame before transmission or storage. When the frame is received or retrieved, its validity is verified by recalculating the CRC for the contents of the frame to ensure that the data is error-free.

In this article, we'll take a quick look at the theory behind CRC calculation and its hardware implementation using linear feedback shift registers. Then we'll direct our attention to the CRC hard block present in Xilinx® Virtex™-5 LXT/SXT devices.

Theory

Addition and subtraction operations are performed using modulo 2 arithmetic; that is, they are the same as the exclusive OR (XOR) operation. Adding two numbers in polynomial arithmetic is the same as adding numbers in ordinary binary arithmetic, except there is no carry.

For example, the binary message stream 11001011 is represented as $x^7+x^6+x^3+x+1$. The transmission and reception points agree on a fixed polynomial called a generator polynomial; this is the key parameter of a CRC calculation.

The data is interpreted as the coefficients of a polynomial, which are divided by a given generator polynomial. The remainder of this division is the CRC. Given an m-bit message sequence and a generator polynomial of degree r, the transmitter creates an n-bit ($n = m+r$) sequence called the frame check sequence (FCS) so that the resultant (m+r)-bit frame is divisible by a predetermined sequence.

The transmitter appends r 0-bits to the m-bit message and divides the resulting polynomial of degree m+r-1 by the generator polynomial. This produces a remainder polynomial of degree (r-1) or less. The remainder polynomial has r coefficients, which form the checksum. The quotient is discarded. The data transmitted is the original m-bit message followed by the r-bit checksum.

At the receiver, you can follow one of two standard approaches to assess the validity of the received data:

- Compute the checksum again for the first m-bits received and compare

CRC is based on polynomial codes defined over a field with two elements. Polynomial codes treat bit streams as polynomial representations with coefficient values of either 0 or 1.

against the received checksum (the last r-bits received)

- Calculate the checksum for all of the (m+r) received bits and compare against a remainder of 0

To see how the second method results in a remainder of 0, let's use the following convention:

M = polynomial representation of the message

R = polynomial representation of the remainder computed at the transmitter

G = generator polynomial

Q = quotient obtained by dividing M by G

The transmitted data corresponds to the polynomial $Mx^r - R$. The variable x^r signifies an r-bit shift of the message to accommodate the checksum.

We know that:

$$Mx^r = QG + R$$

Appending the checksum R to the message at the transmitter is equivalent to subtracting the remainder from the message. The transmitted data then becomes $Mx^r - R = QG$, which is clearly a multiple of G. This is how we obtain a remainder of 0 in the second case.

However, this procedure is insensitive to the number of leading and trailing 0-bits in the data transmitted. In other words, if a message has trailing 0-bits inserted or deleted, the remainder will still remain 0 and the error will go undetected, which manifests that the same bit sequence will not be reproduced back. Let's explain a tactical method to overcome this drawback.

Residue Approach

In practice, the checksum is complemented before appending. This makes the remain-

der calculated (over m+r bits) at the receiver non-0. The remainder, which is obtained at the receiver in such cases, is a fixed value known as the residue value of a polynomial.

A little mathematics helps illustrate this idea more clearly.

Assume that the % symbol denotes a modulo operation in the following representation:

For the case where checksum is appended without inversion:

$$(Mx^r - R) x^r \% G = 0$$

where the receiver again performs the same operation of shifting as the transmitter.

Now consider the case where the checksum is inverted and appended to the message stream at the transmitter:

$$(Mx^r - R^c) x^r \% G$$

where R^c denotes complemented checksum.

This can also be written as:

$$(Mx^r - R + (x^{r-1} + \dots + x + 1)) x^r \% G$$

The complement of a bit is the same as XOR with 1. Here, the + sign represents addition in modulo 2 arithmetic (also note that addition and subtraction are the same in modulo 2 arithmetic).

In which case, the remainder is the same as:

$$(x^{r-1} + \dots + x + 1) x^r \% G$$

which works out to be a constant for a given generator polynomial.

The most commonly used CRC-32 generator polynomial is

$$G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

which is 04C11DB7 in hexadecimal.

The constant residue value corresponding to CRC-32 is C704DD7B in hexadecimal. For a given generator polynomial G, the residue value remains a constant for any data pattern provided at the input.

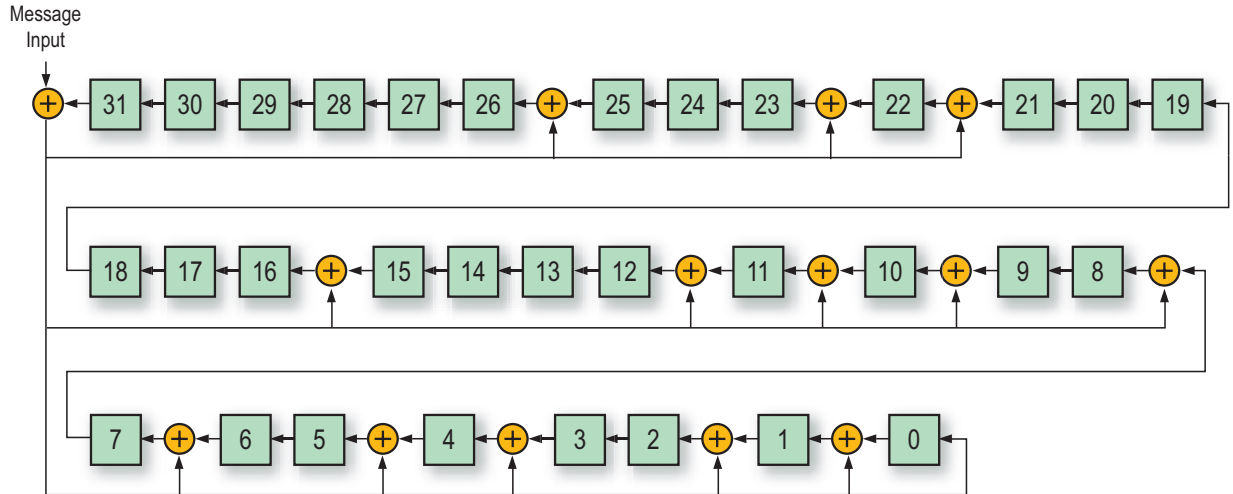


Figure 1 – LFSR implementation for the CRC32 polynomial

Hardware Implementation

The computation of a CRC checksum is a polynomial division process. Implementing it in hardware makes use of a shift register (also called the CRC register). The length of the shift register is the same as the degree of the generator polynomial.

The procedure for CRC calculation is to:

1. Initialize the CRC register.
2. Get the message bit and continue as long as message bits exist. If the higher order bit in the CRC register is 1, shift left one position and XOR the result with G. Otherwise, shift left one position.

When all of these steps are completed for a given message, the CRC register holds the remainder.

These steps can be implemented with a circuit known as the linear feedback shift register (LFSR). Figure 1 shows an LFSR implementation for calculating CRC using the CRC32 polynomial. Note that the placement of XOR gates depends on the coefficient of the corresponding terms being 1 in the generator polynomial. Each of the numbered blocks in the figure represents a memory element (flip-flop).

CRC Block

The hardware implementation of CRC makes use of a simple LFSR. Although such a circuit is simple to implement, it takes n-clock cycles to calculate CRC values for an

n-bit data stream. This latency is intolerable in high-speed data networking applications where data frames must be processed at higher speeds. The implementation of CRC generation and checking on a parallel stream of data become desirable in such high-speed networking applications.

as input. The functionality for CRC comparison is beyond the scope of the hard block and should be built into the FPGA fabric.

Each CRC hard block in the FPGA computes a 32-bit checksum asynchronously. Figure 2 is a block-level diagram

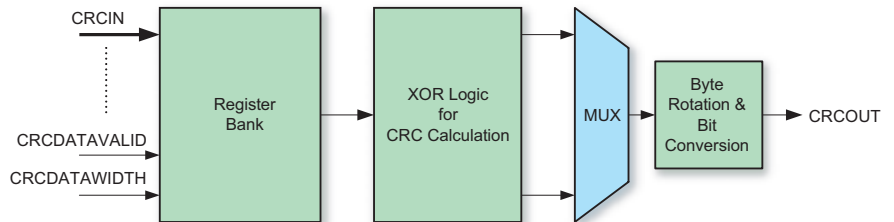


Figure 2 – CRC hard block implementation

The CRC block implemented in Virtex-5 LXT/SXT devices helps designers by speeding up checksum calculations.

The CRC hard block in Virtex-5 LXT/SXT devices is based on the CRC32 polynomial. Virtex-5 FPGAs contain CRC32 and CRC64 hard blocks, which provide a latency of one clock cycle for CRC generation for 4- and 8-byte data input. The interface is easy and simple to use. The hard block functions as a CRC calculator on a given message stream, along with some CRC-specific parameters

depicting the hard block's architecture. The CRC hard block provides an output, which is bit-inverted and byte-reversed.

Figure 3 provides an application overview of the CRC hard block. CRC is calculated and appended at the end for a given data packet at the transmitter. At the receiver, CRC is re-calculated over the entire received packet along with the appended CRC from the transmitter.

The validity of the received packet is established by the residue method. In this case, for the CRC32 polynomial, the

residue works out to be 1CDF4421 in hexadecimal, as it is a bit-inverted and byte-reversed value for C704DDB7. The concept of byte rotation and bit inversion is shown in Figure 4.

Figure 5 illustrates a waveform for normal CRC operation.

We have also come up with a LogiCORE™ CRC wizard, which provides

a LocalLink wrapper for the CRC hard macro present in Virtex-5 devices. The core also provides an example design demonstrating the use of the CRC hard block. Additionally, the core provides various options such as pipelining, complementing, and transposing.

For more information, visit www.xilinx.com/crcwizard.

Conclusion

The presence of CRC blocks in Xilinx FPGAs makes the inclusion of error-detection mechanisms in various designs easier and effortless for designers. You can use the CRC Wizard IP, available in CORE Generator™ software, to incorporate error-detection in various protocols like Aurora and PCI Express.

For more information on the embedded CRC blocks in Virtex-5 FPGAs, see the Virtex-5 RocketIO™ GTP Transceiver User Guide (UG196) at www.xilinx.com/bvdocs/userguides/ug196.pdf.

The theory on CRC presented in this article is a summary of the CRC theory published in various technical journals and textbooks. Please contact the authors for applicable references and URLs.

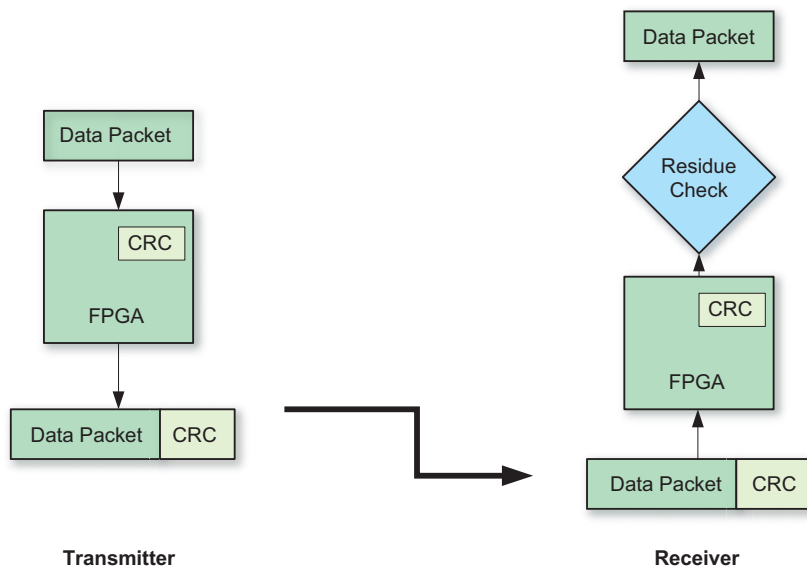


Figure 3 – Application overview of CRC block

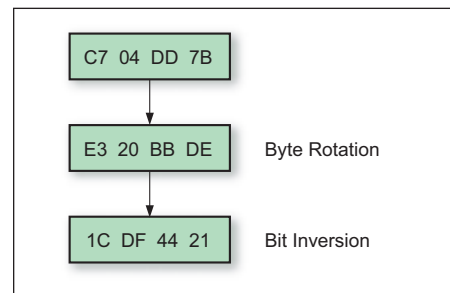


Figure 4 – Byte rotation and bit inversion depiction

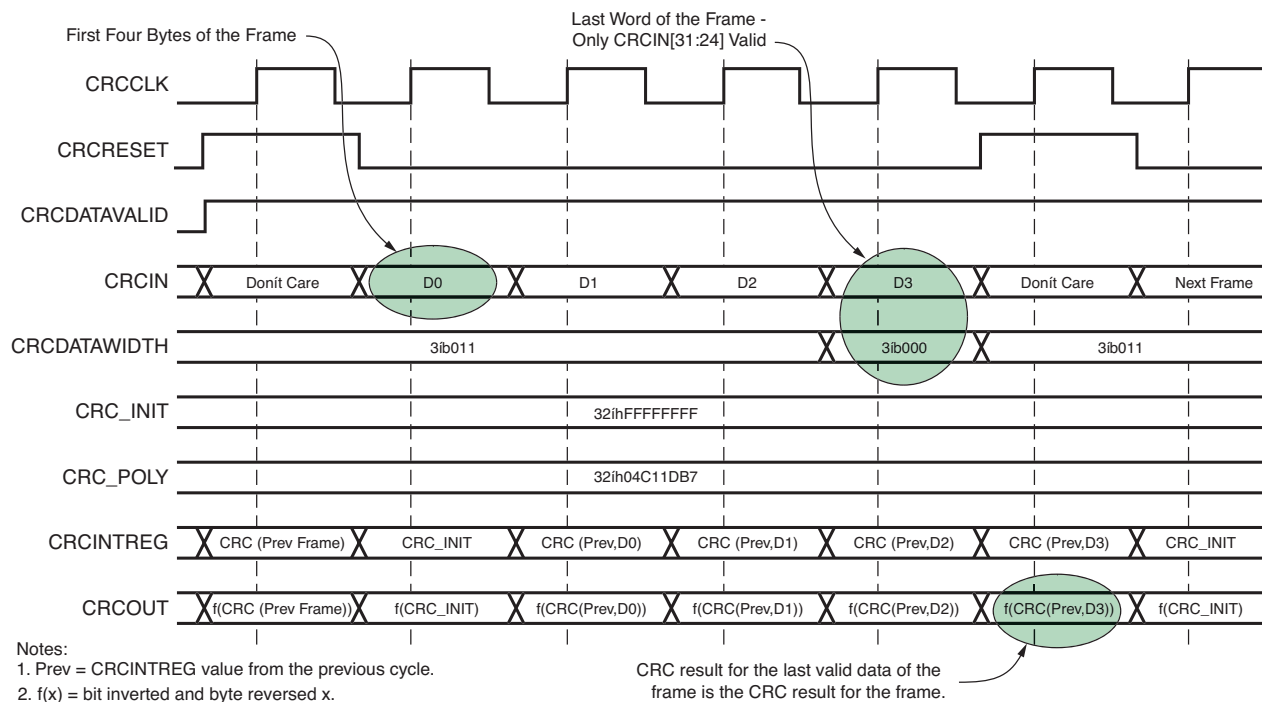


Figure 5 – Waveform for CRC operation