XAPP209 (v1.0) March 23, 2001

IEEE 802.3 Cyclic Redundancy Check

Author: Chris Borrelli

Summary

Cyclic Redundancy Check (CRC) is an error-checking code that is widely used in data communication systems and other serial data transmission systems. CRC is based on polynomial manipulations using modulo arithmetic. Some of the common Cyclic Redundancy Check standards are CRC-8, CRC-12, CRC-16, CRC-32, and CRC-CCIT. This application note discusses the implementation of an IEEE 802.3 CRC in a Virtex™ device. The reference design provided with this application note provides Verilog point solutions for CRC-8, CRC-12, CRC-16, and CRC-32. The Perl script (`crcgen.pl`) used to generate this code is also included. The script generates Verilog source for CRC circuitry of any width (8, 12, 16, 32), any polynomial, and any data input width.

Introduction

In networking systems a significant role of the Data Link layer is to convert the potentially unreliable physical link between two machines into an apparently very reliable link. This is achieved by including redundant information in each transmitted frame. Depending on the nature of the link and the data, one can include just enough redundancy to make it possible to detect errors and then arrange for the retransmission of damaged frames. The cyclic redundancy check or CRC is a widely used parity bit based error detection scheme in serial data transmission applications. This code is based on polynomial arithmetic.

The bits of data to be transmitted are the coefficients of the polynomial. As an example, the bit stream `1101011011` has 10-bits, representing a 10-term polynomial:

$$M(x) = 1 \cdot x^9 + 1 \cdot x^8 + 0 \cdot x^7 + 1 \cdot x^6 + 0 \cdot x^5 + 1 \cdot x^4 + 1 \cdot x^3 + 0 \cdot x^2 + 1 \cdot x^1 + 1 \cdot x^0$$

$$M(x) = x^9 + x^8 + x^6 + x^4 + x^3 + x^1 + 1$$

To compute the CRC of a message, another polynomial called the generator polynomial G(x) is chosen. G(x) should have a degree greater than zero and less than that of the polynomial M(x). Another requirement for G(x) is a non-zero coefficient in the x^0 term. This results in several possible options for the generator polynomial, and hence the need for standardization.

CRC-16 is one such standard that uses the generating polynomial:

$$G(x) = x^{16} + x^{15} + x^2 + 1$$

CRC-16 detects all single and double errors, all errors with an odd number of bits, all burst errors of length 16 or less, and most errors for longer bursts.

CRC-32 uses the generating polynomial:

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

In general, an n-bit CRC is calculated by representing the data stream as a polynomial M(x), multiplying M(x) by x^n (where n is the degree of the polynomial G(x)), and dividing the result by a generator polynomial G(x). The resulting remainder is appended to the polynomial M(x) and transmitted. The complete transmitted polynomial is then divided by the same generator polynomial at the receiver end. If the result of this division has no remainder, there are no transmission errors. Mathematically, this can be represented as:

$$\text{CRC} = \text{remainder of} \left[M(x) \times \frac{x^n}{G(x)} \right]$$

Cyclic Redundancy Check (CRC)

CRC computation involves manipulating M(x) and G(x) using modulo 2 arithmetic. Modulo arithmetic yields the same result for addition and subtraction. Therefore it is necessary only to consider three operations involving polynomials namely, addition, multiplication, and division.

The addition of two polynomials $x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$ and $x^5 + x^4 + x^3 + x^2$ yields $x^8 + x^7 + x^3 + x + 1$.

$$x^8 + x^7 + x^5 + x^4 + 0 + x^2 + x + 1 = 110110111$$
$$\underline{0 + 0 + x^5 + x^4 + x^3 + x^2 + 0 + 0 = 000111100}$$
$$x^8 + x^7 + 0 + 0 + x^3 + 0 + x + 1 = 110001011$$

The multiplication of two polynomials $x^7 + x^6 + x^5 + x^2 + 1$ and $x + 1$ results in $x^8 + x^5 + x^3 + x^2 + x + 1$

$$(x^7 + x^6 + x^5 + x^2 + 1)(x + 1) = (11100101) \times (11)$$

$$x^8 + x^7 + x^6 + 0 + 0 + x^3 + 0 + x + 0 = 111001010$$
$$\underline{0 + x^7 + x^6 + x^5 + 0 + 0 + x^2 + 0 + 1 = 011100101}$$
$$x^8 + 0 + 0 + x^5 + 0 + x^3 + x^2 + x + 1 = 100101111$$

Note that multiplication of a polynomial by x^m results in a shifted bit pattern with zeros in the lower m positions. For example:

$$x^5(x^{11} + x^{10} + x^8 + x^4 + x^3 + x + 1) = x^{16} + x^{15} + x^{13} + x^9 + x^8 + x^6 + x^5$$

Dividing $x^{13} + x^{11} + x^{10} + x^7 + x^4 + x^3 + x + 1$ by $x^6 + x^5 + x^4 + x^3 + 1$ results in a quotient of $x^7 + x^6 + x^5 + x^2 + x + 1$ and a remainder of $x^4 + x^2$ as shown below.

$x^{13} + x^{11} + x^{10} + x^7 + x^4 + x^3 + x + 1 = 10110010011011$

$x^6 + x^5 + x^4 + x^3 + 1 = 1111001$

```
            11100111
        ____________
1111001)10110010011011
        1111001
         1000000
         1111001
          1110010
          1111001
             1011110
             1111001
              1001111
              1111001
                1101101
                1111001
                  10100
```

$Q(x) = 11100111 = x^7 + x^6 + x^5 + x^2 + x + 1$

$R(x) = 10100 = x^4 + x^2$

IEEE 802.3 CRC-32

IEEE 802.3 defines the polynomial M(x) as the destination address, source address, length/type, and data of a frame, with the first 32-bits complemented. The remainder from the calculation of CRC above is complemented, and the result is the IEEE 802.3 32-bit CRC, referred to as the Frame Check Sequence (FCS) field. The FCS is appended to the end of the Ethernet frame, and is transmitted highest order bit first (x^{31}, x^{30},..., x^1, x^0).

Serial Input Hardware Implementation of CRC-32

The single-bit data input (serial) calculation of CRC-32 is implemented with a linear feedback shift register (LFSR). The CRC-32 LFSR is illustrated in Figure 1 (register bits "3" through "25" are left out of the figure to simplify the drawing).

Presetting the flip-flops to `0xFFFFFFFF` is equivalent to complementing the first 32-bits of the data stream. For the first 32 cycles, the right-most XOR gate in the figure is an inverter. The XOR of any data with a binary "1" results in the complement of the original data.

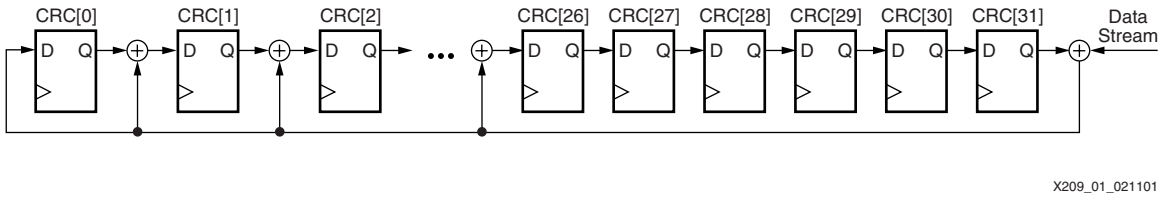

Figure 1: **Linear Feedback Shift Register Implementation of CRC-32**

The serial implementation is not optimal as most IEEE 802.3 transceivers communicate the data stream to the Media Access Controller (MAC) over a 4-bit bus; therefore, a parallel implementation is necessary.

Parallel Data Input Hardware Implementation of CRC-32

A parallel implementation operates on multiple bits of the data stream per clock cycle. An algorithm for generating the next-state equations for parallel implementation of CRC-32 is discussed in "A Symbol Based Algorithm for Hardware Implementation of Cyclic Redundancy Check (CRC)"[1]. The algorithm involves looping to simulate the shifting, and concatenating strings to build the equations after "n" shifts. The Perl script `crcgen.pl` uses this algorithm to generate next-state equations for CRC-32 registers.

Included with this application note is Verilog source code for CRC-32 with 8-bit data input, which was generated by the Perl script `crcgen.pl`. The Perl script also generates equations for any CRC width, data input width, and polynomial, G(x). The Verilog code is included for several popular implementations.

A Complete Implementation of IEEE 802.3 CRC-32

As discussed earlier, there is more to calculating CRC-32 for IEEE 802.3. The Perl script `crcgen.pl` also generates control logic to handle the final complement of the CRC before it is sent out bit reversed on an 8-bit wide bus. A simplified block diagram of the complete implementation is illustrated in Figure 2, along with the functional truth table in Table 1. This module can be used for both transmission and receiving of IEEE 802.3 frames.

The "32-bit CRC" register shown Figure 2 has three modes of operation: initialize, shift, and calculate next CRC value. This register is enabled when either `load_init` or `d_valid` are asserted.

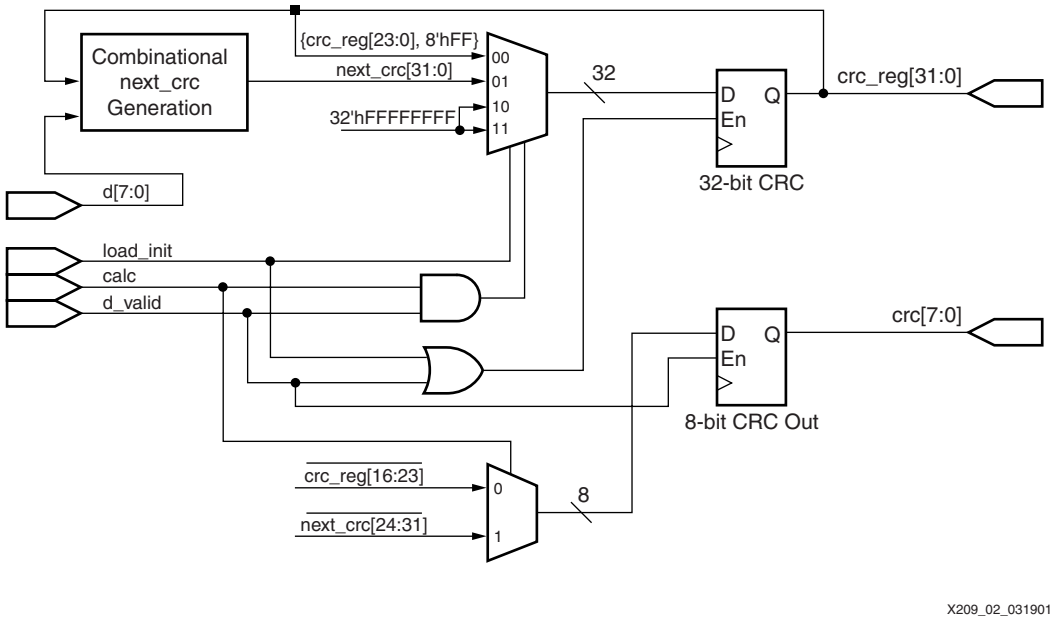

Figure 2: **Block Diagram of CRC-32 Implementation**

The "8-bit CRC Out" register duplicates 8-bits of `crc_reg`[31:0] in order to do the final bit reversal and complement in parallel with the calculation of the CRC. When the `calc` input is asserted, the "8-bit CRC Out" register gets the bit-reversed and complemented most significant eight bits of `next_crc`. On the other hand, when the `calc` input is de-asserted, the "8-bit CRC Out" register gets the bit-reversed and complemented second most significant eight bits of `crc_reg` - the second most significant eight bits are used here to replicate the shift operation of the "32-bit CRC" register.

The "8-bit CRC Out" register always contains the bit-reversed and complimented most significant bits of the "32-bit CRC" register. The final IEEE 802.3 FCS can be read from the "8-bit CRC Out" register by asserting `d_valid` four times after the de-assertion of `calc`. An example waveform is shown in Figure 3 and Figure 4.

Table 1: **CRC-32 Implementation Truth Table**

{load_init, calc, d_valid}	crc_reg[31:0]	crc[7:0]
0 0 0	no change	no change
0 0 1	{ crc_reg[23:0], 8'hFF }	$\overline{\texttt{crc\_reg[16:23]}}$
0 1 0	no change	no change
0 1 1	next_crc[31:0]	$\overline{\texttt{next\_crc[24:31]}}$
1 0 0	32'hFFFFFFFF	no change
1 0 1	32'hFFFFFFFF	$\overline{\texttt{crc\_reg[16:23]}}$
1 1 0	32'hFFFFFFFF	no change
1 1 1	32'hFFFFFFFF	$\overline{\texttt{next\_crc[24:31]}}$

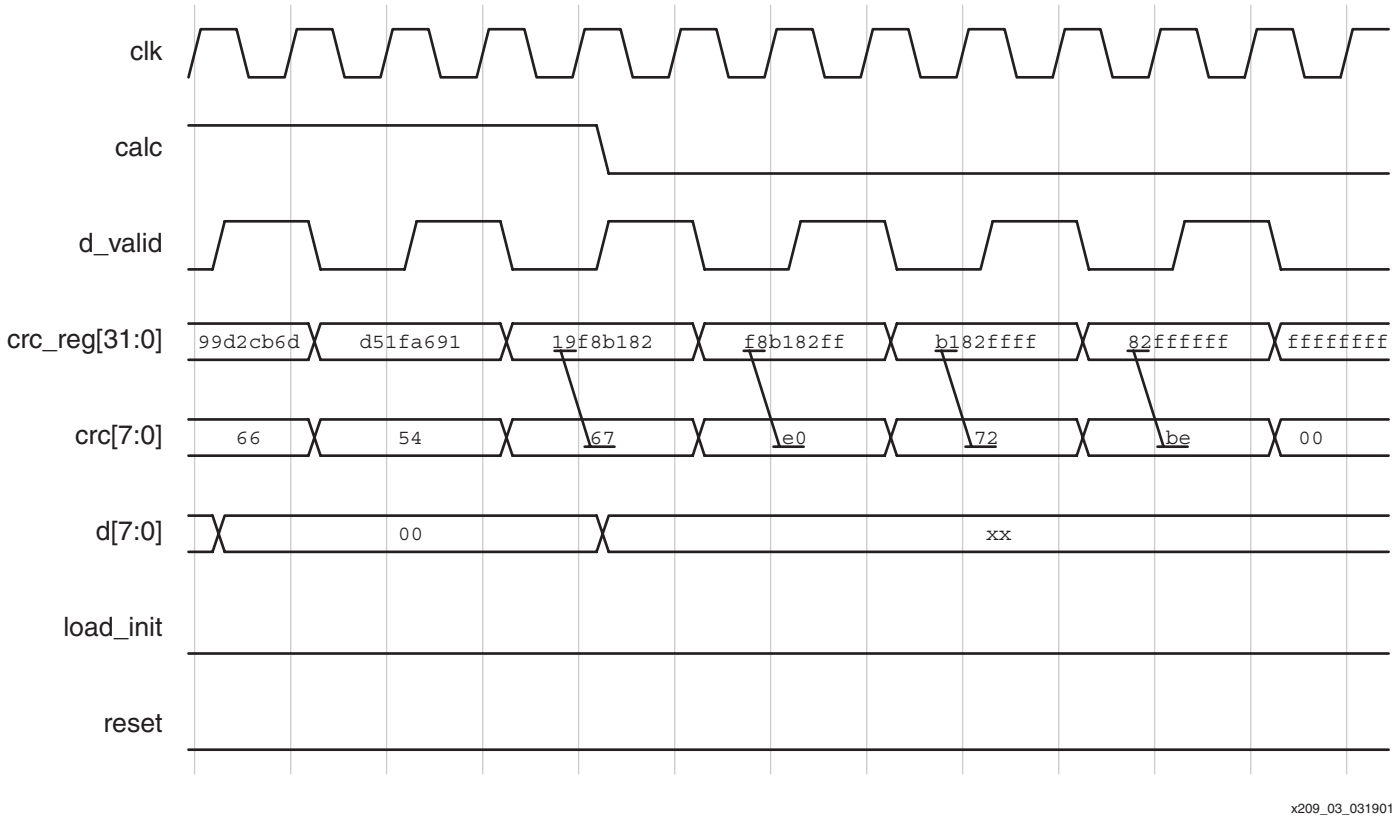

Figure 3: **Example Use of "CRC Out" Register During Transmit**

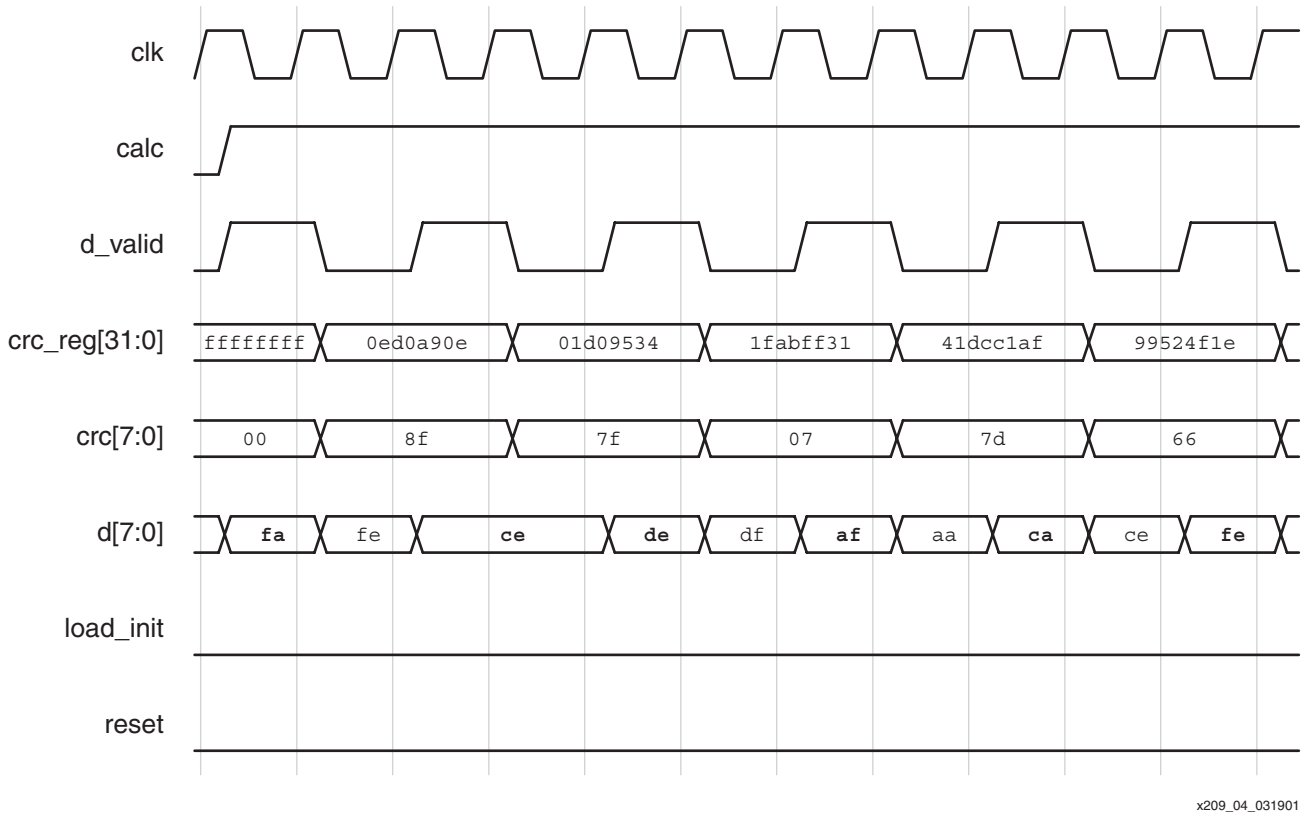

Figure 4: **Example CRC-32 Waveform From the Beginning of the Frame**

A 32-bit compare block can be added for checking the CRC of received frames. First, the data stream and CRC of the received frame are sent through the circuit in Figure 2. Then the value left in the CRC-32 registers can be compared with a constant, commonly referred to as the residue. In this implementation, the value of the residue is `0xC704DD7B` when no CRC errors are detected.

Reference Design

crcgen.pl

The usage of the `crcgen.pl` Perl script is quite simple. The command-line options are shown below.

Usage: `crcgen.pl` [`-crcwidth` <width:8,12,16,32>] [`-inputwidth` <width:1…32>] [`-poly` <polynomial>][`-crcint` <init:preset, reset>], [`-h`] <outfile>

Where:

`-crcwidth` = Width of CRC

Default: 32

`-inputwidth` = Number of bits of the input data stream to operate on per clock cycle. This number should be an even multiple of the value given in the crcwidth option above.

Default: 8

`-poly` = Generating polynomial. This should be entered in binary form as $X^n \ldots X^0$. For example: the polynomial $x^{16} + x^{15} + x^2 + 1$ should be entered as `1100000000000101`. The MSB of the binary value is on the left.

Default: `10000010011000001000111011011011` (IEEE 802.3 CRC-32)

`-crcint` = Initial state of the `crc_reg` register.

Default: preset

`-h` = Displays usage summary.

`<outfile>` = Output filename.

Default: `crc_$crcwidth_$inputwidth.v`

Perl Script Details

The Perl script is divided into several sections. These sections are described below and are labeled as they appear in the script.

Command-line Parsing

This section parses the command-line arguments. The details of this are beyond the scope of this application note. For more information regarding this, please visit http://www.perl.org.

Check Validity of Parameters

This section checks that any parameters passed on the command-line are valid. There are two restrictions on the values of command-line options `crcwidth`, `inputwidth`, and `poly`:

- The `poly` value must be a string of "1s" and "0s" of length (`crcwidth + 1`).
- The `crcwidth` value must be evenly divisible by the `inputwidth` value.

Generate XOR Equations

This section of the script generates the XOR equations based on the algorithm discussed in "A Symbol Based Algorithm for Hardware Implementation of Cyclic Redundancy Check

(CRC)"[1]. It involves looping to simulate the shifting operations and concatenating strings representing the CRC registers and data input.

Optimize XOR equations

This section of the script optimizes the XOR equations generated above. It looks for an even number of occurrences of a particular variable in each equation and eliminates them. For example, "`A XOR A XOR B`" would be reduced to "B" because the even number of the variable A causes it to be optimized out of the equation.

Generate Verilog source for Module Statement and I/O Definitions

This section of the script simply generates Verilog source code for the module statement. It also uses the values parsed from the command-line options to define the input/output widths of the registers.

Generate Internal Strings for Later Verilog Source Output

This section of the script calculates the most significant n-bits and the second most significant n-bits given the command-line options `-crcwidth` and `-inputwidth`.

Table 2 lists the five character strings that are defined in this section of the script.

Table 2: **Script Character Strings**

`$crc_out1`	String representing the input to the `crc` register when the `calc` input is asserted.
`$crc_out2`	String representing the input to the `crc` register when the `calc` input is de-asserted.
`$crc_reg_init`	String representing the initial state of the `crc_reg` register.
`$crc_init`	String representing the initial state of the `crc` register.
`$crc_reg_shift_index`	String representing the upper bus index of the `crc_reg` operating in shift mode (`calc == 0`).

Generate Verilog Source for Registers and Control Logic

This section outputs the Verilog source for the `crc_reg` and `crc` registers, as well as the control logic shown in Figure 2. The Perl strings in the previous section are used here to permit parameterization of register widths.

Generate Verilog Source From Optimized XOR Equations

The optimized XOR equations are printed in this section. This logic is represented as "Combinational next_crc Generation" in Figure 2 above.

Generate Verilog Source for Endmodule Statement

This section simply outputs the `endmodule` Verilog statement.

Design Files

The reference design is available on the Xilinx web site: **xapp209.zip**

Performance and Utilization

Performance and Utilization of IEEE 802.3 CRC-32 Implementation

The CRC-32 implementation described here was synthesized using XST and targeted to the Virtex-II architecture. The performance and utilization are summarized in Table 3.

Table 3: **Utilization and Performance Summary**

Design	Device	Speed Grade	Utilization	Performance
CRC-32, 8-bit input	XC2V40-FG256	-4	65 slices	212 MHz

Conclusion

The Ethernet CRC specified by IEEE 802.3 requires complementing and bit-reversed transmission of the generated CRC. This application note demonstrates an effective implementation of the Ethernet CRC in Virtex series devices.

References

[1] Rajesh Nair, Gerry Ryan and Farivar Farzaneh, A Symbol Based Algorithm for Hardware Implementation of Cyclic Redundancy Check (CRC), Bay Networks.

[2] Krishna Rallapalli, Cyclic Checks for Error Detection

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
03/23/01	1.0	Initial Xilinx Release