safeassign by Blackboard

Filename:	VHDL implementation of an error detection and correction module based on Hamming code	**Submitted:**	Tue, May 08 2012, 1:47 PM
Matching:	92 %	**Paper ID:**	47260347

Print version:

Direct link

Suspected Sources

Click on a source to view the original, or click on the magnifying glass to see the source highlighted in the text below.

1. Highlight All | Unhighlight All
2. 1 http://www.xilinx.com/support/documentation/application_notes/xapp645.pdf
3. 2 http://www.altera.com/literature/an/an415.pdf
4. 3 http://www.eetasia.com/ARTICLES/2004DEC/A/2004DEC10_CT_AN.PDF?SOURCES=DOWNLOAD
5. 4 http://www.scribd.com/doc/20706609/Error-Correcting-Codes
6. 5 http://www.hackersdelight.org/ecc.pdf
7. 6 http://www.iis.ee.ethz.ch/~kgf/aries/all.html
8. 7 http://www.ijaest.iserp.org/archieves/14-Jul-1-15-11/Vol-No.8-Issue-No.2/23.IJAEST-Vol-No-8-Issue-No
9. **Re-process the paper without the selected sources**

Paper Text

1 15Abstract − Error detection and correction (ECC) is found in many high-reliability and performance

applications. 2 Error correction codes (ECCs) permit detection and correction of errors that result from noise or other impairments during transmission from the transmitter to the receiver. 1 18ECC becomes an important feature for many communication applications; it is more performance and cost efficient to correct an error rather than retransmit the data.
3 19This paper describes the implementation of an Error Correction Control (ECC) module in a Virtex-4 device.
Keywords: Hamming code; 18error detection, error correction; FPGA circuits.

I. INTRODUCTION

The design described in this paper implements error detection and correction at the speed of the data read/ write rate. 1 It detects double bit errors and corrects single bit errors anywhere within the codeword. The design detects and corrects all single bit errors (in a codeword consisting of either 64-bit data and 8 parity bits, or 32-bit data and 7 parity bits), and it detects double bit errors in the data.

This design utilizes Hamming code, a simple yet powerful method for ECC operations. As a result, this design offers exceptional performance and resource utilization.

2 Hamming code is a relatively simple yet powerful ECC code. 1 It involves transmitting data with multiple check bits (parity) and decoding the associated check bits when receiving data to detect errors. The check bits are parallel parity bits generated from XORing certain bits in the original data word. If bit error are introduced in the codeword, several check bits show parity errors after decoding the retrieved codeword. The combination of these check bit errors display the nature of the error and the position of any single bit error is identified from the check bits.

The Hamming codeword is a concatenation of the original data and the check bits (parity). The data to be transmitted consists of a certain number of information

4 12bits u, and he adds to these a number of check bits p such that if a block is received that has at most one bit in error, then p identifies the bit that is in error (which may be one of the check bits). Specifically, in Hamming"s code p is interpreted as an integer which is 0 if no error occurred, and otherwise is the 1 - origined index of the bit that is in error. Let k be the number of information bits, and m the number of check bits used [2]. Because the m check bits must check themselves as well as the information bits, the value of p, interpreted as an integer, must range from 0 to which is distinct values.

Because m bits can distinguish cases, we must have:

5 52m ≥ m + k + 1 (1)

where:
4 12k - Number of "information" or "message" bits.
m - Number of parity-check bits ("check bits," for short).
6 The value of „m" can be determined by substituting

the value of „k“ (the original length of the data to be transmitted).

4 The check bits will be interspersed among the information bits. Because p indexes the bit (if any) that is in error, the least significant bit of p must be 1 if the erroneous bit is in an odd position, and 0 if it is in an even position or if there is no error. A simple way to achieve this is to let the least significant bit of p, p0, be an even parity check on the odd positions of the block, and to put p0 in an odd position. The receiver then checks the parity of the odd positions (including that of p0). If the result is 1, an error has occurred in an odd position, and if the result is 0, either no error occurred or an error occurred in an even position [5]. This satisfies the condition that p should be the index of the erroneous bit, or be 0 if no error occurred.

Similarly, let the next from least significant bit of p, p1, be an even parity check of positions 2, 3, 6, 7, 10, 11, .n
(in binary, 10, 11, 110, 111, 1010, 1011,.), and put p1 in one of these positions. Those positions have a 1 in their second from least significant binary position number. The receiver checks the parity of these positions (including the position of p1). If the result is 1, an error occurred in one of those positions, and if the

---

result is 0, either no error occurred or an error occurred in some other position. Continuing, the third from least significant check bit, p2, is made an even parity check on those positions that have a 1 in their third from least significant position number, namely positions 4, 5, 6, 7, 12, 13, 14, 15, 20,. , and p2 is put in one of those positions.

Putting the check bits in power-of-two positions (1, 2, 4, 8,.) has the advantage that they are independent. That is, the sender can compute p0 independently of p1, p2,. and, more generally, it can compute each check bit independently of the others[1].

Hamming codes can be computed in linear algebra terms through matrices because Hamming codes are linear codes. Two Hamming matrices can be defined:
the code generator matrix G and the parity-check matrix H.

1 18The parity matrix [P] can be expressed as (equation 1):

$$[P]=[D]\bullet[G] \quad (1)$$

where [D] is the data matrix and [G] is the generator matrix.

The [G] matrix consists of an identity matrix [I] and a creation matrix [C] (equation 2).

$$[G] = [I:C] \quad (2)$$

To detect errors, the codeword vector multiplies with the transpose of the generator matrix to produce an 8-bit vector [S], known as syndrome vector (equation 3):

$$[S]=[D,P]\bullet[G'] \quad (3)$$

For example, the (7,4) Hamming code is shown in equation (4):

> ≅≈ ≈ ≈ ≈

.nn
≡

↔ ↔ ↔ ↔

←

♠

≈ ≈ ≈ ≈

.nn
≡

↔ ↔ ↔

↔

←

♠

011

101

110

111

1000

0100

0010

0001

G (4)

If all of the elements of the syndrome vector are zeros, no error is reported. Any other non-zero result represents the bit error type and provides the location of any single bit errors. It is then used to correct the original incoming data[4].

II. PARITY ENCODER AND PARITY DECODER

A. The Encoder

2 Encoding is performed by multiplying the original

message vector by the generator matrix. 1 The encoder consists of XORs and a bit-error generator implemented in look-up tables (LUTs). It is computationally inexpensive. Figure 1 shows the block diagram of the parity encoder.

3 19This design uses the (72,64) Hamming code. 1 18It mens that the Hamming codeword width is 72 bits, comprised of 64 data bits and eight check bits.em

The check bits are written in the memory along with the associated 64-bit data. During a memory read, the data and the check bits are read simultaneously. Any error introduced during the read/write access between the FPGA and memory are detected. The parity bits are generated based on an unmodified Hamming code.

Figure 1. 2 Parity encoder block diagram

B. The Decoder

Decoding is performed by multiplying the codeword vector by the parity check matrix. The Hamming decoder module generates a syndrome vector to determine if there is any error in the received code word.
The ECC detects single-bit and double-bit errors, but only single-bit errors are corrected.

1 The decoder unit shown in Figure 2 consists of three blocks: Syndrome generation block, Syndrome LUT and mask generation block and Data correction block [4].

Figure 2. 2 ECC functional block diagram

In figure 1 and figure 2 the input/output signals

descriptions are: 1 PARITY_OUT - Parity bits generated from the encoder based on the data (encin) registered at the same clock edge; PARITY_IN - Parity bits associated with the incoming data (DECIN) registered at the same rising clock edge; ENCIN - Original data input to the encoder; ENCOUT - Registered original data through the encoder; DECIN - Incoming data to the decoder; DECOUT - Corrected data from DECIN; FORCE_ERROR - Introduce bit error in the encoded dataword for test purpose (00 – Normal operation, 01 – Inject single bit error, 10 – Inject double bit error, 11 – Inject triple bit error).

The incoming 64-bit data along with the 8-bit parity are XOR'd together to generate the 8-bit syndrome (S1 through S8). This is very similar to check bit generation, for example (equation 6):

S1=DECIN0 ⊕DECIN1 ⊕DECIN3 ⊕DECIN4 ⊕

DECIN6 ⊕DECIN8 ⊕DECIN10 ⊕DECIN11 ⊕
DECIN13 ⊕DECIN15 ⊕DECIN17 ⊕DECIN19 ⊕
DECIN21 ⊕DECIN23 ⊕DECIN25 ⊕DECIN26 ⊕DECIN28 ⊕DECIN30 ⊕DECIN32 ⊕DECIN34 ⊕
(6) DECIN36 ⊕DECIN38 ⊕DECIN40 ⊕DECIN42 ⊕
DECIN44 ⊕DECIN46 ⊕DECIN48 ⊕DECIN50 ⊕
DECIN52 ⊕DECIN54 ⊕DECIN56 ⊕DECIN57 ⊕DECIN59 ⊕DECIN61 ⊕DECIN63 ⊕PARITY_IN(1)

Then the next stage uses the syndrome to look for the error type and the error location. An optional pipeline stage can be added here to improve performance further.

In order to correct a single bit error, a 64-bit correction mask is created. Each bit of this mask is generated based on the result of the syndrome from previous stage. When no error is detected, all bits of the mask become zero. When a single bit error is detected, the corresponding mask masks out the rest of the bits except for the error bit. The subsequent stage then XORs the mask with the original data. As a result, the error bit is reversed (or corrected) to the correct state. If a double bit error is detected, all mask bits become zero. The error type and corresponding correction mask are created during the same clock cycle.

In the data correction stage, the mask is XOR'd together with the original incoming data to flip the error bit to the correct state, if needed. When there are no bit errors or double bit errors, all the mask bits are zeros. As a result, the incoming data goes through the ECC unit without changing the original data.

To display the error type, the reference design also supports diagnostic mode. Single, multiple, and triple bit errors can be introduced to the output codeword.

When the ERROR port is 00, no single, two, or greater bit error is detected. In other words, the examined data has no parity error. When the ERROR port is 01, it indicates single bit error occurred within the 72-bit codeword. In addition, the error is corrected, and the data is error free. When the ERROR port is 10, a two bit error has occurred within the codeword. In this case, no error correction is possible in this case. When the ERROR port is 11, errors beyond the detection capability can occur within the codeword and no error correction is possible. This is an invalid error type.

A deliberate bit error can be injected in the codeword at the output of the encoder as a way to test the system. Force_error provides several types of error modes.

Force_error = 00 This is the normal operation mode. No bit error has

been imposed on the output of the encoder.
Force_error = 01 Single bit error mode. One bit is reversed (0 to 1 or 1

7 to 0) in the codeword at every rising edge of the clock.
1 The single bit error follows the sequence moving from

2 bit 0 of the codeword to bit 72. 1 The sequence is repeated as long as this error mode is active.

Force_error = 10 Termed double bit error mode. Two consecutive bits

are reversed (0 becomes 1 or 1 becomes 0) in the codeword at every rising edge of the clock. The double bit error follows the sequence moving from bit (0,1) of the codeword to bit (71, 72). The sequence repeats as long as this error mode is active.

Force_error = 11 Termed triple-error mode. Three-bits are reversed (0

becomes 1 or 1 becomes 0) in a codeword generated at every rising edge of the clock. The double bit error follows the sequence moving from bit (0,1, 2) of the codeword together to bit (70, 71, 72) sequentially. The sequence repeats as long as this error mode is active.

III. IMPLEMENTATION OF HAMMING CODE USING FPGA CIRCUITS

The Hamming code implementation was done in

VHDL [3] using the Xilinx ISE 8.2i package and the XC4VLX25 board [6][7]. We implemented three versions of the Hamming code: for 32, 64 bits wide words and finaly for the 64 bits word we introduced a pipelining stage.

I/Os of the module are registered. For the encoder, the latency from the time the input data is presented at ENCIN to encoded data becomes available at ENCOUT is two clock cycles unpipelined or three clock cycles pipelined. For the decoder, the latency from the time the input data is presented to DECIN to the processed data becomes available at DECOUT is two clock cycles unpipelined or three clock cycles pipelined. The status signal ERROR is synchronous to DECOUT.

The device utilization is shown in figure 3.
Analyzing the file Map Report we can conclude that only 1% of the total CLB circuits have been used.

Figure 3. Device utilization

The next figures demonstrate the way single errors

appear and are then corrected. First the data vector is set to 0. On the next clock, using the FORCE_ERROR signal we introduce an error − the COUNT signal differ from ENCOUNT signal on a single bit (figure 4). On the rising edge of the next clock the error is detected and the ERROR signal changes from 00 to 01. The next clock corrects the error (figure 5).

_____

Figure 4. Introducing an error

Figure 5. 12Correcting the error

IV. CONCLUSIONS

The biggest advantage of the Hamming ECC is its

absolute simplicity in generating of the Hamming code and thereafter recovery of the original data word. Moreover, by use of the relatively simple Hamming rule, data words of any length can be encoded along with the corresponding number of parity bits.

18The reference design utilizes a minimum amount of resources and has high performance.

The design was synthesized using the Xilinx Synthesis Tool (XST). The performance summary is based on ISE 10.1 speed specifications and reflects the 64-bit version ECC reference design only [7].

15Error detection and correction is found in many high- reliability and performance applications. 18For example, in enterprise data storage systems, memory caches are utilized to improve system reliability. The cache is typically placed inside the controller between the host interfaces and the disk array. A robust cache memory design often includes ECC functions to avoid single point of failure losses of customer data. ECC becomes an important feature for many communication applications, such as satellite receivers; it is more performance and cost efficient to correct an error rather than retransmit the data.