FPGA design of the fast decoder for burst errors correction

E A Mytsko[1], **A N Malchukov**[1], **I V Zoev**[1], **S E Ryzhova**[1], **V L Kim**[1]
[1]Tomsk Polytechnic University, 30, Lenina Ave., Tomsk, 634050, Russia

E-mail: evgenvt@tpu.ru

Abstract. The paper is about FPGA design of the fast single stage decoder for correcting burst errors during data transmission. The decoder allows correcting burst errors with 3 bits for a 15 bit codeword and a 7 bit check unit. The description of a generator polynomial search algorithm for building error-correcting codes was represented. The module structure of the decoder was designed for FPGA implementation. There are modules, such as *remainder, check_pattern, decoder2*, implemented by asynchronous combinational circuits without memory elements, and they process each codeword shift in parallel. Proposed implementation allows getting high performance about ~20 ns.

1. Introduction

Error-correcting codes (ECC) are used to detect and correct different errors during data transmission through the communication channels and reading information from storage devices. In papers [1–14], FPGA design of the BCH's code decoder which corrects independent errors is described. However, there are tasks that need to correct errors grouped into bursts which is a special case of independent errors. Reed-Solomon (RS) codes allow solving the tasks of this type. In papers [15–19], FPGA hardware implementations of the Reed-Solomon decoders which are multistage are considered. In the paper, the single stage hardware implementation will be reviewed at a high-speed FPGA decoder which corrects burst errors based on a binary cyclic error-correcting code.

2. Generator polynomial search

For building the cyclic error-correcting code which corrects burst errors, it is required to find a generator polynomial with length $k + 1$ (where k is the length of the check unit) for the determined length of data unit (m) and error-correcting capability (p). A step by step description of the generator polynomial [20] search algorithm is presented later.

Start.

Step 1. Put m, the length of the data unit, and p, the error-correcting capability.

Step 2. The minimum length of the check unit (or the highest power of the generator polynomial) is computed by formula

$$k = \begin{cases} p = 1, n + 1, \\ p = 2, 2*n + 1, \\ p > 2, 2*n + 1 + \sum_{i=1}^{p-2} n * 2^{n}. \end{cases}$$

Where *k* is the highest power of the generator polynomial, *p* is an error-correcting capability, *n* is a codeword length.

Step 3. The first value of the polynomial weight is $w = p$. The minimum code distance for correcting the burst errors of length *p* is $d = p$.

Step 4. The polynomial is selected from a variety of polynomials with highest power *k* and weight *w*.

Step 5. The error-correcting polynomial code is built based on a selected generator polynomial and all the codewords are generated.

Step 6. The minimum code distance of the codeword is computed (the minimum weight among all codewords except zero).

Step 7. If the computed code distance meets the specifications then it goes to step 8, else it goes to step 11.

Step 8. Error syndromes are computed by dividing all combinations of burst errors by the generator polynomial.

Step 9. If all syndromes are unique for the built code, then it goes to step 10, else it goes to step 11.

Step 10. If polynomial $x^n + 1$ is divided by the generator polynomial without the remainder where *n* is the length of the codeword, then it goes to the end, else it goes to step 11.

Step 11. If all polynomials from the variety are checked (the variety is determined by step 4), then it goes to step 12, else it goes to step 4.

Step 12. If the weight *w* equals *k*, then increment *k* and it goes to step 4, else increment *w* and it goes to step 4.

End.

The polynomial search is a preliminary stage for coding and decoding. The cyclic error-correcting code is built by formula

$$CW(x) = M(x) \bullet 2^k \,|\, mod((M(x) \bullet 2^k) / G(x)), \quad (1)$$

where *G(x)* – generator polynomial, *CW(x)* – codeword, *M(x)* – data polynomial, *mod* – remainder operator.

Next, the hardware design of the fast single stage decoder for the cyclic binary code which corrects burst errors is presented.

3. FPGA design of the fast decoder

The block diagram of the cyclic binary decoder which corrects burst errors is presented in figure 1. *N*-1 cyclic shifts of the codeword where *n* is the codeword length are generated by the first step. Generated codewords are supplied to a parallel computation module (*remainder*) which generates remainders of division of codewords by the generator polynomial. Computed remainders are supplied to a *check_pattern* module for matching with one of the burst error patterns. If the remainder matches the pattern, it is summed by modulo 2 with a codeword, and reverse cyclic shifts for *i* bits are carried out where *i* is a sequence number of the remainder which matches with the pattern.

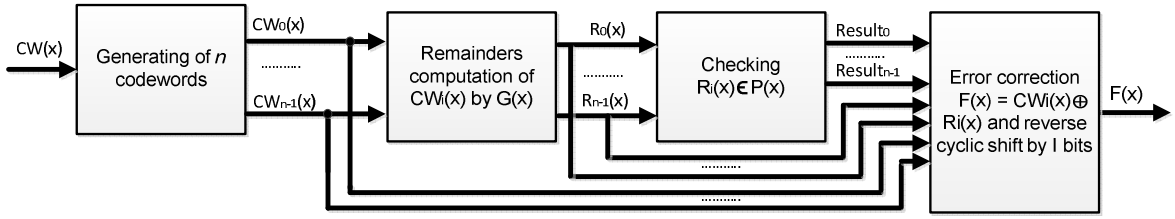

Figure 1. The block diagram of the fast decoder which corrects burst errors.

Figure 2 presents a functional block of the cyclic binary decoder with parameters $n = 15$ (codeword length), $k = 7$ (check unit length), $p = 3$ (errors burst length) which is designed for the *Cyclone* FPGA

using module structure and hardware description language Verilog. The input of the block is a 15-bit codeword and the output is corrected data with length $m = 8$ bits.

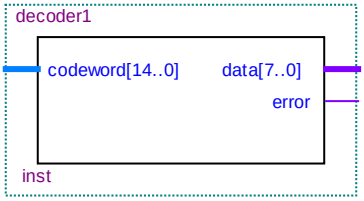

Figure 2. The functional block of the decoder.

The functional block of the decoder includes such modules as *remainder*, *check_pattern*, *pri*, *decoder2*. Each module contains a combinational circuit.

Module *remainder* computes the remainder of division of the codeword by the generator polynomial using matrix division. Figure 3 presents the graphical interpretation of the module at the register-transfer level (RTL view).

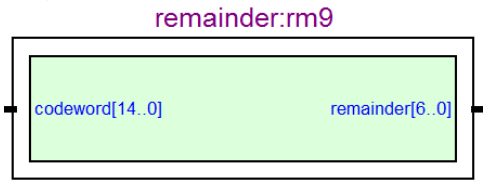

Figure 3. The remainder computation module.

Figure 4 presents a block diagram of module *remainder*. The result of the vector (codeword) by predetermined matrix multiplication is the formula for the combinational circuit which is built using only «exclusive OR» gates. Sign «+» represents «modulo 2» addition in this case.

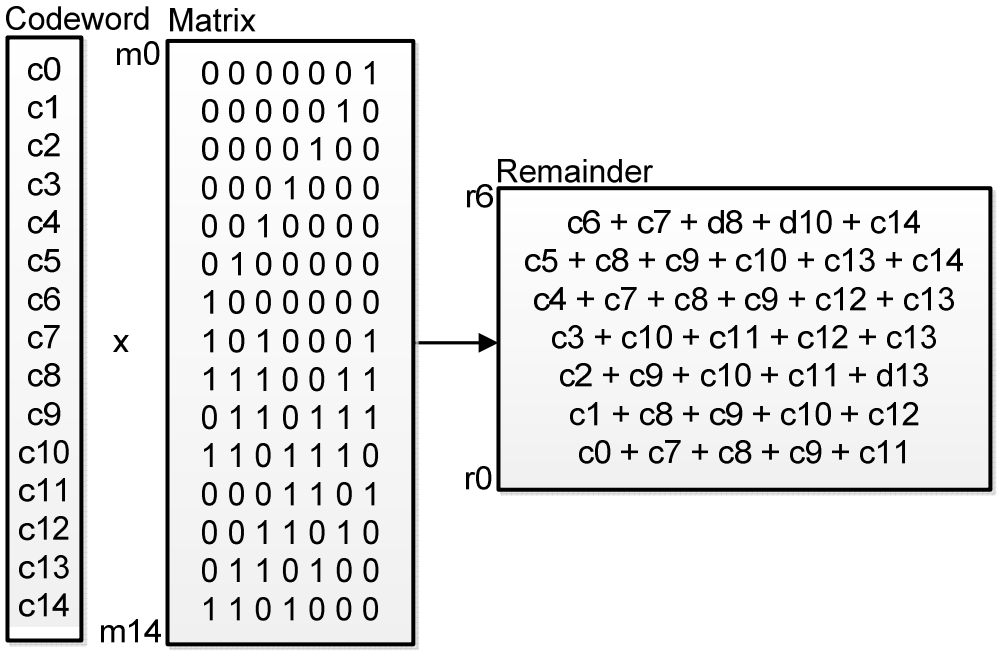

Figure 4. The block diagram of module *remainder*.

Module *check_pattern* checks the remainder for including in the error patterns area ($Ri(x) \in P(x)$). Figure 5 presents a graphical interpretation of module *check_pattern* in the RTL view. If output

result has value «1», then it means that the remainder is included in the error patterns area.

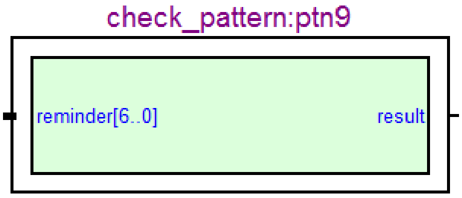

Figure 5. Module *check_pattern*.

The block diagram of *check_pattern* is presented by figure 6. The remainder multiplied (with a bitwise «AND» operation) by the matrix of error patterns with length 3. The combinational circuit based on gates «AND», «OR», «NOT» is built using formula.

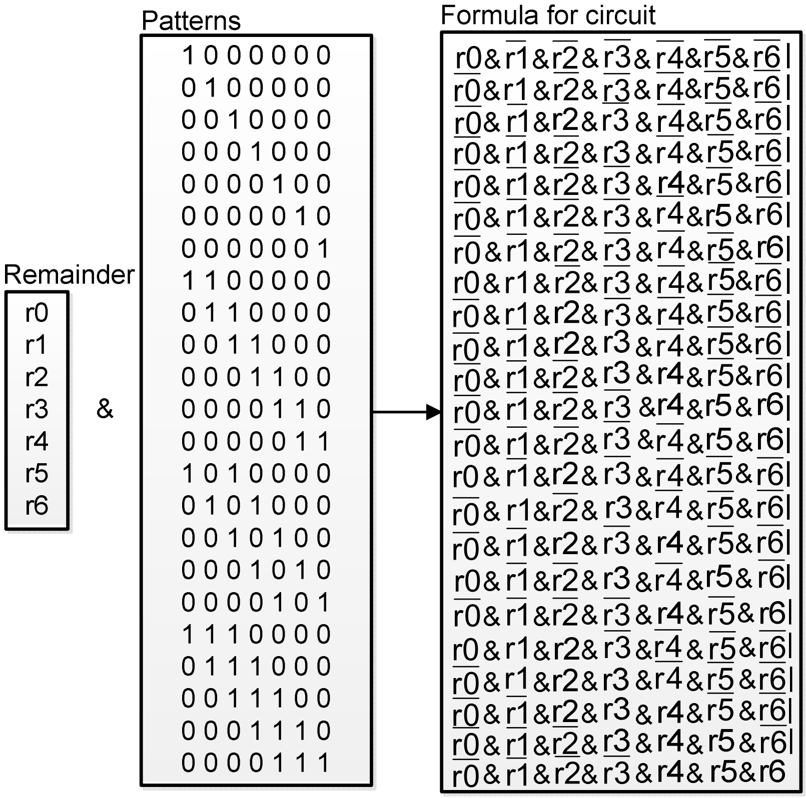

Figure 6. The block diagram of module *check_pattern*.

Module *pri* chooses the highest «1» bit in the code. Code *result[14..0]* is supplied to the module input from *check_pattern* modules. There is a 15-bit code at the output which contains only one single-bit value in the position corresponding to the position of the highest «1» bit of code *result [14..0]*.

Figure 7 presents a graphical interpretation of module *decoder2* in the RTL view. The module has inputs *sw_decoder2* for decoding permission flag, *code_decoder2 [14..0]* for the codeword and *r_decoder [6..0]* for remainders. Output *cw_decoder2 [14..0]* is a corrected codeword.

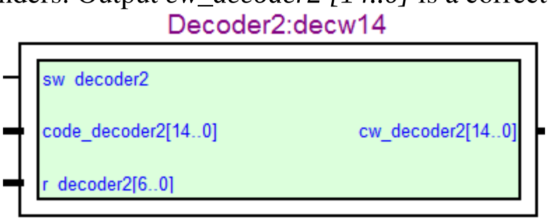

Figure 7. Module *decode2*.

Figure 8 presents a functional diagram of the decoder in the RTL view. Codeword *code_decoder2[14..0]* is summed by modulo 2 with remainder *r_decoder [6..0]*. If the value of permission flag *sw_decoder2* is «1» (condition is realized using a multiplexer 2 in 1), then the result is shifted to the right (the shift is performed by the code bits permutation) by counting bits equal the index number of module *decoder2*.

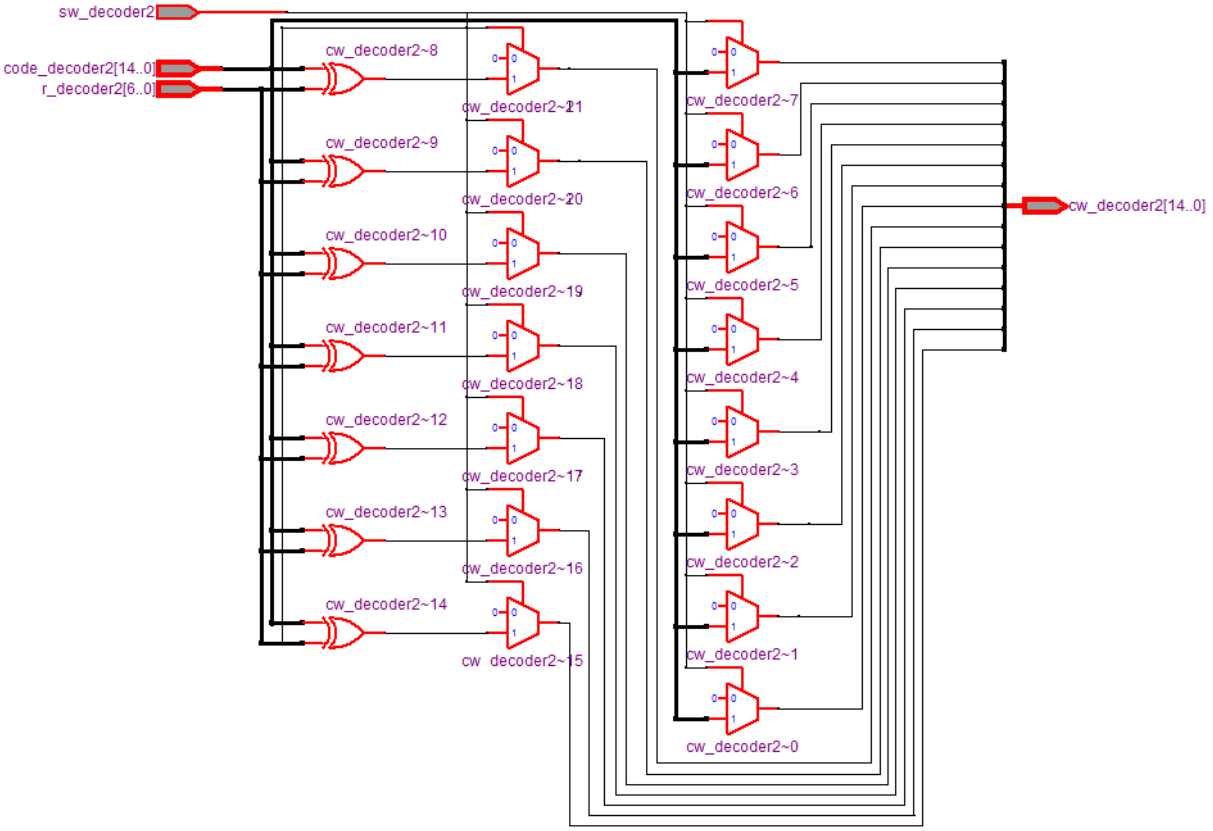

Figure 8. The unctional block of module *decoder2* in the RTL view.

After error correction 8, the bit data unit is selected from the codeword and supplied to the decoder device output.

4. Fast decoder implementation results

Codewords are generated for data byte AA (hex) with different error types to check decoder efficiency (figure 9). There are codewords without errors (556Fh or 101010101101111b), burst of 3 errors (5568h or 101010101101000b), independent two-time error (546Eh or 101010001101110b) and burst of 4 errors (5560h or 101010101100000b). All codewords are represented in hexadecimal.

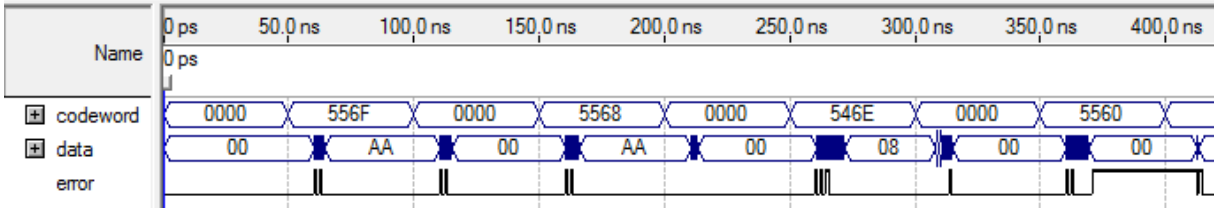

Figure 9. Decoding results.

As a result original data bytes are obtained from the codeword without errors and with the burst of 3 errors. For the burst of 4 errors, signal *error* is generated. It means that the burst length exceeds 3 and the decoder cannot correct this. Table 1 presents the decoder performance (ns) and the number of LUT cells (LUTs).

Table 1. Decoder performance.

Decoding time (ns)	Cells count (LUTs)
19,73	223

5. Conclusion

The structure and FPGA *Cyclone* hardware implementation of the single stage decoder for the cyclic binary error-correcting code, which corrects burst errors, were considered. Features of the generator polynomial search algorithm were described to build error-correcting codes for burst errors. Modules of the decoder functional block were built by combinational circuits. The fast decoder allows correcting 3-bit burst errors for a 15-bit codeword (with 7 check bits) with ~20 ns performance. This performance is achieved by using a matrix division algorithm implemented in the combinational circuits and parallel processing of all codeword shifts. However, there are no results of the performance research in papers [15–19] so the results cannot be compared.

References

[1] Habti A E, Gouri R E, Lichioui A and Laamari H 2015 *J. of Theor. and Appl. Inf. Technol.* **79** 22–28
[2] Elumalai R, Ramachadran A, Alamelu J V and Vihba B R 2014 *Int. J. of Adv. Res. in Electr., Electron, and Instrum. Eng.* **3** 7782–7788
[3] Prakash G and Muthamizhse I 2016 *Int. J. of Innov. Res. in Sci., Eng. and Technol.* **5** 4467–4474
[4] Sunita M S, Chiranth V, Akash H C and Kanchana Bhaaskaran V S 2015 *ARPN J. of Eng. and Appl. Sci.* **10** 3397–3404
[5] Yathiraj H U and Hiremath M R 2014 *Int. J. of Computer Sci. and Mobile Appl.* **2** 45–54
[6] Sutaria H and Khurge D 2013 *Int. J. for Sci. Res. & Develop.* **1** 665–668
[7] Mohammed S J and Abdulsada H F 2013 *Int. J.l of Computer Appl.* **71** 35–42
[8] Mohammed S, Abdulsad H F 2013 *J. of Telecommun.* **19** 11–17
[9] Anas E, Rachid E, Llichioui A and Laamari H 2015 *J. of Theor. and Appl. Inf. Technol.* **79** 22–28
[10] Lee J-H and Shakya S 2013 *Int. J. of Sens. and Its Appl. for Control Syst.* **1** 1–12
[11] Yeon J, Yang S-J and Kim C 2013 *J. of Semicond. Technol. and Sci.* **13** 465-472
[12] Hiremath M and Devi M 2013 *Int. J. of Res. in Eng. Technol. and Manag.* 1-8
[13] Rohith S and Pavithra S 2013 *Int. J. of Res. in Eng. and Technol.* **2** 209-214
[14] Mytsko, E., Malchukov, A., Novogilov, I., Kim, V. (2016) *Proceedings of 2016 International Siberian Conference on Control and Communications, MEACS 2016*, art. no. 7491748.
[15] Chen Y H, Chu C C and Yeh C C 2013 *Progr. In Electromagn. Res. Symp. Proc.* **1** 1091–1096
[16] Elharoussi M, Hamyani A and Belkasmi M 2013 *Int. J. of Adv. Computer Sci. and Appl.* **4** 33–37
[17] Singh A and Kaur M 2013 *Int. J. of Computer, Elect., Autom., Control and Inf. Eng.* **7** 1248–1250
[18] Dayal P and Patial R K 2013 *Int. J. of Computer Appl.* **68** 42–45
[19] Babreak V J and Sakhare S V 2014 *Int. J. of Computer Appl.* **87** 16–19
[20] Evgeniy, M., Andrey, M. Novogilov I and Kim V (2014) *Proceedings of 2014 International Conference on Mechanical Engineering, Automation and Control Systems, MEACS 2014*, art. no. 6986902