Pipelined Error-detecting Codes in FPGA Testing

Oleg BREKHOV, Maksim RATNIKOV
Moscow Aviation Institute (National Research University), 125993, Moscow, Russian Federation.
m.o.ratnikov@mail.ru

***Abstract*—This article approaches the solution of FPGA testing and research of characteristics at early development stages. The approach offers error-detection code based on universal test firmware. The performed test firmware based on CRC and Hamming codes detect single and multiple faults, and locate fault place (for Hamming code based test firmware).**

***Index Terms*—Field programmable gate arrays, Design for testability, Automatic testing, Cyclic redundancy check codes, Error correction codes**

I. Introduction

FPGA (Field-Programmable Gate Array) based system design ensures a variety of tasks among which the FPGA environment testing. At some stage, these tasks may require not only error detection but also the determination of the location of the error. Existing approaches do not guarantee identification of multiple failures, and do not allow accurate finding of a failure place. In addition, they require the creation of a separate firmware for each development stage and do not meet the scalability requirements.

In this paper, a new approach for creation of the test firmware base on pipelined error-detection code generator realization is proposed. It allows detecting failures and their occurrence place.

II. Requirements to the Test Firmware

We introduce the definitions of target firmware and test firmware. Test firmware is intended to perform a test or test group. Target firmware is intended to perform all functions that are specified in the assignment for the development of a system.

In addition, we introduce the definition of input vector concept. It is set of binary values transmitted on system inputs at this moment.

Test firmware is used in the following stages of the system testing:

- hardware platform selection;
- FPGA chips grading (selection the most suitable chips);
- FPGA environment testing (system board and its subsystems).

Existing approaches described in [1] and [2] have the following disadvantages:

- cannot guarantee detection of single or multiple failures;
- do not provide precise location of failure;
- require the development of the test firmware for each of the testing phases;
- do not guarantee the correct handling of multiple failures.

The test firmware must meet the following requirements:

1. being synthesizable – the functional description must precede synthesis, tracing and firmware generation successfully, i.e. ensuring FPGA platform restrictions;
2. fault sensitivity – providing maximum sensitivity of faults, including multiple faults;
3. scalability - allowing to change easily the degree of FPGA resources utilization;
4. predictability - values obtained as a result of testing device operations can be calculated in advance. These values can also be compared.

III. Pipelined Error-Detecting Codes Generators

More precisely, these requirements meet the pipeline computer systems. Each stage consists of register and logical elements that implement a selected function. These systems are scaled by varying the number of pipeline stages. Values of the FPGA inputs can be used as initial values. The regular structure is another advantage of pipeline computer systems. It simplifies the placing stage of FPGA development.

As basic test functions, generator error-detecting codes are proposed. The input data for these generators is the input data of the test system. Output data consists of error detection bits calculated for this input data. The result of pipeline's each stage operation is the intermediate value of error detection bits, which is computed for i bits of the input data stream. Here, i is the number of the current stage of the pipeline. In addition, intermediate values and signs that are used to perform the next step of the algorithm can be passed between stages.

The input bit array is called the input stream. We separate the concept of the reference input stream and the actual input stream. Reference input stream is the set of binary values, which were forwarded to the inputs of the test system or calculated as the result of self-test nodes correct work. The actual input stream is the set of binary values adopted by the device or received from internal hosts. The pipelined generator performs the calculation of the error detection code for the actual input stream. Failure occurrence causes the difference between reference input stream and actual input stream, or distort error detection bits. The combination of the reference input stream and the corresponding error detection bits is called the reference error detection code.

Reference input stream can be set constant or variable values. Constant values are set to the entrances of the test system before starting the test, and do not change before the end of the test.

The input stream from the variable values is an array of

Digital Object Identifier 10.4316/AECE.2014.02010

multiple input vectors. The new input vector is sent to the inputs of the system under test at each step. The result of system work is a set of error detection codes calculated for different input vectors. We consider that the input vector is sent to all inputs simultaneously, so the new input value will be obtained by all stages of the pipeline.

Let there be a reference input stream *S*. So, by definition:

$$S = \{V_1, V_2, ..., V_t\} \quad (1)$$

$V_1, V_2, ..., V_t$ - input vectors, and:

$$V_i = \{b_{i,1}, b_{i,2}, ..., b_{i,n}\} \quad (2)$$

where $b_{i,1}$, $b_{i,2}$, ..., $b_{i,n}$ are bits of the input vector V_i, i.e. these bits will be transmitted to the inputs of the system under test at i^{th} step.

Let *R* be the set of output vectors:

$$R = \{R_1, R_2, ..., R_t\} \quad (3)$$

and R_1, R_2, ..., R_t is the set of error detection codes in the corresponding moments of time. Then:

$$R_i = \{r_{i,1}, r_{i,2}, ..., r_{i,k}\} \quad (4)$$

where $r_{i,1}$, $r_{i,2}$, ... $r_{i,k}$ are error detection bits that make up the i^{th} error detection code.

We can say that:

$$R = F(s) \quad (5)$$

So, the test system is a pipeline and none of its stages has elements of long-term data storage (it is using data received from the inputs or calculated in the previous stage). R_i depends on all input vectors *V* that were transmitted to the entrances of the test system during *n* steps before the i^{th} step, and *n* is the number of pipeline stages. The last input vector used for R_i calculation has been read from the input for one step before R_i calculation has been finished. So:

$$R_i = F(V_{i-n-1}, V_{i-n}, ..., V_{i-1}) \quad (6)$$

At each step in the calculation of R_i only one bit of the input stream was used, and all intermediate values were calculated in the appropriate steps (number of intermediate value of R_i equal to the number of pipeline stage for which it was calculated). Thus, we have:

$$R_i = F(b_{i-n-1,1}, b_{i-n,2}, ..., b_{i-1,n}) \quad (7)$$

Bits of input array *S*, included in the V_i vectors and used in calculation of R_i are shown in Table I.

Any generation algorithm of the error detection codes working with the incoming input data stream can be used to solve the testing problem. Typically, these algorithms are cyclical and should be converted.

TABLE I. DYNAMIC INPUT STREAM PIPELINE'S INPUT DATA

	Input 1	Input 2	Input 3				Input n	
V_1	$b_{i-n-1\ 1}$							
V_2		$b_{i-n-1+1\ 2}$						
V_n							$b_{i-1\ n}$	
V_t								

The algorithm is converted into a pipeline structure, and each stage of the pipeline is allocated the corresponding bit of the input stream. The main element of the test system is the unit that implements the selected algorithm of the error-detection code generator.

For failure place detection, test firmware must rely on algorithms of self-corrected codes generator.

A. Algorithm of device testing

We determine the method of input values generation and value of the input stream bits. Flow chart of the device testing algorithm is shown in Figure 1.

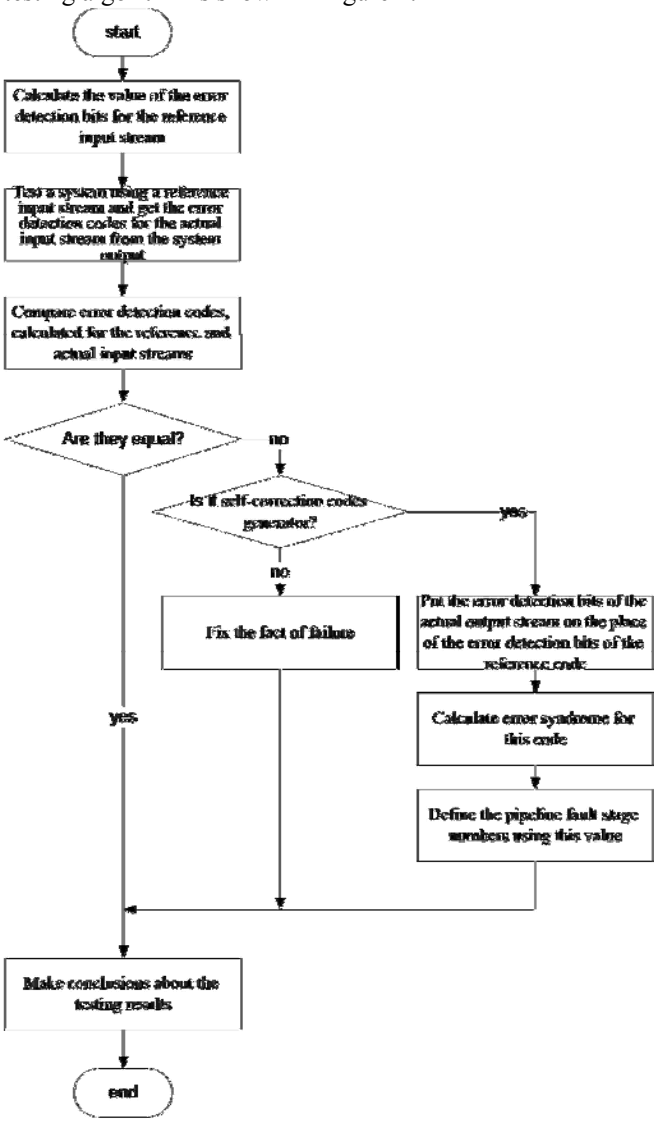

Figure 1. Algorithm of device testing.

General view of the test system is shown in Fig. 1

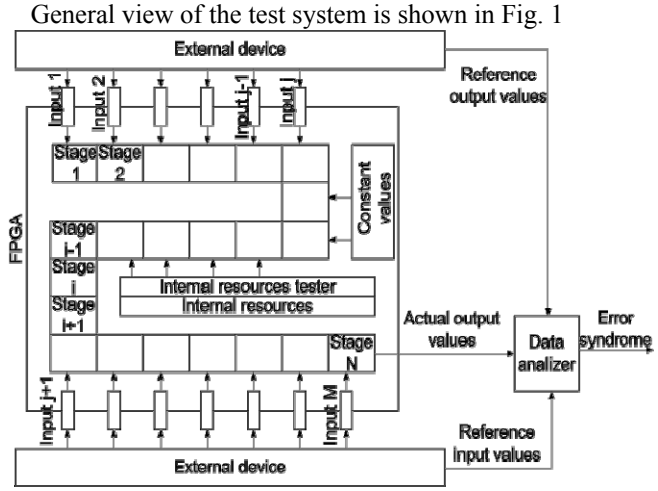

Figure 2. General view of test system.

Fig. 3 (A) and (B) show a general view of the reference and pipelined algorithms.

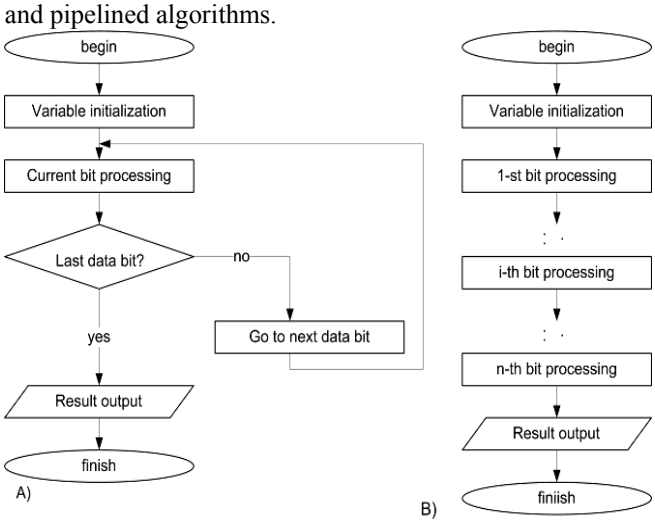

Figure 3. Reference (A) and pipelined (B) algorithms.

The block diagrams of devices, and the implementation of reference and pipelined algorithms are presented in Fig. 4 and Fig. 5, respectively.

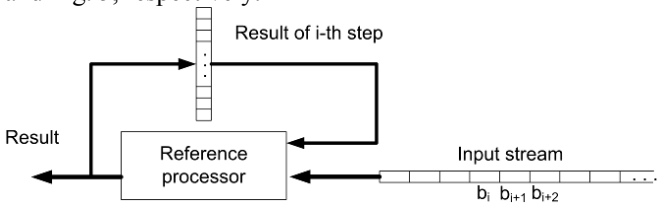

Figure 4. Reference algorithm realization.

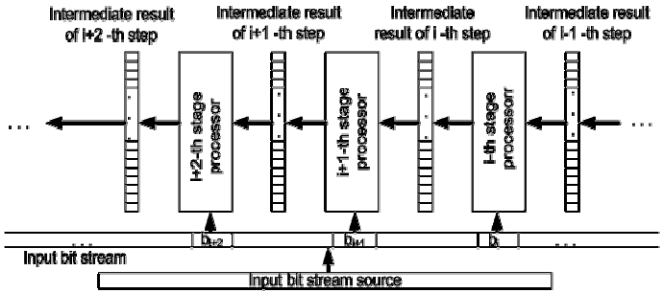

Figure 5. Pipelined algorithm realization.

Work cart of pipeline with constant input data is shown in Figure 6. This figure shows the beginning of the work process and error processing. The values of *R1…R5* are intermediate values, when working without failures. *R1…R5* values are the result of the work stages 1 to 5 in processing the error values. The last calculated value (here, it is *R5*) is transmitted to test system outputs. In case of normal operation, the *R5* value is set by the test system output, which is compared with a reference value.

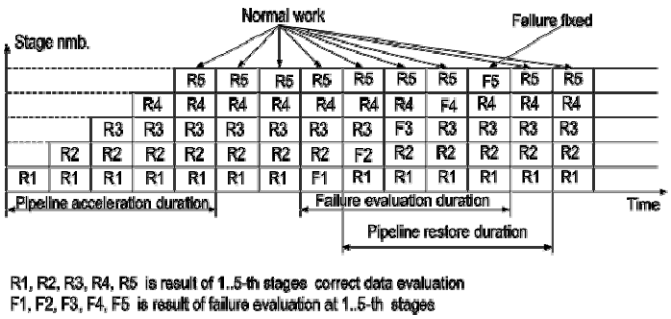

Figure 6. Timing diagram of the pipelined test system work.

If *R5* and reference values are equal, then we assume that no errors were detected in the tested system. If a system error occurs, the outputs will be set to 5 (not equal to *P5*), and such value does not match reference value. At the end of the external impact, pipeline work will be restored, and the outputs of the test system will be set to *R5* again.

B. Test system based on pipelined CRC-generator

Cyclic redundancy check (CRC) generators are often used for the failure detection during data transmission. The following example shows CRC-based test system creation. General scheme of the reference CRC generator is shown in Figure 7.

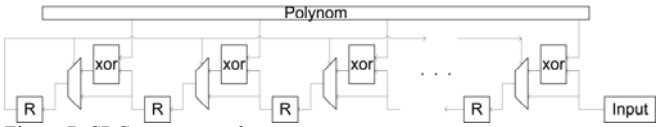

Figure 7. CRC generator scheme.

This scheme allows processing of the input data stream received from the block “input”. Current value of the CRC code is stored in the register formed by storage elements *R*. After receiving the next bit from the input stream, the register value is shifted to the left by one position. Value in upper bit of the register determines the action, which is done at this step (“shift left” or “XOR data in register and polynomial and shift left”). The value obtained from the input stream after the calculation is written in the lowest bit of the register.

FPGA testing requires simultaneous receiving of the current input vector bits at all stages of the pipeline. Test system should provide new error detection code at each step after the end of the pipeline acceleration.

Pipeline consists of the following stages: register (the length is equal to the degree of a polynomial), and XOR logical element.

The number of bits of the input stream is equal to the number of stages in the pipeline. At each step during operation, each stage (of the test system pipeline) receives an intermediate value of the error detection code from previous stage, and then executes the current action, defined by the intermediate value and the corresponding bit of the input stream. After that, this value is passed to the next stage. Stage block diagram is shown in Figure 8.

Since this scheme is an advanced implementation of the CRC generator, it preserves all properties of the basic CRC generator scheme. The result of the pipeline work is the correct value of CRC for the test system outputs.

During testing, the emergence of failure will lead to a significant change in CRC. Depending on the subsequent behavior, we can make conclusions about the duration and failure type.

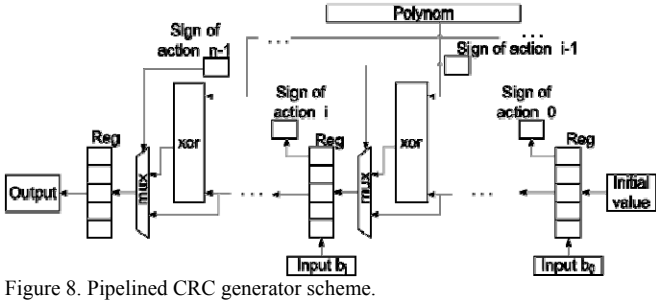

Figure 8. Pipelined CRC generator scheme.

C. Result of test CRC-based firmware implementation

This work was performed to implement the CRC-based test firmware. Source codes of the test system are written in the SystemVerilog 2005 (IEEE 1800-2005) HDL language. Results of synthesis obtained by Mentor Graphics Precision Synthesis are shown in Table II.

TABLE II. PIPELINED CRC GENERATOR SYNTHESIS RESULT

	Altera Stratix FPGA					**Xilinx Virtex FPGA**		
Stages	**Cell used**	**Freq (MHz)**	**nets**	**LCs**	**Inst.**	**LUTs**	**CLB**	**Freq (MHz)**
100	765	925.926	794	765	784	271	382	698.324
200	1565	819.672	1594	1565	1584	571	782	677.507
300	2365	819.672	2394	2365	2384	871	1182	663.570
400	3165	819.672	3194	3165	3184	1171	1582	663.570
500	3965	819.672	3994	3965	3984	1471	1982	663.570
600	4765	819.672	4794	4765	4784	1771	2382	663.570
700	5565	819.672	5594	5565	5584	2071	2782	663.570
800	6365	819.672	6394	6365	6384	2371	3182	663.570
900	7165	819.672	7194	7165	7184	2671	3582	663.570

Two FPGAs from different manufacturers obtained similar results: linear increase of the employed resources relative to the number of pipeline stages. The maximum system frequency is close to the maximum possible for the appropriate FPGA model. This confirms the fulfillment of the requirements for scalability.

D. Test system based on pipelined Hamming-codes generator.

Let us consider a system defining the place of failure. The basic algorithm is pipelined Hamming codes generator that detects two errors and is able to correct one error.

Each of the pipeline stages in this test system handles a single bit of the input stream. To ensure the minimal changes in the algorithm, and simplify the transformations, stages with numbers 0, 1, 2, 4…,2^n do not perform actions except for storing the data (these stages are registers).

The processing of each bit represents the addition modulo 2 of this bit, and error detection bits of code, from the same error detection group. The diagram of the test system is shown in Figure 9.

The diagram represents the alternation of the processing stages (generator), and non-processing stages (the register). Each of the processing stages is the addition modulo 2-bit input (b_i) with one or several error detection bits. The numbers of the error detection bits involved in the process depends on the stage number. For example, for the Hamming code *H (2, 1)*, the 1st and 2nd bits of the error detection code will be used at stage 3, the 1st and 3rd at stage 5, etc.

In accordance with the above formula, we have to calculate the error detection bits r_i to obtain the final value. This value is to be added to the module 2 to all data bits, related to j^{th} error detection group. The operation of addition modulo 2 has the property of associativity, so:

$$r_j = r_{j,k-1} \oplus b_{k,j}\,, \tag{8}$$

$$r_j = r_{j,k-2} \oplus b_{k-1,j} \oplus b_{k,j} \tag{9}$$

$$r_j = r_{j,1} \oplus b_{2,j} \oplus ... \oplus b_{k-1,j} \oplus b_{k,j} \tag{10}$$

$r_{j,1}$... $r_{j,k-1}$ are intermediate results of addition modulo 2 information bits j, the error detection group. In this case, at each processing stage of the pipeline calculation, the next value r_j can be implemented.

Let R be the set of error detection bits in this stage. Then:

$$R = (r_1, r_2, ..., r_k) \tag{11}$$

where k is the number of error detection bits (error detection groups). In fact, R_i is a set of error detection bits for the input stream length i.

As shown above, due to the property of associativity of operation "addition modulo 2", each of the following R can be calculated based on the previous values. In this case, for each set of error detection bits R_i at the i^{th} stage of the pipeline, we have the following:

$$R_i = H(R_{i-1}, i, b_i) \tag{12}$$

where H is the function that computes the next set of error detection bits of Hamming code, i is the number of stages, and b_i is the i^{th} bit of the input stream. Then the task of converting the algorithm for computing the Hamming code is to get H-function. In accordance with the algorithm of Hamming code calculating, each error detection bit is the result of addition modulo 2 of all data bits of the corresponding error detection group. Then, from (4), (5) and (12), we can conclude that:

$$r_{j,l} = h(r_{j,l-1}, l, b_i) \tag{13}$$

So, an error detection bit of j^{th} error detection group at stage l depends on error detection bits of the same error detection group at stage *l-1*, stage number, and the appropriate bit of the input stream.

Since calculation of the current value of $r_{j\,t}$ uses bits only from error detection group j, at processing stage l, we have:

$$r_{j,l} = \begin{cases} r_{j,l-1}, l \notin J \\ r_{j,l-1} \oplus b_l, l \in J \end{cases} \tag{14}$$

In this formula, J is a set of the elements j^{th} of the error detection group. In Hamming codes algorithm, each error detection bit covers all bits, where the bitwise AND of the error detection position and the bit position are non-zero. During $r_{j,t}$ calculation, we will check the value of the j^{th} bit at binary-coded stage number.

Non-processing stage ensures the correctness of the calculation of the error detection code, and it is does not perform any calculations. Numbers of non-processing stages are 1, 2, 4, 8..2^m. For non-processing stages, we have:

$$r_{j,l} = \begin{cases} 0, l \in J, l = J[0] \\ r_{j,l-1}, l \notin J[0] \end{cases} \tag{15}$$

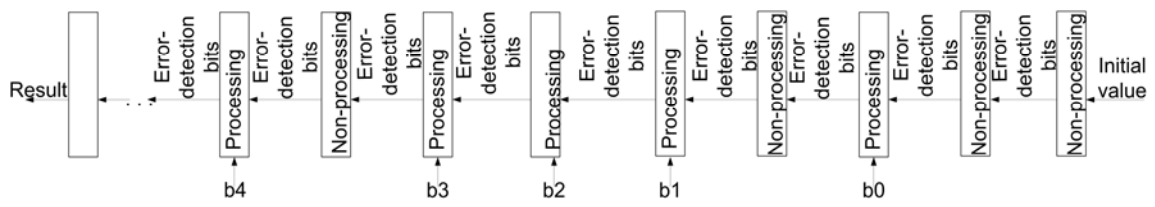

Figure 9. Structural diagram of the Hamming code-based test system.

So, at this stage, a new error detection group calculation begins and error detection bits from this error detection group were not previously used. Initial value of the error detection bit is 0.

For double failure evaluation, an overall error detection bit has been added. It checks parity of the overall error detection code and indicates (but does not correct) double failures.

The diagram of the pipeline stage (processing stage) is shown in Figure 10

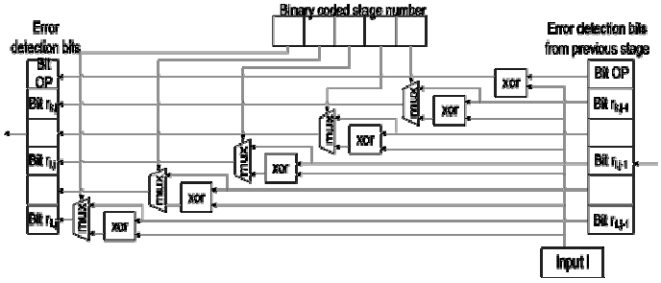

Figure 10. Pipelined Hamming code generator stage.

Stage number is a constant. DC bit is the overall parity bit for this Hamming code.

Here is an example: set the input stream 10111001. In this case, stage calculation results are the values shown in the first row of Table. If failure is caused during system test operation, the 3rd bit of input stream changes to 1. Stage calculation results are shown in the second row of Table. When the correction of the failure ends, pipeline operation will be restored. These values are shown in the third row of Table. The following example shows output value calculation.

Stage 1

It is a non-processing stage. First error detection bit has 0 value. At this stage, we do not know values of other error detection bits and suppose that they are equal to 0.

Stage 2

Similarly with the stage 1, it is a non-processing stage. Second error detection bit has value 0.

Stage 3

It is the first processing stage. Input bit value is 1. Binary coded stage number is 00011, so:

$$r_{1,3} = 0 \oplus 1 = 1$$

$$r_{2,3} = 0 \oplus 1 = 1$$

Overall parity bit is:

$$r_{OP,3} = 0 \oplus 1 = 1$$

Values of the 3rd and 4th error detection bits were not changed.

Stage 4

It is a non-processing stage. Fourth error detection bit has value 0.

Stage 5

It is the second processing stage. Input bit value is 1. Binary coded stage number is 00101, so:

$$r_{1,5} = 1 \oplus 0 = 1$$

$$r_{2,5} = 1 = 1$$

$$r_{3,5} = 0 \oplus 0 = 0$$

$$r_{4,5} = 0 = 0$$

$$r_{OP,5} = 1 \oplus 0 = 1$$

After making similar calculations for the stages 6-12, calculations for changed input stream and for restored input stream, we can fill in Table III.

TABLE III. INTERMEDIATE VALUE OF THE ERROR DETECTION BITS AT EACH STAGES OUTPUT

Stage number	Input bits before failure	Error detection bits before failure	Input bits after failure	Error detection bits after failure	Input bits after restore	Error detection bits after restore
$1_{10} = (00001)_2$	-	00000	-	00000	-	00000
$2_{10} = (00010)_2$	-	00000	-	00000	-	00000
$3_{10} = (00011)_2$	1	10011	1	10011	1	10011
$4_{10} = (00100)_2$	-	10011	-	10011	-	10011
$5_{10} = (00101)_2$	0	10011	0	10011	0	10011
$6_{10} = (00110)_2$	0	10011	1	00101	0	10011
$7_{10} = (00111)_2$	1	00100	1	10010	1	00100
$8_{10} = (01000)_2$	-	00100	-	10010	-	00100
$9_{10} = (01001)_2$	1	11101	1	01011	1	11101
$10_{10} = (01010)_2$	1	00111	1	10001	1	00111
$11_{10} = (01011)_2$	0	00111	0	10001	0	00111
$12_{10} = (01100)_2$	1	11011	1	01101	1	11011

Note: overall parity bit in this table (calculated for input stream values only) does not consider error detection bits. Calculation of the error detection bit is not executed because the values at all stages (except the final one) are intermediate. The double-check bit (which is the part of error detection bits) is calculated for all bits of the actual input stream (excluding error detection bits itself). Before Hamming code decoding, it is necessary to update the overall parity bit value considering the final error detection code value. Therefore, for example, in Table III, the result values of overall parity bits are:

- Before failure - «0»
- After failure - «1»
- After restore - «0».

E. Result of CRC-based firmware implementation test

The source codes of the test system are written in SystemVerilog 2005 (IEEE 1800-2005) HDL language. The timing diagram for constant input stream with failure evaluation and restoring is shown in Figure 11.

Testing system consists of:

- module that implements a single stage of a testing pipeline;
- testing pipeline itself (predefined number of connected modules);
- it can also be supplemented by the nodes that implement checks of inner FPGA subsystems or realize any additional processing of input values.

F. Detecting a place of failure

To solve this problem, we replace the error detection code in the reference code with the Hamming code generated by the test system. After that, we calculate the error syndrome and the number of the segment where the failure occurred.

Reference Hamming code for this input stream is 0101111000111. After replacing error detection code, the result code is 101111001101. Error syndrome calculation gives the value 00110. It means that an error occurred in the 6th Hamming code bit, that corresponds to 3rd information (input stream) bit.

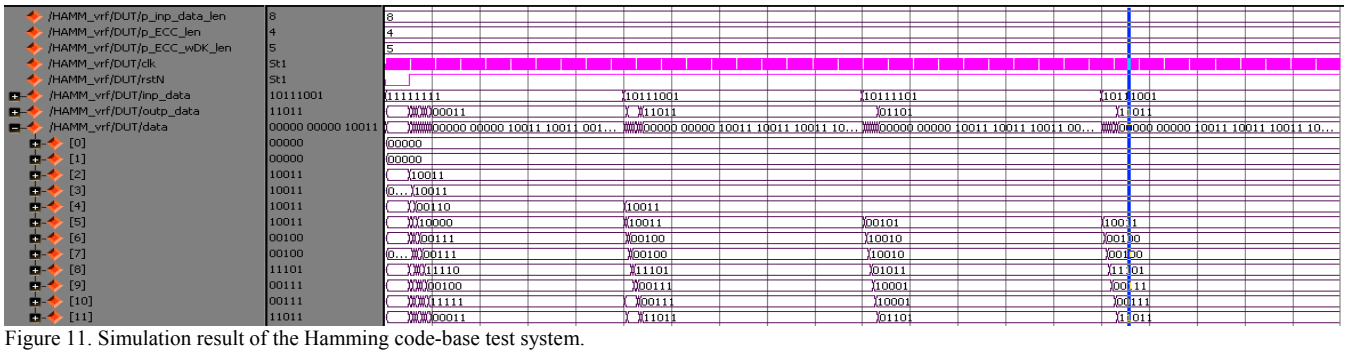

Figure 11. Simulation result of the Hamming code-base test system.

Synthesis result for FPGA Actel APA 1000 is shown in Table IV.

TABLE IIV. PIPELINED HAMMING CODE GENERATOR SYNTHESIS RESULT

Stages	Tiles	Resource used %	Freq (MHz)
100	1629	2.89	336.36
200	3695	6.56	336.36
300	5948	10.56	336.36
400	8313	14.76	336.36
500	10774	19.13	336.36
600	13305	23.62	336.36
700	15912	28.25	336.36
800	18551	32.94	336.36
900	21224	37.68	336.36

Similar to pipelined CRC-based test systems, these test systems obtain linear increase of employed resources relative to the number of pipeline stages. The maximum system frequency is close to the maximum possible for this FPGA. This confirms the fulfillment of the requirements for scalability.

Testing has shown that test system confirms requirement of failure detection and localization.

IV. CONCLUSION

In this paper, we proposed and implemented a method for testing the FPGA-based data storage and data processing systems using pipelined error detection code generators. Compared with conventional testing methods, this method is more flexible and more sensitive to single and multiple failures. In addition, this method allows not only to detect errors, but also to define the place of their occurrence.

REFERENCES

[1] B. Pratt, M. Caffrey, P. Graham, K. Morgan, M. Wirthlin, "Improving FPGA Design Robustness with Partial TMR," IRPS 2006.

[2] D.V. Bobrovsky, O.A. Kalashnikov, P.V.Nekrasov, "Functional Control Technique for FPGA Total Ionizing Dose Testing," Proceedings of the Conference RADECS-2012.

[3] R.N. Williams. A Painless Guide to CRC Error Detection Algorithms. Rocksoft Pty Ltd., Australia, 1993.

[4] P.P. Shirvani, E.J. McCluskey, "Fault-Tolerant Systems in a Space Environment: The CRC ARGOS Project," CRC Technical Report No. 98-2 (CSL TR No. 98-774), December 1998, Center For Reliable Computing, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, California 94305.

[5] IEEE Standard for SystemVerilog— Unified Hardware Design, Specification, and Verification Language, The Institute of Electrical and Electronics Engineers, Inc. 3 Park Avenue, New York, NY 10016-5997, USA, 2005.

[6] K. Arshak, E. Jafer, C. Ibala, "Testing FPGA based digital system using XILINX ChipScope logic analyzer," in Electronics Technology. ISSE '06, pp. 355-360, May 10-14, 2006. doi: 10.1109/ISSE.2006.365129.

[7] K.S. Morgan, D.E. Johnson, B.H. Pratt, M.J. Wirthlin, M.P. Caffrey, P.S. Graham, "SEU Induced Error Propagation in FPGAs," in Proceedings of NSREC Conference, Seattle, WA, July 11-15, 2005, Brigham Young University, 459 CB Provo, UT 84602, Los Alamos National Laboratory, Los Alamos, NM 87545.

[8] H.H. Schmit, S. Cadamni, M. Moe, S.C. Goldstein, "Pipeline Reconfigurable FPGAs," Journal of VLSI Signal Processing Systems 24, Kluwer Academic Publishers, pp. 129–146, 2000.

[9] M. Abramovici, C.E. Stroud, "BIST-Based Delay-Fault Testing in FPGAs," in Journal of Electronic Testing: Theory and Applications archive, Volume 19 Issue 5, October 2003, pp. 549-558, Kluwer Academic Publishers Norwell, MA, USA

[10] M. B. Tahoori, "Application-Dependent Diagnosis of FPGAs," in Proceedings of ITC 2004, pp. 645-654, Oct 26-28, 2004, doi: 10.1109/TEST.2004.1387002

[11] B.F. Dutton, C.E. Stroud, "Built-In Self-Test of Configurable Logic Blocks in Virtex-5 FPGAs," in Proceedings of 41st Southeastern Symposium on System Theory, pp. 230-234, 2009.

[12] I.G. Harris, Russell Tessier, "Testing and Diagnosis of Interconnect Faults in Cluster-Based FPGA Architectures," in ICCAD'00 Proceedings of 2000 IEEE/ACM International Conference on Computer-aided design, pp. 472-476, IEEE Press Piscataway, NJ, USA ©2000.

[13] M.Latha, M.Senthilmurugan, "Fault Detection and Fault Diagnosis in SRAM-Based FPGA Using BIST," in IRACST – Engineering Science and Technology: An International Journal (ESTIJ), ISSN: 2250-3498, Vol.2, No. 4, August 2012 609

[14] A. Sarvi, C.A. Sharma, R.F. DeMara, "Bist-Based Group Testing for Diagnosis of Embedded FPGA Cores," ESA, pp. 279-283, CSREA Press, 2008.

[15] Dr.K.Babulu, M.K. Kumar, "FPGA Realization of Multiple Fault Diagnosis Technique for Faults in SRAM Based FPGAs," in International Journal of Engineering Science and Innovative Technology (IJESIT), Volume 1, Issue 1, September 2012, 48, ISSN: 2319

[16] M. Rozkovec, J. Jenicheck, Z. Pliva, "Using deterministic test vectors to test FPGA circuit," Proceedings of the 2013 IEEE 16th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), pp. 175-180, 2013.

[17] F. Noorbasha, K. Harikishore, Ch. Hemanth, A. Sivasairam, V. Vijaya Raju, "LFSR Test Pattern For Fault Detection and Diagnosis for FPGA CLB Cells," International Journal of Advances in Engineering & Technology, ISSN: 2231-1963, 240, Vol. 3, Issue 1, pp. 240-246, March 2012.

[18] C.-F. Wu, C.-W. Wu, "Testing and Diagnosing Dynamic Reconfigurable FPGA," VLSI Design, Volume 10, Issue 3, pp. 321-333, 2000.

[19] M.G. Gericota , G.R. Alves , M.L. Silva , J.M. Ferreira, "Active Replication: Towards a Truly SRAM-Based FPGA On-Line Concurrent Testing," Proceedings of the Proceedings of The Eighth IEEE International On-Line Testing Workshop (IOLTW'02), p.165, July 08-10, 2002

[20] Y.-B. Liao, P. Li, A.-W. Ruan, Y.-W. Wang, W.-C. Li, "A HW/SW Co-Verification Technique for FPGA Test," Journal of Electronic Science and Technology of China, Vol. 7, No. 4, 390, December 2009.

[21] Y.-C. Chiu, B. Tarun, S. Ye, P.-I Yeh, P.-A. Shen, "The FPGA test system. Project Final Report," Group 5, University of Southern California, December 2010.