Hardware Implementation of Self Checking Circuits on FPGA

Sakshi Gupta, Vignesh Chandrasekaran, Chandru Loganathan

Department of Electrical and Computer Engineering
University of Wisconsin - Madison
Madison, USA
{sgupta54, vchandrasek3, cloganathan}@wisc.edu

***Abstract*- Project aims at exploring and implementing different self-checking circuits on FPGA with the intention of reconfiguring the hardware to prolong system halt. In a system with hardware redundant function blocks, the concept of reconfiguration logic tries to detect the faulty block, isolate the faulty block and ensure that the correct output is still produced with the non-faulty function blocks. Implemented self-checking circuits include Residue checker, Less than or equal to checker (LTOETC) which was extended to implement the Range checker. Based on the synthesis report, the hardware utilization was analyzed for all the circuits and for the residue checker, its hardware complexity was computed as function of modulo.**

***Keywords*- Self-checking, fault-tolerance, sorting, FPGA, reconfiguration, range checker, residue checker.**

I. INTRODUCTION

A fault-tolerant system is one that continues to perform at a desired level of service in spite of failures in some components that constitute the system. In the recent years, a considerable amount of work has been done in the field of fault-tolerant computing, especially in the area of self-checking checkers for detecting various types of errors. A circuit is said to be fault secure if in the presence of a fault, the output is either always correct, or not a code word for valid input code words. A circuit is said to be self-testing if only valid inputs can be used to test it for faults. A circuit is said to be totally self-checking if it is both fault secure and self-testing.

In a Totally Self-checking Circuit (TSC), if there is a fault in the inputs and/or within the TSC itself, the system no longer functions as desired. So, in order to make the system tolerate faults in the input we propose a reconfiguration logic which allows the system to function properly in presence of at most two faults, except when both the faults occur within the same input pair which cannot be detected, i.e. if the fault occurs in both x_0 as well as x_1 or y_0 as well as y_1, then it is not possible to reconfigure in such cases. There are other applications of TSC, for example, sorting checker such as residue checker which compares the residue of the implemented function, LTOETC (Less than or equal to checker) which checks if the series of inputs are sorted. LTOETC can be further modified to design a range checker which checks if the input lies in a particular range.

The paper is organized as follows. In Section II, we discuss a TSC implementation with reconfiguration logic. In Section III, we discuss about the LTOETC. In Section IV, we discuss about the Range checker which is followed by the discussion of Residue Checker in Section V. Finally, results and conclusion are presented in Section VI and VII respectively.

II. TSC WITH RECONFIGURATION LOGIC

A circuit is said to be totally self-checking if it is both fault secure and self-testing as mentioned before. A TSC should have two outputs and, hence four output combinations are possible (00, 01, 10, 11) as shown in Fig. 1. Two out of the four output combinations are considered valid (01, 10). A non-valid output from the TSC (00, 11), indicates either a non-code word at the input of the TSC or a fault in the TSC itself. The main reason for TSC needing two outputs is because if there is only one output, and the valid output value is, for instance 1, then a stuck-at-1 fault at the output cannot be detected during normal operation. Also, output combinations (00, 11) are not considered as valid because a unidirectional multi-bit error may change 00 to 11 and vice-versa.

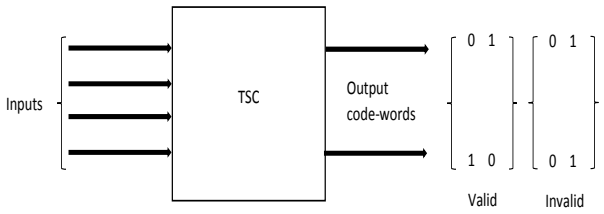

Fig. 1 Totally Self-checking

In our implementation, the TSC is a two-rail checker circuit shown in Fig. 2. The two-rail checker two groups of inputs (x_0, y_0) and (x_1, y_1) where y_0 is the complementary of x_0 and y_1 is the complementary of x_1, and two outputs f and g. In a non-error situation when $x_0x_1 = 11$ and $y_0y_1 = 00$, the output of the circuit is f = 0 and g = 1. Now due to a fault suppose $y_0y_1 = 10$, then the output of the circuit becomes f = g = 1, a non-code word output thus indicating an error. This circuit is totally-self checking for all single and unidirectional

multiple errors. For an arbitrary number of bits, TSC can be designed by using two-level AND-OR gates. For example, a 3-bit TSC is shown in Fig. 3. An n-bit TSC can implemented in a similar manner. For our design, we consider a two input pair TSC with each input having a width of 16-bit.

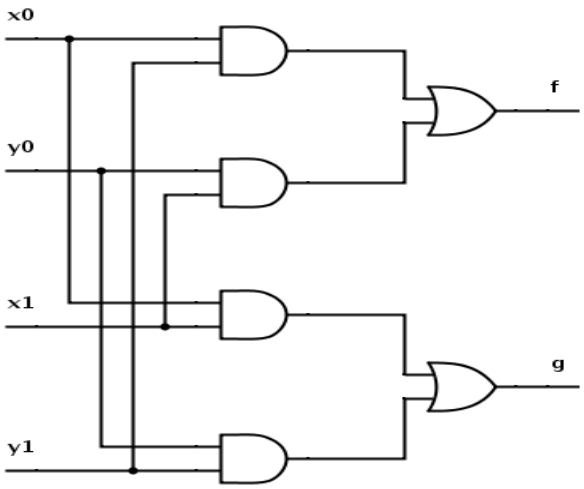

Fig. 2 Two-rail self-checker circuit

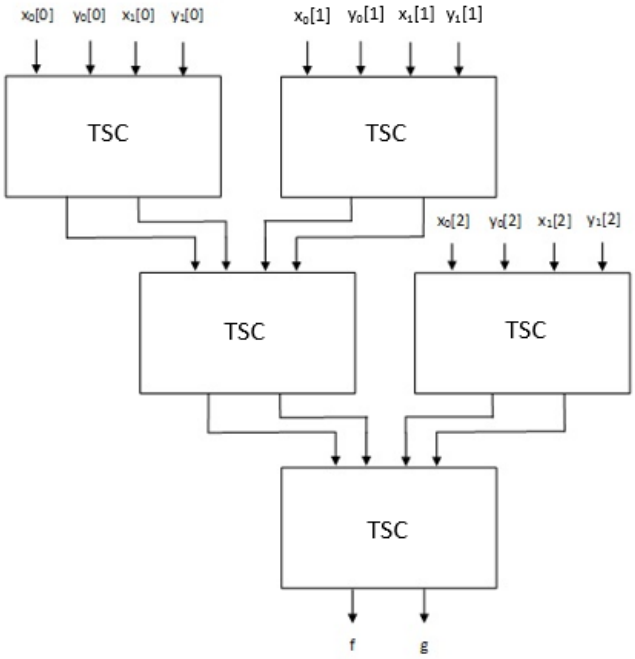

Fig. 3 3-bit TSC circuit

Each of the inputs for the TSC are generated independently using different architectures to arrive at the same expression rather than merely making them hardware redundant with the same architecture. This has been discussed in [8].

If suppose a fault occurs in the inputs and/or within the TSC itself, then the system no longer functions as desired. In order to make the system tolerate faults in the input we propose a reconfiguration logic which allows the system to function in the presence of at most two faults, except when both the faults occur within the same input pairs. This can be well understood by an example, consider that the four inputs to the TSC are $x_0x_1 = 11$ and $y_0y_1 = 00$, which produces an output f = 0 and g = 1. Now due to a fault (in y_0), $y_0y_1 = 10$, the output becomes f = g = 1, indicating a fault. This non-code word output triggers the reconfiguration logic, which then compares the four inputs, which is merely an Exclusive-OR (XOR) operation between all the input combinations and, finds which input is faulty, i.e., y_0 in this case. Then, the new y_0 value is computed from the non-faulty input, i.e., $y_0 = 0$. These new set of inputs are then fed to the system whose inputs were faulty, and were detected by the TSC. The reconfiguration logic can detect all single faults and two faults (except when both the faults are in the same input pair i.e., x_0y_0 or x_1y_1) in the inputs and correct them. The algorithm followed in detecting and isolating the faulty units is depicted in Fig. 4.

```
while reconfiguration logic enable
        if one input in pair 0 (i.e., x0 or y0) fails
                if x0 fails
                        Compute new x0 from x1;
                else if y0 fails
                        Compute new y0 from y1;
        else if one input in pair 1 (i.e., x1 or y1) fails
                if x1 fails
                        Compute new x1 from x0;
                else if y1 fails
                        Compute new y1 from y0;
        else if both x0 and x1 fails
                Compute new x from complement of y0;
        else if both y0 and y1 fails
                Compute new y from complement of x0;
        else if both inputs in pair 0 or in pair 1 fails
                Enable error flag
```

Fig. 4 Algorithm followed in reconfiguration logic

III. LTOETC (LESS THAN OR EQUAL TO CHECKER):

Sorting plays an important role in data processing as well as digital signal/image processing, and there is a huge focus on the development and analysis of sorting algorithms. When a failure occurs in a processing element of a VLSI sorter, functional error may occur where the operands are incorrectly ordered but the value of each operand is not changed.

Fig. 9 shows the sequence N_1, N_2, N_3, N_4, N_5 where each number is a non-negative number. The input code space includes all sequences that are in non-increasing order and the input non-code space includes all sequences that are not in non-increasing order. This has been discussed in [1].

LTOETC consists of (k-1) bit comparators, inverters, exclusive-or gates, TSC checkers for two-part two-rail code, translator and 1-out-of-4 generator. If $(x_2... x_k) >= (y_2... y_k)$ then output of the comparator $co_1 = 1$ & $co_2 = 1$ else $co_1 = 0$ and $co_2 = 0$.

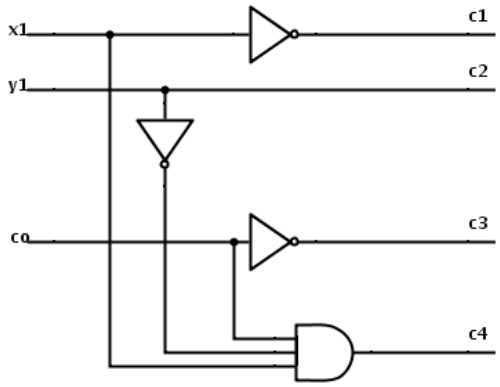

Fig. 5 Generator

In order to prove LTOETC is a self-testing checker, we need to prove that it's code-disjoint and self-testing.

A. LTOETC is code-disjoint

Since inputs to LTOETC are sorted binary operands where $(x_1, x_2...x_k) >= (y_1, y_2... y_k)$ is always true during normal operation. Let $X_1 = (x_2... x_k)$ and $Y_1 = (y_2... y_k)$. The input code space includes:

a) $x_1 y_1 = 00$ and $X_1 \geq Y_1$
b) $x_1 y_1 = 11$ and $X_1 \geq Y_1$
c) $x_1 y_1 = 10$ and $X_1 \geq Y_1$
d) $x_1 y_1 = 10$ and $X_1 < Y_1$

Table I lists the values of o_1o_2 obtained from Fig. 5, 6 and 8. As the output for each of these input code spaces are distinct, we conclude that LTOETC is code-disjoint.

B. LTOETC is self-testing circuit

1) Each functional block receives all necessary test vectors so that it is fully tested during normal operation.
   - TSC checkers for the two-pair two-rail code receive 0011, 0110, 1001, 1100
   - Translator receives 0001, 0010, 0100, 1000
   - Each inverter receives 0 or 1
   - Each XOR gate receives three vectors from the set {00, 01, 10, 11}.
   - 3-input AND gate in the generator block receives 110, 101, 011, 111
   - (k-1) bit comparators receive all possible input combinations.
2) When a fault in each functional block is excited, a non-code word will be generated at the outputs o_1, o_2.
   - If a fault occurs in generator, translator, TSC_1, TSC_2, one of XOR gates and one of the inverters shown in Fig. 8, then a test vector will generate 00 or 11 either at $a_{31}b_{31}$ or $a_{32}b_{32}$ (not both). Hence o_1o_2 become either 00 or 11.

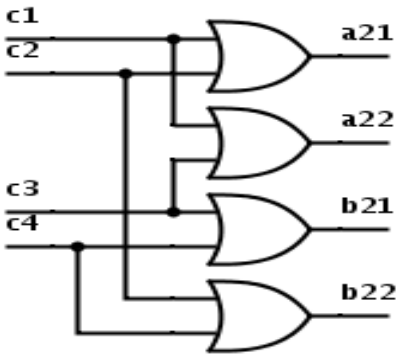

Fig. 6 Translator

- If a fault occurs in TSC_3, then a test vector will generate either 00 or 11.
- If a fault occurs in either of the (k-1)-bit comparators, then a test vector will generate 00 or 11 only at $a_{31}b_{31}$ or $a_{32}b_{32}$. Hence, o_1o_2 become either 00 or 11.

LTOETC compares two numbers. We can detect if more than two numbers are sorted correctly using multiple such LTOETC blocks. Fig. 9 shows one example where 5 numbers, i.e., N_1, N_2, N_3, N_4 and N_5 are checked if they are arranged in a sorted sequence. If $N_1 < N_2$ or $N_2 < N_3$ or $N_3 < N_4$ or $N_4 < N_5$, output o_1o_2 of this circuit would be 11, otherwise the circuit will generate valid code words.

IV. RANGE CHECKER

LTOETC is used as the building block for implementing the range checker, which can be used to check if a number lies within a range. If the number does not lie within the given range, then the outputs o_1o_2 will generate 11 otherwise it will generate valid code words, i.e. 01 or 10. Fig. 7 illustrates the implementation of the range checker which requires two LTOETC blocks and a TSC. Upper bound of the range is specified as the input to the first LTOETC and the lower bound of the range is specified as the input to the second LTOETC. The number which needs to be checked, if it lies within the range, is fed as the second input to both the LTOETC blocks.

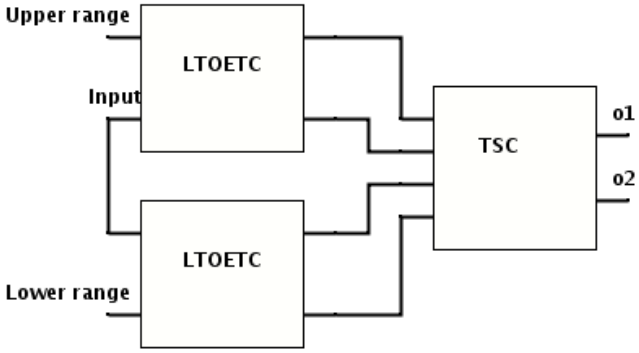

Fig. 7 Non-increasing sorting checker

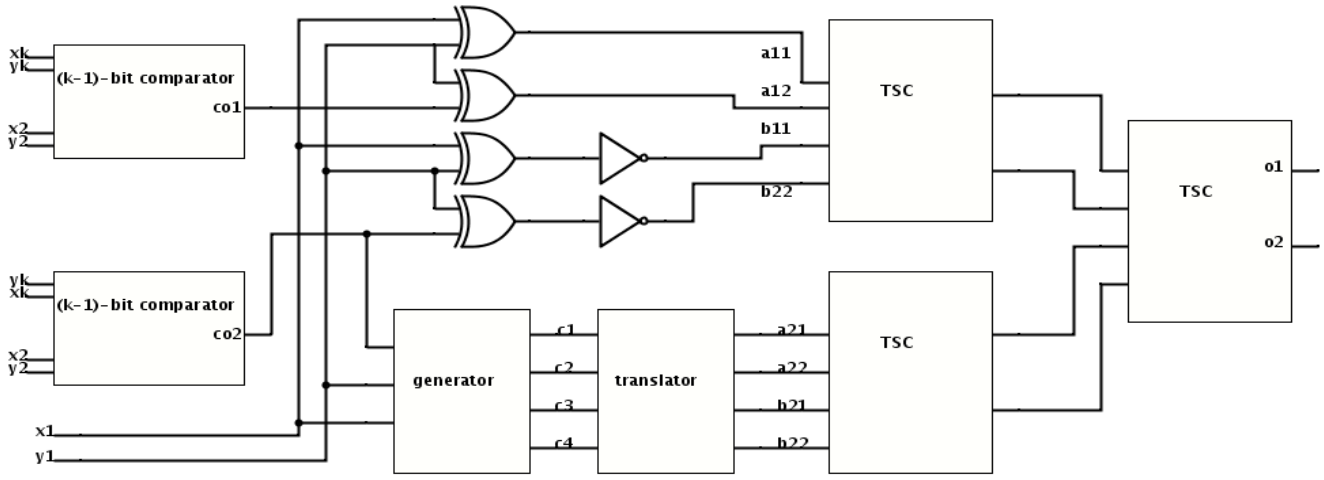

Fig. 8 Less than or equal to checker

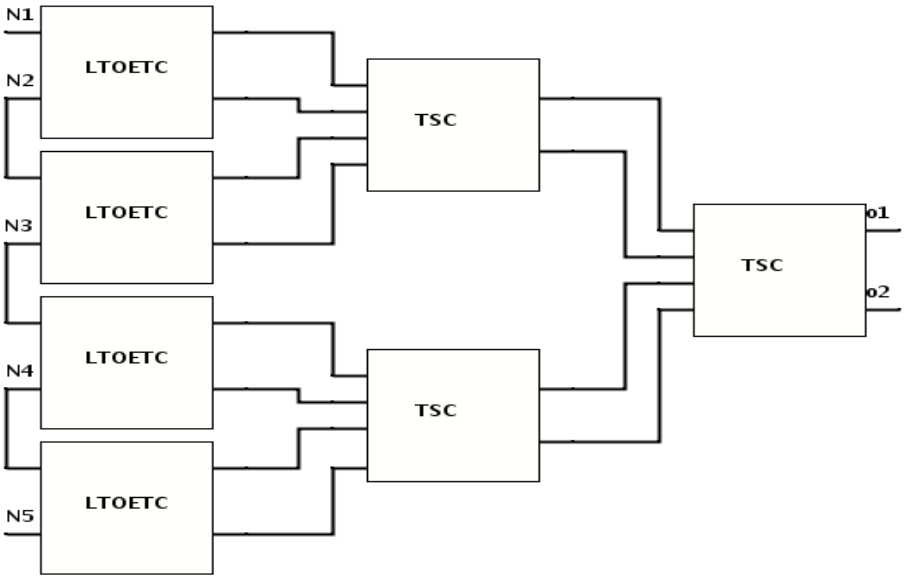

Fig. 9 Non-increasing sorthing checker with 5 inputs

TABLE I

INPUT/OUTPUT RELATIONSHIP OF FUNCTIONAL BLOCKS

x_1 y_1	(X1, Y1)	co_1 co_2	$a_{11}a_{12}b_{11}b_{12}$	$c_1c_2c_3c_4$	$a_{21}a_{22}b_{21}b_{22}$	$a_{31}a_{32}b_{31}b_{32}$	o_1 o_2
00	X1 ≥ Y1	11	0110	1000	1100	0110	01
11	X1 ≥ Y1	11	0011	0100	1001	1001	01
10	X1 ≥ Y1	11	1100	0001	0011	1100	10
10	X1 < Y1	00	1001	0010	0110	0011	10
00	X1 < Y1	00	0011	1010	1110	1101	11
11	X1 < Y1	00	0110	0110	1111	0111	11
01	X1 ≥ Y1	00	1001	1100	1101	0111	11
01	X1 < Y1	00	1100	1110	1111	1101	11

V. Residue Checker

Residue checker is based on the idea of computing the residue of a given function for a given modulo and comparing it against the residue obtained by computing the same function broken down by modulo arithmetic [2].

Consider a multiple-accumulate (MAC) unit of a processor which computes the following function:

$$Z = A*B+C$$

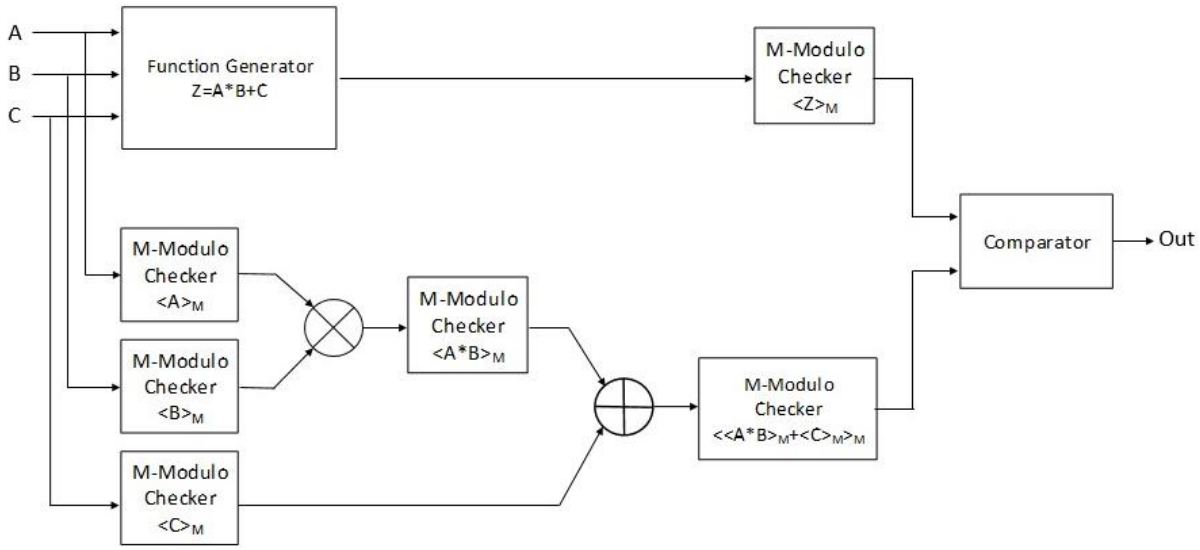

Fig. 10 Residue checker

The first modulo checker circuit computes the residue for this entire function as $\langle Z \rangle_{m.}$ Now, the second modulo checker computes the residue for this function based on the properties of modulo arithmetic.

Property 1: $\langle X+Y \rangle_m = \langle \langle X \rangle_m + \langle Y \rangle_m \rangle_m$
Property 2: $\langle X.Y \rangle_m = \langle \langle X \rangle_m * \langle Y \rangle_m \rangle_m$

Based on the above properties the second modulo checker will evaluate the function Z by converting each of its operands into residue numbers, in the following manner:

$$\langle Z' \rangle_m = \langle\langle\langle A \rangle_m * \langle B \rangle_m \rangle_m + \langle C \rangle_m \rangle_m$$

Once the residue $\langle Z \rangle_m$ and $\langle Z' \rangle_m$ is obtained, these values are sent to the comparator which generates an error signal if they are unequal. This entire design relies on the reliability of the comparator as that becomes as a single point of failure. This functionality has been illustrated in Fig. 10.

Modulo checkers can be implemented in several ways in hardware. The most common method that is adopted for computing modulo of 2^n is bit selection for the lower *n* bits of the applied input. For example, modulo 8 for a given input can be computed by bit selecting 3 bits of its LSB. As this method involves merely bit selection, it is very hardware conservative. This design was implemented on FPGA for different modulo values of 2^n, $\forall\, n \in (1,16)$. Synthesis report for the design for various modulo values were analyzed by considering the slice LUTs utilized as the parameter for hardware utilization. The plot depicting hardware utilization as a function of modulo is illustrated in Fig. 11. This plot was curve fitted to obtain the equation for hardware complexity as a function of modulo which was found to be a logarithmic curve with the following equation, units for which would be the number of slice LUTs utilized on the target FPGA.

$$HC(m) = 4.2814\ln(m) + 5.275$$

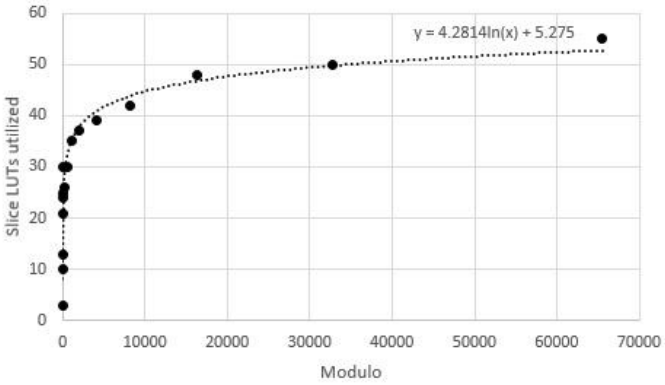

Fig. 11 Plot illustrating hardware utilization vs Modulo for 2^n modulo checker.

The second method computes the residue for any value of modulo, therefore this demands more hardware as oppose to the earlier method. This design incorporates two counters, the first counter is initialized to the input's value and then decremented in each iteration. The second counter is incremented in each iteration and gets reset when it equals the value of modulo. This design was implemented on FPGA for several values of modulo *n*, where $n \in (2, 65536)$. Hardware utilization plot was plotted using the data obtained from the

synthesis report which turned out to be a logarithmic curve and is illustrated in Fig. 12. Hardware complexity for this design is computed by curve fitting this plot which yields the following equation:

$$HC(m) = 27.319\ln(m) + 186.95$$

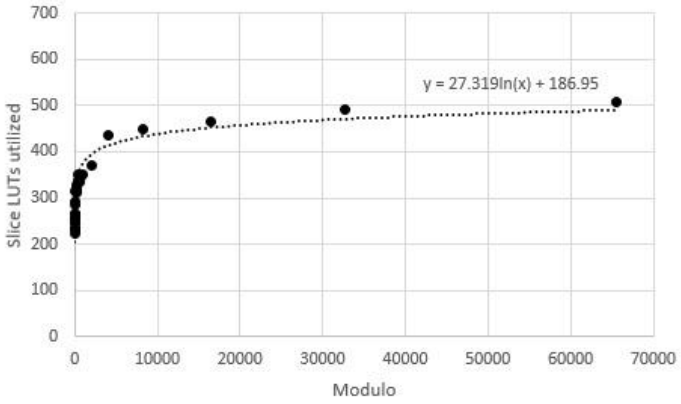

Fig. 12 Plot illustrating hardware utilization vs Modulo for any value modulo checker.

VI. Results

The self-checking circuits were implemented on Xilinx Virtex 5 XUPV5-LX110T FPGA with the speed grade of -1. The circuits were implemented by coding in Verilog HDL using gate level modelling and behavioral modelling. The faults were induced by creating modules to intentionally corrupt a single bit or a stream of bits on a bus. ChipScope Pro was used to monitor and analyze waveforms obtained during runtime and faults were induced to check the functionality of the reconfiguration logic.

The following results were observed while implementing the less than equal to checker or the non-increasing sorting checker and the range checker. Less than or equal to checkers are used as building blocks in constructing Non-increasing sorting checker which in turn is used as the building block for implementing the Range checker.

TABLE II

HARDWARE UTILIZATION AND TIMING DELAY FOR NON-CHECKING SORTING CHECKER AND RANGE CHECKER

Checker circuit	Number of sliced LUTs	Number of LUT flip flop pairs used	Worst case combinational delay (ns)
Non-increasing sorting checker	70	70	8.636
Range checker	34	34	6.958

The following table shows the hardware utilization and combinational delay incurred in implementing a multi-bit TSC with reconfiguration logic, which in turn consists of single bit TSC, 16 bit reconfiguration logic and a multiplexer as sub-modules. We notice that the overhead in adding the reconfiguration logic is less but it does lead to an increase in combinational delay.

TABLE III

HARDWARE UTILIZATION AND TIMING DELAY FOR TSC WITH RECONFIGURATION LOGIC

Checker circuit	Number of Sliced LUTs	Number of LUT flip flop pairs used	Worst case combinational delay (ns)
TSC (1 bit)	2	2	4.494
TSC (16 bit)	49	49	12.555
Reconfiguration logic (16 bit)	32	32	4.678
TSC with reconfiguration logic (16 bit)	64	64	18.713

VII. Conclusion

Reconfiguration logic prolongs system halt by detecting the faulty block, isolating it and reconfiguring the functional units to produce disrupted fault free output. This reconfiguration unit can be implemented with very little overhead in terms of hardware utilization.

Other circuits such as less than or equal to checker, range checker and residue checker can also be utilized as viable circuits for determining the validity of the obtained output.

For computing the residue for modulo of 2^n, the residue checker built using the simpler 2^n modulo checker unit can be preferred over the modulo checker which is capable of calculating the residue for any value of modulo, as it is more conservative in terms of hardware utilization. The generalized modulo checker could be utilized for computation of residue for any modulo other than modulo of 2^n.

This paper only talks about self-checking circuits but not all of them are fault secure. Future scope of this project could include conceptualizing checker circuits which are fault secure and have the ability to reconfigure the given hardware with minimal overhead and latency.

VIII. Acknowledgement

The authors would like to thank Prof. Kewal Saluja for his valuable comments and suggestions that helped in improving the quality of this research.

IX. REFERENCES

[1] D.L. Tao, "A Self-Testing Non-increasing Order Checker", IEEE Transactions on Computers, Vol. 46, No. 7, July 1997

[2] Shugang Wei, Kensuke Shimizu, "Error Detection of Arithmetic Circuits Using a Residue Checker with Signed-Digit Number System", IEEE International Symposium on Defect and Fault Tolerance in CLSI Systems (DFT'01)

[3] Chung-Lung Hsu and Ting-Hsuan Chen, "Built-in Self-Test Design for Fault Detection and Fault Diagnosis in SRAM-based FPGA", IEEE Transactions on Instrumentation and Measurement, Vol. 58, No. 7, July 2009

[4] Stanislaw J. Piestrak, Abbas Dandache and Fabrice Monteiro, "Designing Fault-Secure Parallel Encoders for Systematic Linear Error Correcting Codes", IEEE Transactions on Reliability, Vol. 52, No.4, December 2003

[5] Mou Hu, H.T. Mouftah, "Fault-Tolerant System Using 3-Value Logic Circuits", IEEE Transactions on Reliability, Vol. R-36, No.2, June 1987

[6] Wonhak Hong, Rajashekhar Modugu and Minsu Choi, "Efficient Online Self-Checking Modulo $2^n + 1$ Multiplier Design", IEEE Transactions on Computers, Vol. 60, No. 9, September 2011

[7] Joseph L.A. Hughes, Edward J. McCluskey and David J. Lu, "Design of Totally Self-Checking Comparators with an Arbitrary Number of Inputs", IEEE Transactions on Computers, Vol. c-33, No. 6, June 1984

[8] Vitaly Ocheretny, "Self-Checking Arithmetic Logic Unit with Duplicated Outputs", IEEE 16th International On-Line Testing Symposium, 2010

[9] T.-P. Chuang, C.W. Chiou and S.-S. Lin. "Self-checking alternating logic bit-parallel Gaussian normal basis multiplier with type-t", IET Information Security

[10] Mohsin Amin, Abbas Ramazani, Fabrice Monteiro, Camille Diou and Abbas Dandache, "A Self-Checking Hardware Journal for a Fault-Tolerant Processor Architecture", International Journal of Reconfigurable Computing, Vol. 2011, Article ID 962062, 15 pages

[11] P.K. Lala, "Self-Checking and Fault-Tolerant Digital Design", USA: Morgan Kaufmann, Jun. 2000.

[12] D.A. Anderson and G.Metze, "Design of Totally Self-Checking Checker for m-out-of-n Codes," IEEE Trans. Computers, vol. 22, pp. 263-269, Mar. 1973.

[13] F.J.Taylor, "A VLSI residue arithmetic multiplier," IEEE Trans. Computers, vol. C-31, pp. 540-546, June 1982

[14] C.D. Thompson, "The VLSI Complexity of Sorting," IEEE Trans. Computers, vol. 32, no. 12, pp. 1,171-1,184, Dec. 1983.