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.**

# 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. --------------------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, i.e., if an input lies in a particular range.

The rest of this 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 Residue Checker. Finally, conclusions are given in Section V.

# 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 figure-----------------. Two out the four out 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 the TSC needs 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.

In our implementation, the TSC is a two-rail checker circuit-----figure no--------. The two-rail checker two groups of inputs (x0, y0) and (x1, y1) where y0 is the complementary of x0 and y1 is the complementary of x1, and two outputs f and g. In a non-error situation when x0x1 = 11 and y0y1 = 00, the output of the circuit is f = 0 and g = 1. Now due to a fault suppose y0y1 = 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 ------figure-----. 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.

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 x0x1 = 11 and y0y1 = 00, which produces an output f = 0 and g = 1. Now due to a fault (in y0), y0y1 = 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., y0 in this case. Then, the new y0 value is computed from the non-faulty input, i.e., y0 = 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., x0y0 or x1y1) in the inputs and correct them.

# LTOETC (Less than or equal to Checker):

Sorting plays an important role in data processing and 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.

Consider a sequence N1, N2, N3, N4, N5 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. When an error occurs, all the numbers remain the same and it’s the sorting function that has not been implemented correctly which is in fact a functional error.

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 (x2,...,xk) >= (y2,...,yk) then output of the comparator co1 = 1 & co2 = 1 else co1 = 0 and co2 = 0.

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 (x1, x2...xk) >= (y1, y2... yk) is always true during normal operation. Let X1 = (x2... xk) and Y1 = (y2... yk). The input code space includes:

1. x1 y1 = 00 and X1 ≥ Y1
2. x1 y1 = 11 and X1 ≥ Y1
3. x1 y1 = 10 and X1 ≥ Y1
4. x1 y1 = 10 and X1 < Y1

From figures ------, 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.

1. When a fault in each functional block is excited, a non-code word will be generated at the outputs o1, o2.

* If a fault occurs in generator, translator, trc1, trc2, one of XOR gates and one of the inverters shown in Figure blah, then a test vector will generate 00 or 11 either at a31b31 or a32b32 (not both). Hence o1o2 become either 00 or 11.
* If a fault occurs in trc3, then a test vector will generate either 00 or 11.
* If a fault occurs in (k-1)-bit comparator, comp1 or comp2, then a test vector will generate 00 or 11 only at a31b31 or a32b32. Hence, o1o2 become either 00 or 11.

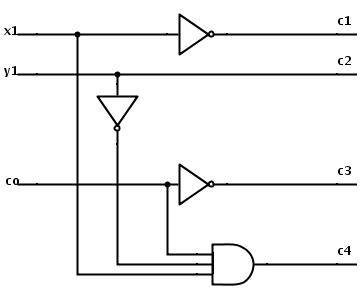


Fig. 1 Generator

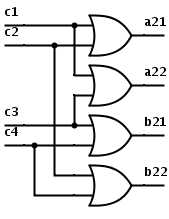


Fig. 2 Translator

LTOETC compares two numbers. We can detect if more than two numbers are sorted correctly using multiple such LTOETC blocks. Figure blah shows one example where 5 numbers, i.e., N1, N2, N3, N4 and N5 are checked if they are arranged in a sorted sequence. If N1 < N2 or N2 < N3 or N3 < N4 or N4 < N5, output o1o2 of this circuit should be 11 otherwise the circuit will generate valid code words.

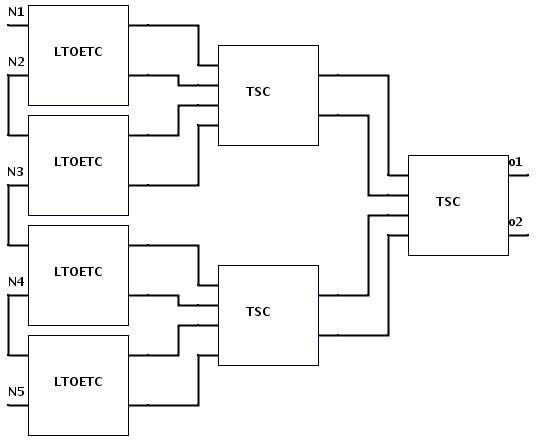


Fig. 3 Non-increasing sorting checker with 5 inputs

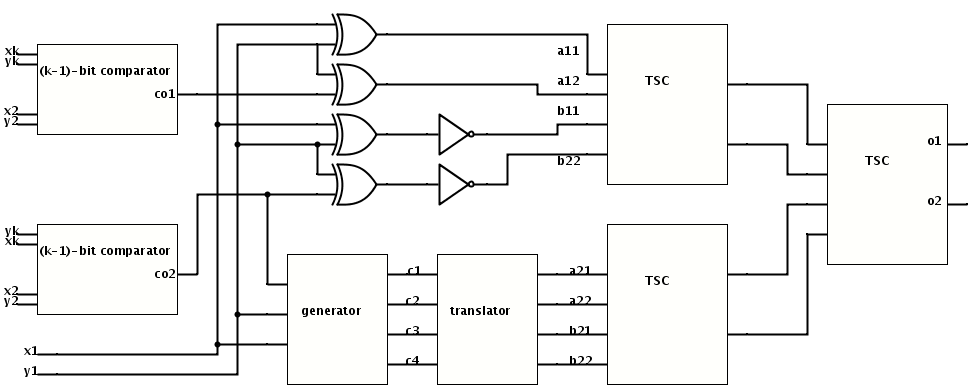
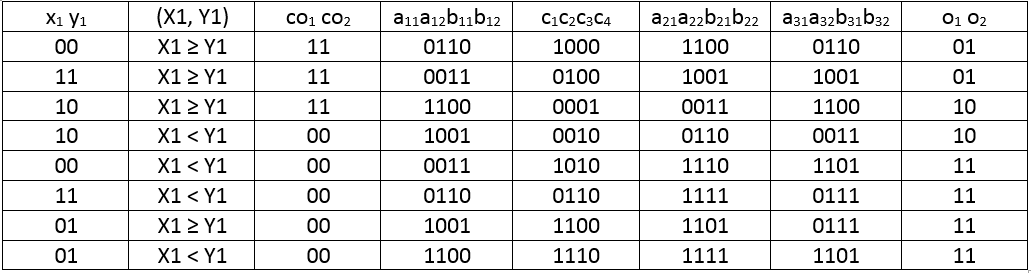


TABLE I

INPUT/OUTPUT RELATIONSHIP OF FUNCTIONAL BLOCKS

Fig. 4 Less than or equal to checker

# RANGE CHECKER

LTOETC block can be used to check if a number lies within a range. If the number doesn’t lie within the range, then the outputs {o1, o2} will generate 11 otherwise it will generate valid codewords. Figure blah shows that the range checker requires two LTOETC blocks where the first LTOETC has inputs as the input to be checked and the highest limit of the range and the second LTOETC will have the same input and the lowest limit of the range.

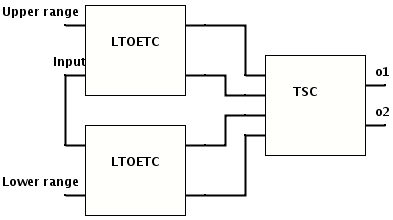


Fig. 5 Non-increasing sorting checker

# 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. ­­­­­­

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

Z = A\*B+C

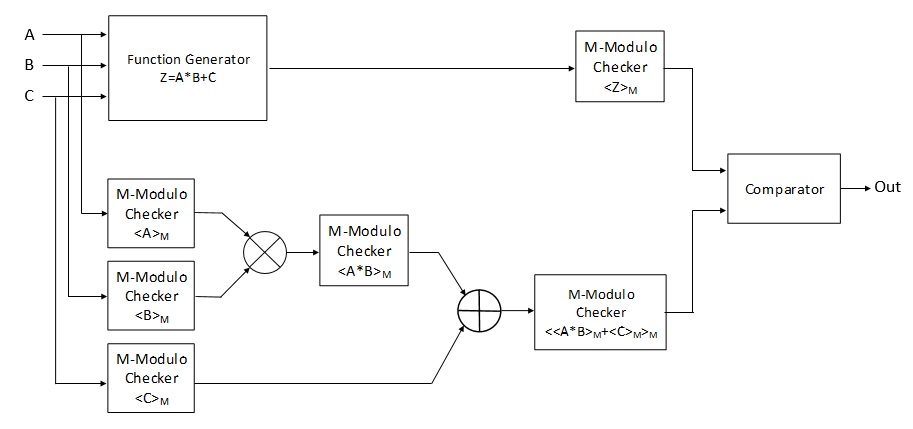


Fig. 6 Residue checker

The first modulo checker circuit computes the residue for this entire function as <Z>m. Now, the second modulo checker computes the residue for this function based on the properties of modulo arithmetic.

**Property 1:** <X+Y>m = < <X>m + <Y>m >m

**Property 2:** <X.Y>m = < <X>m \* <Y>m >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:

<Z’>m = <<<A>m\*<B>m>m+<C>m>m

Once the residue <Z>m and <Z’>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 Figure ------x------.

Modulo checkers can be implemented in several ways in hardware. The most common method that is adopted for computing modulo of 2­n 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 2n, n . 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 Figure -----. 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) = 4.2814ln(m) + 5.275

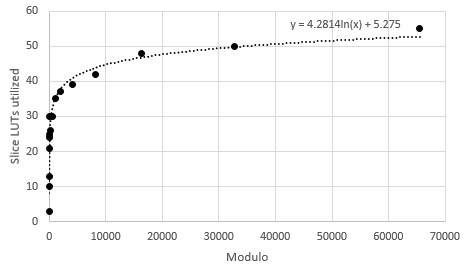


Fig. 7 Plot illustrating hardware utilization vs Modulo for 2n modulo checker.

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 . 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 Figure---. Hardware complexity for this design is computed by curve fitting this plot which yields the following equation:

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

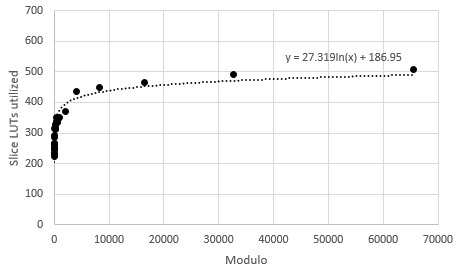


Fig. 8 Plot illustrating hardware utilization vs Modulo for any value modulo checker.

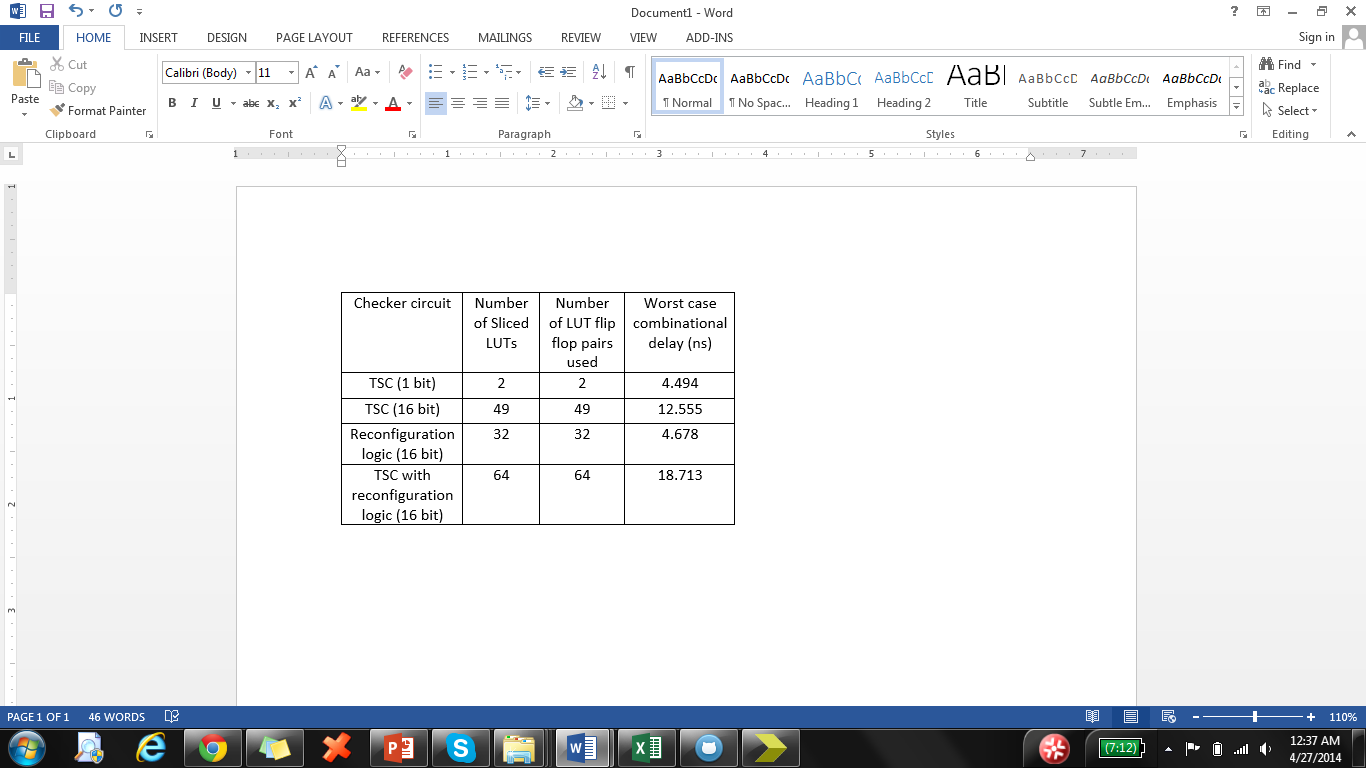
# Results

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 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 II

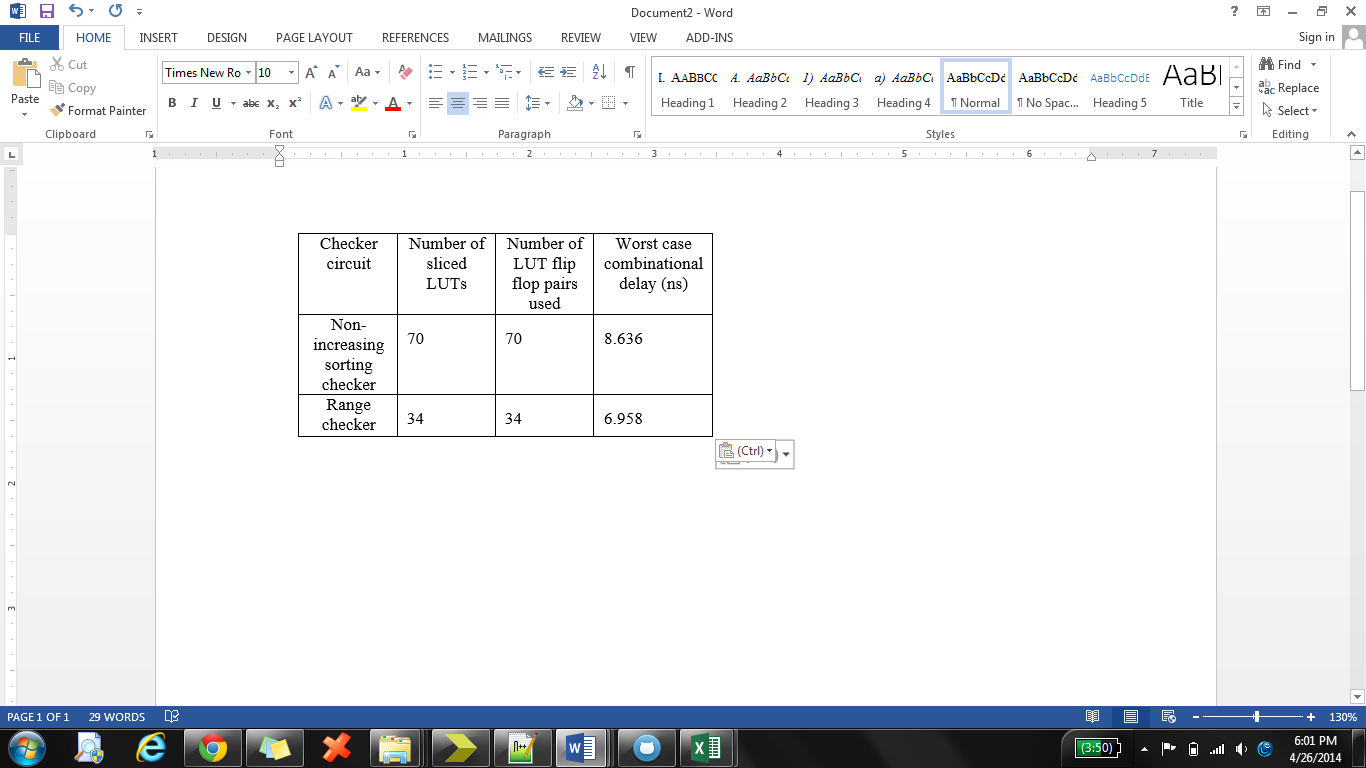
HARDWARE UTILIZATION AND TIMING DELAY FOR TSC WITH RECONFIGURATION LOGIC



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 III

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



# 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.

Residue checker built using the simpler 2n modulo checker unit can be preferred over modulo checker which is capable of calculating the residue for any value of modulo, as it’s more conservative in terms of hardware utilization.

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.

# REFERENCES

### D.L. Tao, “A Self-Testing Nonincreasing Order Checker”, IEEE Transactions on Computers, Vol. 46, No. 7, July 1997

### Shugang Wei, Kensuke Shimizu, “Error Detection of Arithmetic Circuits Using a ResidueChecker with Signed-Digit Number System”, IEEE International Symposium on Defect and Fault Tolerance in CLSI Systems (DFT’01)

### Chung-Lung Hsu and Ting-Hsuan Chen, “Buit-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

### 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

### Mou Hu, H.T. Mouftah, “Fault-Tolerant System Using 3-Value Logic Circuits”, IEEE Transactions on Reliability, Vol. R-36, No.2, June 1987

### Wonhak Hong, Rajashekhar Modugu and Minsu Choi, “Efficient Online Self-Checking Modulo 2n + 1 Multiplier Design”, IEEE Transactions on Computers, Vol. 60, No. 9, September 2011

### 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

### Vitaly Ocheretny, “Self-Checking Arithmetic Logic Unit with Duplicated Outputs”, IEEE 16th International On-Line Testing Symposium, 2010

### 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

### 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