Design of Fault Tolerant State Machine for a Configurable RISC Processor

Anupama N G[1]
VLSI Design & Embedded Systems,
Dept. of E&CE
Vemana Institute of Technology
Bangalore Karnataka India

***Abstract*—The efficient approach of exhibiting a system's ability to tolerate failure is incorporating a fault tolerant application consisting of limited resources in it. In this paper, three new fault tolerant methodologies are designed for a configurable RISC Processor on its control unit. The control unit of the configurable RISC processor is a finite state machine consisting of states fetch, decode, execute, write back and memory access. The fault tolerant techniques are designed such that even in the presence of faults in the finite state machine, the faults are corrected and there will be no disturbance in the execution of the state cycle in the RISC processor. A detailed result analysis is performed among the three fault tolerant methodologies to get the lowest area overhead and maximum frequency of operation. The designed fault tolerant methodologies uses standard libraries and the state errors are corrected within N clock cycles where N is the number of the state elements of the state machine.**

Keywords—Decode; Execute; Fetch; Memory Access; State Machine; Write Back

I. Introduction

The fault tolerant and reliable computing deals with many issues in different stages of system design. Computing systems regularly encounter many failures such as software, disk, bus faults etc. The parameters to measure capability of a system concerning fault tolerance are its availability as well as reliability. Reliability is the measure of number of chances that a system will remain in operation despite failures for a specific duration. Availability is the slice of time a system is operational. It is significant to note that a system with high availability would be vulnerable to failures. The ideal system is said to be completely reliable and never fail which is not possible to achieve in reality.

In applications where computers are needed, functional outages and its repair are very costly. To get the desired reliability and availability, fault tolerant systems are required. They have the capacity to tolerate faults by spotting failures, and separating the defective modules so that the rest of the system can operate properly. Systems fail for many reasons such as the system can be specified in such a way that can lead to an incorrect design or the system might contain a fault that occurs only under certain conditions that weren't tested.

The background environment may also cause a system to work in an undesired manner. For example the computers in industries and factories operate in environment where there are temperature variations, dust, moisture etc. Finally, aging old components can also work incorrectly. Hence the Fault Tolerant applications are very necessary in today's world. To explore the problem of tolerating faults in a given set of Finite State Machines, the classic approach involves replication and using the reserve FSMs. For example, given two state machines, say A and B, to tolerate faults among them, the approach maintains two copies of each FSM's, thus resulting in a total of six FSMs in the system. The first drawback to this method is replication of the state machines is a major factor which subjects to more cost and area overhead. The next and a distinct downside of this approach is that an error caused by a transient fault in the data flow path might get overlooked resulting in an erroneous state of triplicated systems. Hence an approach at the gate level is necessary rather than directly triplicating the state machines.

In this paper, single bit transient errors in a state machine of the configurable processor is corrected with three different fault tolerant methodologies where the state machine is not triplicated. Instead fault tolerant strategy is included at the gate level by making little modifications to the state machine of the configurable processor. Hence the fault tolerance is achieved with the advantage of lowest area overhead in the regular state machine of the configurable processor.

II. Design Methodology

A. *Fault Tolerant Finite State Machine*

Fault tolerance systems describes a logic designed in such a way that in an incident that a component stalls, a strategy that can instantly take its place for computation with no loss of service. Fault tolerant finite state machines are same as finite state machines but can work in the faulty conditions.

Fundamentally failures begin from physical failure which can be a result of radiation induced faults or any change in its physical configuration from which logical faults arise and then the system errors are the end results. Two general approaches in increasing system reliability are fault avoidance and fault tolerance. The goal of fault avoidance is to avoid the faults occurring in the system but even after most cautious implementations of fault avoidance methodologies, failures will occur after a long period of time.

In comparison, fault tolerance deals with the system design with the assumption that defects might very much likely occur in any way during system operational stage. Hence the design is intended towards making the system keep operating correctly even with the presence of defects and errors. Hence fault tolerance strategies are much in demand.

B. The 8 bit Configurable RISC Processor

When designing a new processor module, there are steps that needs to be pursued to ensure that process flow is logical and easy. The steps in involved are: i. Finalizing the features the new processor must poses. ii. Ensuring that datapath are inline with capabilities so desired.iii. Specifying the code format. This has to be as Machine Instructions. iv. Building the desired logic to control the flow of data.

Before designing a new processor element, it is essential to first find what is the purpose of the new design is. What new thing should the new processor handle. Here the configurable processor designed has the following capabilities. It is a custom made processor using the Gumnut core for fault tolerance strategy. It is self contained and can be interfaced with external peripherals. It is designed using limited set of instructions.

The datapath is the route that the data follows in a processor. During the journey to different segments of the processor, the signals from the control unit cause the data to be manipulated in specific desired aspects, as per to the instruction. The datapath is made up of the logic for manipulation of data to obtain the required result and for holding interim data. It contains Arithmetic and Logic Units and all the registers capable of handling data. In the processor designed there are eight general purpose registers which are 8 bit and a 12 bit program counter. Once the processor reset happens, the program counter is cleared and then the instructions are executed accordingly. Once a basic datapath is ready Instruction Set Architecture can be designed.

The configurable RISC Processor designed uses a Gumnut Instruction Set which is one of the most abundantly used RISC architecture. It has an instruction memory of 4096 bytes and data memory of 256 bytes. It starts executing the instructions when the processor is reset low i.e to zero. It has eight general purpose registers i.e r0 to r7. The registers hold data operated by user instructions. Each instruction in the RISC processor designed is coded in binary form. Therefore add is coded as 000, addc which is add with carry is coded as 001, sub as 010, subc as 011. The logical instructions such as and is coded as 100, or as 101, xor as 110, mask as 111. The shift instructions i.e shift left and shift right are coded as 00 and 01 respectively. The rotate instructions such as rotate left and rotate right are coded as 10 and 11 respectively. In this way every instruction in the configurable RISC processor designed is coded according to the Gumnut Instruction Set Architecture. These instructions include memory, I/O instructions, jump instructions etc. Determining the length of the instruction word in a RISC is a very important matter, and one that is worth a considerable amount of thought. Here the word size of 8 bit is used for this configurable processor. On an embedded system however, with limited program ROM, the length of the instruction word will have a direct effect on the size of potential programs, and the usefulness of the chips. The length of the opcode field will directly impact the number of distinct instructions implemented. If the opcode field is very small, there is no space to all the instructions. If its large, valuable bits in the instructions are wasted. Some instructions are larger than other instructions. An example to prove this statement is instructions with memory locations being operated, or jump instructions used are larger than instructions which operates on registers only. Hence an optimal desired extra space is left in the opcode field.

The control logic units are implemented as state machine having the states as fetch, decode, execute , memory and write back states. The state Fetch is binary coded as 000, Decode as 001, Execute as 010, Memory as 011 and Write Back as 100.Depending on the value of the transition function, the states change accordingly. When the transition function is 0, it remains in the same state. When the transition function is 1, it goes to the next state. The state transition table of the finite state machine for the configurable processor is as shown in the Table 1 below .

Present State	Name	Binary Coded Form	Transition Function	Next State
s1	Fetch	000	0	s1
			1	s2
s2	Decode	001	0	s2
			1	s3
s3	Execute	010	0	s3
			1	s4
s4	Memory Access	011	0	s4
			1	s5
s5	Write Back	100	0	s5
			1	s1

Table 1: State transition table of the finite state machine

The states Fetch, Decode, Execute, Memory Access and Write Back are designed using a state machine of the control unit in the configurable processor. Each state is explained as follows.

The instruction is fetched from physical memory, i.e. RAM.(Block Memory in Xilinx). The address of the instruction must be fetched in a register called PC (program counter). The instruction is copied from memory to the instruction register, which is also another register. When the given instruction is copied to the instruction register, the operation of this state ends and the value of the transition function becomes one. The Fig 1 shows the schematic of the State Fetch for the state machine of the Configurable Processor.

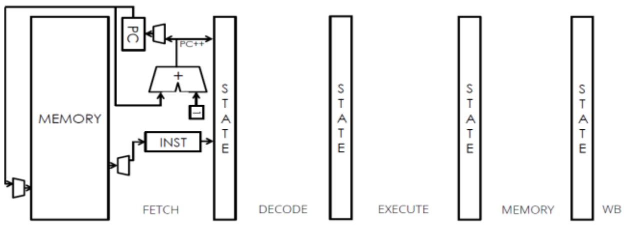


Fig 1: Schematic Representation of the State Fetch

To find out the operation which the instruction is meant to do, the instruction has to be decoded. The bits used for the opcode are used to determine how the instruction should be executed. This is meant by "decoding" the instruction. The opcode bits for the specific instructions are explained in detail in Instruction Set Architecture. When the fetched instruction is completely decoded, the state operation is over. Fig 2 shows the schematic for the State Decode of the state machine of the Configurable Processor.

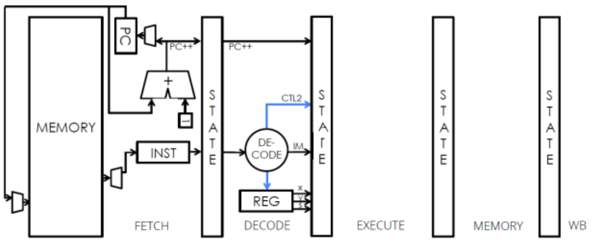


Fig 2: Schematic Representation of the State Decode

The part of the cycle where the data processing actually takes place is in the state Execute. The desired instruction is carried out upon the data. This is known as execution. This is usually done by the ALU on the processor core. Fig 3 shows the schematic for the State Execute of the state machine of the Configurable Processor.

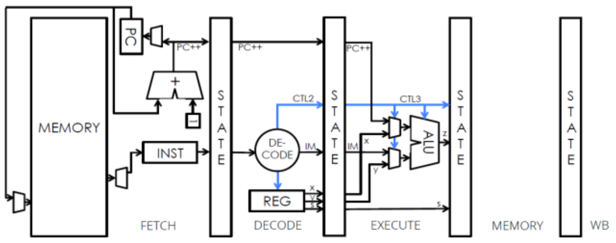


Fig 3: Schematic Representation of the State Execute

In the configurable processor, if data memory needs to be accessed, it is done so in the state Memory Access. The accessing of the memory happens only while load and store instructions are being executed. This step is simply forwarded and will not do anything when any other instructions are encountered. Fig 4 shows the schematic of the State Memory Access for the state machine of the Configurable Processor.

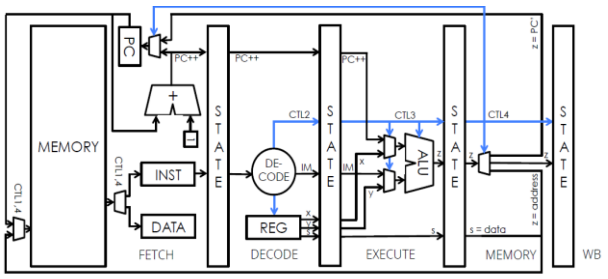


Fig 4: Schematic Representation of the State Memory Access

Once the desired operation is performed, the result is stored in the register in the register file. This is the operation of the state Write Back. It writes back the result to the appropriate file as instructed. Fig 5 shows the schematic for the State Write Back for the state machine of the Configurable Processor.

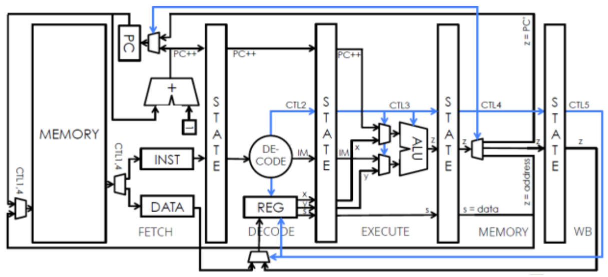


Fig 5: Schematic Representation of the State Write Back

III. FAULT TOLERANT METHODOLOGIES

The three methods of fault tolerance discussed in this section are triple modular redundancy, LUT based approach and fault tolerance using error correcting codes

A. Triple Modular Redundancy

Triple Modular Redundancy (TMR) is one of the frequently used fault masking technique. TMR technique is based on implementation of the redundancy of function. The outputs of these modules are given to a voter. In the standard approach of Triple Modular Redundancy a system is triplicated into three functionally identical systems and the outputs of these three triplicated systems are connected to a three input voter. The voter determines the output of the fault-tolerant systems which is the majority of the outputs of the triplicated systems. The TMR flip flop concept is applied to simplify the implementation in the control unit of the configurable processor. Data paths in digital systems are treated as redundant modules. Each TMR flip flop has three redundant data paths. These are based on majority voting algorithm. Fig 6 demonstrates the application of Triple Modular Redundancy applied to digital systems.

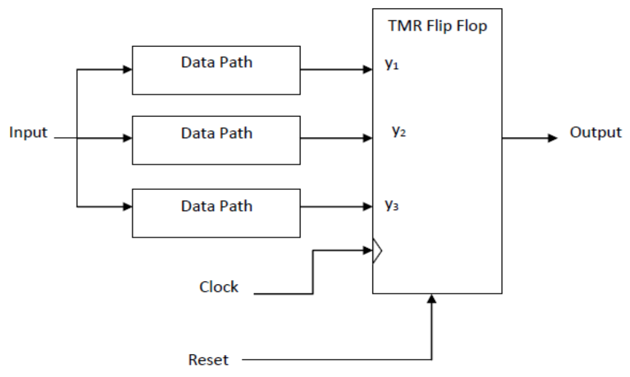


Fig 6: TMR Concept applied to data paths of digital system

The one or more TMR flip flops when jointly used comprises a network on digital logic structure. A TMR hierarchy can be formed. This is possible when multiple modules of TMR flip flops consists of other TMR flip flops. The TMR network is created when one TMR flip flop supports the other TMR flip flops. The TMR network when implemented in a state machine of the configurable processor becomes fault tolerant.

B. LUT Based Approach of Fault Tolerance

A lookup table is an array structure that can enhance the effectiveness of execution. This is achieved by using a an array and using an index for reading the array. There is significant speed up of processing time. This is because fetching a value from memory is always faster compared to time consuming I/O operations. The values are calculated in prior and loaded in static program storage. Lookup tables can be also used for accepting input values by comparing with a list of valid/invalid data in an array. It can include pointer functions, offsets ,labels or can define macros to process the matching input.

However, the LUT is not regular logic but a different method of implementation of combinational logic. SRAM based LUT and MUX along with Flip Flops forms the basic building blocks of fault tolerance technique used here. The states of the state machine of the processor are conFigd as combinational logic in a LUT and stored. Any access of the control unit of the processor is through this conFigd LUT. Therefore, it’s a wise choice to implement the state machines combinational logic part using memories which are LUTs. The combinational logic part is stored in a SRAM based LUT. Hence state lines and the state variables are accessed from the memory.

The contents of the memory is tabulated in the Table 2 and the address lines of the memory are treated as inputs of the combinational logic and the look-up values stored in the memory are corresponding output values. The Current State and IR_decode_mem(Instruction Register for decode) are inputs and depending upon the values the next state is decided. The Table 2 is the state table that defines the state machine conFigd in a LUT.

Current State	IR_decode_mem	Next State
Fetch	0	Fetch
	1	Decode
Decode	0	Decode
	1	Execute
Execute	0	Write Back
	1	Memory
Memory	0	Fetch
	1	Fetch
Write back	0	Fetch
	1	Fetch

Table 2: State table conFigd in the LUT

C. Fault Tolerance Using Error Correcting Codes

This method is an experimental method of error correction with in a chip. In this method, the input state values are encoded with values greater than the number of bits. These bits can be ordinary values or error correction values. The encoded bits after comparison with the stored actual values are decoded with an appropriate decoder. The state value is encoded using a ECC check bits. The check bit is calculated as follows and shown in the Fig7. Numbering of the bits in 8 bit is done. This is from d1 till d8. The error correcting bits are from e1 to e12. The error correcting bits are the ones which have the indices as powers of 2. The rest are data bits.

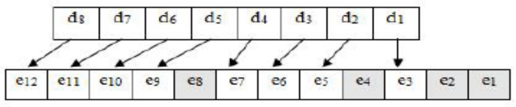


Fig 7: Formation of error correction bit from a data bit

If the bits are written in binary, then the index where 1 occurs is noted. All the other bits in the array which have one in the same index as the check bit are considered and an XOR is performed on them. The value which is an output of this operation is the Error Correction bit. Example: check bit e2(index=0010) is the XOR of e3, e6,e7,e10 and e11.

Upon data in an address being read, the whole ECC is considered. The 12 bit ecc word is computed from the 8 bit data word. The transmitted ecc and received ecc are computed by applying XOR operation. If the output is 0000, then there is no error. Else, the position of the bit flipped is known. This can be used for correction The encoded code words for the each state is stored in a Block memory of Xilinx. Every time the encoded state appears it is compared with the stored value.

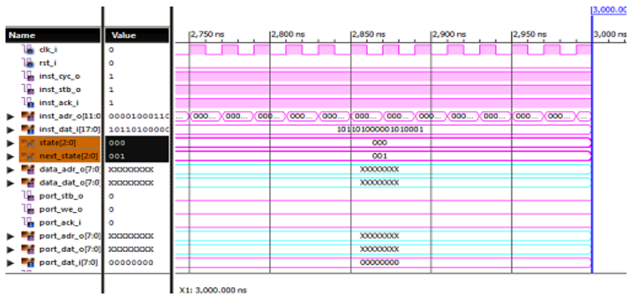
If there is no error the encoded value matches with the stored value and the bits are decoded back to regain the state value. Else the error is corrected and the state is regained. For example: (001)2 encoded as (000000111)2 Let’s insert an error (001000111)2. (001000111)2 can still be decoded as (001)2.

IV. SIMULATION RESULTS

The simulations are done using the Xilinx ISE Design simulation tool. The states of the configurable processor are being coded in binary as 000 for Fetch, 001 for Decode, 010 for Execute, 011 for Memory Access and 0100 for Write Back. When there is no fault with the processor's state machine the control unit goes with the regular cycles of fetch decode execute etc according to the instruction. When the present state is Fetch the next state Decode will be ready at the clock edge thus proving the correctness in the working of the finite state machine by showing the updated value of the led status register of the led GPIO peripheral.

A. *Without fault tolerance*

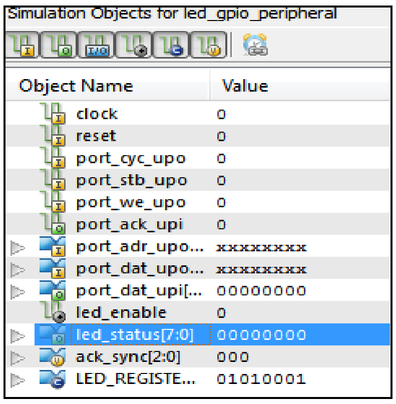
The Fig 8 shows the simulation of the processor core with errors introduced. It is clearly seen that that the state machines state is stuck at 0. From the design we know that the state machine should go through all the states for every instruction. But due to the error in the state machine logic, state variable is not changing. This will lead to not executing the appropriate logic.



Name	Value
clk_i	0
rst_i	0
inst_cyc_o	1
inst_stb_o	1
inst_ack_i	1
inst_adr_o[11:0]	00001000110
inst_dat_i[17:0]	10110100000
state[2:0]	000
next_state[2:0]	001
data_adr_o[7:0]	XXXXXXXX
data_dat_o[7:0]	XXXXXXXX
port_stb_o	0
port_we_o	0
port_ack_i	0
port_adr_o[7:0]	XXXXXXXX
port_dat_o[7:0]	XXXXXXXX
port_dat_i[7:0]	00000000

Fig 8: Simulation Result of Processor with faults

In the design of the processor, the processor wants to print value $(00001111)_2$ on the LED GPIO peripheral. Though the instruction is fetched for printing a value on LED, it's not updated. Hence the led status register is not updated with the desired value as shown in the Fig 9 because of the errors encountered during its operation.



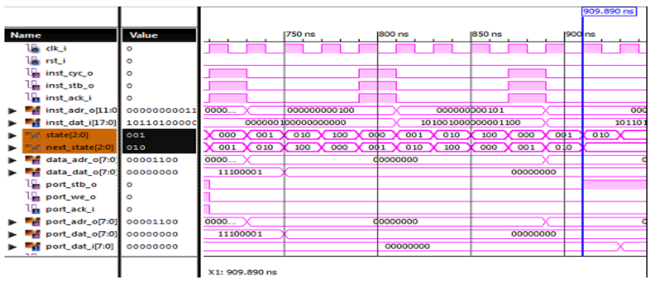
Simulation Objects for led_gpio_peripheral

Object Name	Value
clock	0
reset	0
port_cyc_upo	0
port_stb_upo	0
port_we_upo	0
port_ack_upi	0
port_adr_upo...	xxxxxxxx
port_dat_upo...	xxxxxxxx
port_dat_upi[...	00000000
led_enable	0
led_status[7:0]	00000000
ack_sync[2:0]	000
LED_REGISTE...	01010001

Fig 9: The LED status Register not updated due to incorrect operation

B. *With Fault Tolerance*

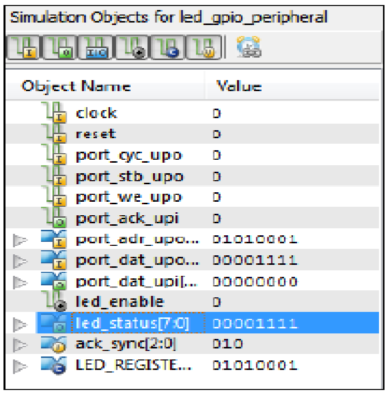
After applying each three different fault tolerant methodologies, the processor works fine even in the presence of faults. The processor's state machine which is the control unit goes with the regular cycles of fetch decode execute etc according to the instruction written for it. When the present state is Fetch the next state Decode will be ready at the clock edge thus proving the working of the finite state machine. Fig 10 shows the correct working of the configurable processor even in the presence of faults due to fault tolerant methodologies.



Name	Value
clk_i	0
rst_i	0
inst_cyc_o	0
inst_stb_o	0
inst_ack_i	0
inst_adr_o[11:0]	00000000011
inst_dat_i[17:0]	10110100000
state[2:0]	001
next_state[2:0]	010
data_adr_o[7:0]	00001100
data_dat_o[7:0]	00000000
port_stb_o	0
port_we_o	0
port_ack_i	0
port_adr_o[7:0]	00001100
port_dat_o[7:0]	00000000
port_dat_i[7:0]	00000000

Fig 10: Simulation result of Processor operating correctly in the presence of faults

The errors are corrected by the fault tolerant methodologies and the next state is rectified in every clock cycle and the led status register is updated with the desired value of (00001111)2 which is as shown in the Fig 11.



Simulation Objects for led_gpio_peripheral

Object Name	Value
clock	0
reset	0
port_cyc_upo	0
port_stb_upo	0
port_we_upo	0
port_ack_upi	0
port_adr_upo...	01010001
port_dat_upo...	00001111
port_dat_upi[...	00000000
led_enable	0
led_status[7:0]	00001111
ack_sync[2:0]	010
LED_REGISTE...	01010001

Fig 11: The LED status Register updated after subjected to fault tolerant methodologies

V RESULT SUMMARY

The simulation results of various parameters of a configurable processor without state error correction logic and with different state error correction methodologies are tabulated in the Table 3.

Parameters/ Design	No of LUT's	No of FlipFlops	No of Logic Gates	Max Frequency (Mhz)	Min Time Period (ns)	Propagation Delay (ns)
Processor with no State error Correction Logic	1835	1578	1140	73.452	13.614	5.493
TMR	1845	1598	1210	73.452	13.614	10.596
LUT	1959	1518	1141	57.197	17.483	6.981
ECC	1838	1609	1238	57.197	17.483	6.981

Table 3: Result summary of configurable processor without state error correction logic and with different state error correction methodologies

From the Table 3 it can be seen that there is definitely an area overhead in processor with state error correction methodologies than compared to regular configurable processor with no state error correction strategies. But TMR methodology gives the lowest area overhead and highest maximum frequency of operation. So the TMR is correcting the state errors at a higher rate at a given clock cycle compared to the other two methodologies where the maximum frequency of operation is reduced and also there is a larger area overhead compared to the TMR technique. Hence because of higher maximum frequency of operation in TMR the speed is increased though it is having a little higher propagation delay which can be overlooked when compared to the other two methods of fault tolerance in state machine of configurable processor. Thus TMR technique can be used readily for fault tolerance in state machines of processor.

VI CONCLUSION

In this paper, three methodologies of fault tolerance are discussed for a state machine of configurable RISC processor. A detailed result analysis is performed to show which of the three methods best suites the given system in terms of lowest area overhead and maximum frequency of operation. Simulations are done using Xilinx ISE design tool and verified by implementing on Spartan 3A FPGA. It is proved that TMR technique gives the lowest area overhead and maximum frequency of operation. It suites the configurable processor by only including fault tolerance strategy and not altering its other properties like clock cycle etc. The fault tolerant methodologies use no specific error models and transient errors in the state elements are easily corrected within N clock cycles by use of the corrected output states where N is the no of the state elements. The simple design libraries are used and thus fault tolerant methodologies discussed in this thesis proves to be efficient one by increasing the reliability of the system.

ACKNOWLEDGMENT

I would like to thank Mr.Purushotham Sannakariyappa, Chief Consultant at Mindful Learning Ltd. for his helpful suggestions for the synthesis and implementation of the design.

REFERENCES

[1] Stefan Weidling and Michael Goessel, "Fault Tolerant Linear State Machines", in Proceedings of the IEEE Int. Conf. on Electronics, Circuits and Systems,2014, pp.4799 - 4711

[2] Jiang,G,"Reconfiguring Three-dimensional Processor Arrays for Fault tolerance Hardness and Heuristic Algorithms"in Proceedings of IEEE Conf. on Computer Systems,2015, pp. 345-349

[3] Psarakis, M., Vavousis, A, Bolchini, C " Design and implementation of a self healing processor onSRAM-based FPGAs"in Proceedings of the IEEE Conf on Defect and Fault Tolerance in VLSI and Nanotechnology Systems, 2014, pp. 165 - 170

[4] Safarulla, I.M and Manilal, K, " Design of Soft error tolerance technique for FPGA based soft core processors" in Proceedings of the IEEE Conf. on Advanced Communication Control and Computing Technologies, 2014,pp. 1036 - 1040

[5] Peng Wang, Kaiyuan Zhang and Rong Chen,"Replication-Based Fault-Tolerance for Large-Scale Graph Processing" in Proceedings of the IEEE Conf. on Dependable Systems and Networks, 2014, pp. 562 - 573

[6] N.Ramkumar, and Boopathy.S, "Analysis of berger code based Fault tolerant techniques for embedded system" in Proceedings of IEEE Conf. on Advanced Communication Control and Computing Technologies, 2013,pp 145- 149

[7] Gowtham Raj, G. and Kannan, B. and Aravind, T, " Fault scanning and repairingin processor based systemusing dynamic reconfiguration" in Proceedings of the IEEE Conf. on Smart Structures and Systems, 2013, pp.120 - 124

[8] Ferlini, Frederico , " Non- intrusive fault tolerance in soft processors through circuit duplication" in Proceedings of IEEE Test Workshop (LATW), 2012 , pp.1-6

[9] Augustin, M., Gossel, M. and Kraemer, R , " Selective fault tolerance for finite state machines"in Proceedings of the IEEE Conf. on On-Line Testing and applications, 2011, pp. 43 - 48

[10] Ju-Yueh Lee, Yu Hu, Rupak Majumdar and Minming Li "Fault Tolerant Resynthesis with dual output LUT's"in Proceedings of IEEE Int. Conf on Design Automation ,2010 ,pp. 325 - 330

[11] Hadjicostis, C.N. and Verghese, George C,"Coding Approaches to Fault Tolerance in Linear Dynamic Systems", in Proceedings of the IEEE Int. Conf. on Information Theory,2005,pp.210-228

[12] Samudrala, P.K, Ramos, J. and Katkoori, S, " Selective triple Modular redundancy (STMR) based single event upset (SEU) tolerant synthesis for FPGAs"in Proceedings of the IEEE Conf. on Nuclear Science Systems,2004, pp. 2957- 2969