University of Texas Rio Grande Valley
ScholarWorks @ UTRGV

Theses and Dissertations - UTB/UTPA

7-2011

Design for soft error tolerance in FPGA-implemented asynchronous circuits

Yu Bai
University of Texas-Pan American

Follow this and additional works at: https://scholarworks.utrgv.edu/leg_etd

Part of the Electrical and Computer Engineering Commons

Recommended Citation

Bai, Yu, "Design for soft error tolerance in FPGA-implemented asynchronous circuits" (2011). *Theses and Dissertations - UTB/UTPA*. 244.
https://scholarworks.utrgv.edu/leg_etd/244

This Thesis is brought to you for free and open access by ScholarWorks @ UTRGV. It has been accepted for inclusion in Theses and Dissertations - UTB/UTPA by an authorized administrator of ScholarWorks @ UTRGV. For more information, please contact justin.white@utrgv.edu, william.flores01@utrgv.edu.

DESIGN FOR SOFT ERROR TOLERANCE IN FPGA-IMPLEMENTED ASYNCHRONOUS CIRCUITS

A Thesis

by

YU BAI

Submitted to the Graduate School of the
University of Texas Pan-American
In partial fulfillment of the requirements for the degree of

Master of Science

July 2011

Major Subject: Electrical Engineering

DESIGN FOR SOFT ERROR TOLERANCE IN FPGA-IMPLEMENTED ASYNCHRONOUS CIRCUITS

A Thesis

by

YU BAI

COMMITTEE MEMBERS

Dr. Weidong Kuang

Chair of Committee

Dr. Hasina Huq

Committee Member

Dr. Yul Chu

Committee Member

July 2011

Copyright 2011 Yu Bai
All Rights Reserved

ABSTRACT

Bai Yu, Design for Soft Error Tolerance in FPGA-Implemented Asynchronous Circuit. Master of Science (MS), July, 2011, 82 pp., 4 tables, 46 figures, references, 24 titles.

This research in its present form is the result of experimentation on effect of soft error in FPGA-implemented asynchronous circuit. The conclusion are drawn that asynchronous circuit are much easier to detect soft error than synchronous circuits. The asynchronous circuit is implemented in FPGA with software fault injection method to analyze the behavior of soft error generation in FPGA implementation asynchronous circuits. The proposed detection circuit can detect all soft errors that generated in FPGA-implemented asynchronous circuit.

The contributions include: investigation of FPGA structure, investigation of soft error model in FPGA, mechanism of FPGA implemented asynchronous circuit, behavior of soft error injection in FPGA look up table that implemented asynchronous circuit, and proposed detection scheme. The research on soft error injection in FPGA routing system and soft error rate estimation will be done in the future.

DEDICATION

I would like to give my thanks to the mighty God. He loves me and guides me in His way. Without God these all works are impossible to be done. While I was on the path through the valley of the shadow of death, He helped and guided me. Thanks God for everything You did. The completion of my MS studies would not have been possible without the support of my family. My mother, Guimin Zhang, my father Shaogang Bai, wholeheartedly inspired, motivated and supported me by all means to accomplish this degree.

ACKNOWLEDGEMENTS

I want to give thanks to my supervisor Dr. Weidong Kuang for his real love. He treats me as my father. His real abundant academic experience and excellent teaching method built up the good foundation for my future studies. His precise research attitude gives me a good model to follow. He is my lighthouse that gives me the way to go. I would like to give thanks to Dr. Yul Chu and Dr. Hasina Huq too. You are really faithful to serve department and help students. I also want to thanks to all teachers taught me in UTPA Electrical Engineering Department. Your lesson helped me a lot.

The work presented in this thesis was supported by Electrical Engineering Department. Thank Dr. Foltz the chair of Electrical Engineering Department for his faithful serving to students. Without your support I cannot finish this thesis. The teaching and researching with Electrical Engineering Department gives me good experiences for future research. I take this opportunity to thank all staffs in UPTA for always being willing to help me.

TABLE OF CONTENTS

Page

ABSTRACT ………………………………………………………………………………. iii

DEDICATION ……………………………………………………………………….. iv

ACKNOWLEDGEMENTS ………………….………………………………………… v

TABLE OF CONTENTS ………………………………………..………..……….. vi

LIST OF TABLES …………………………………………………………………. viii

LIST OF FIGURES …………………………………………………………………...ix

CHAPTER I. INTRODUCTION ……………………………………...……………… 1

1.1 overview of modern FPGA ……………………………………………………… 2

1.2 soft error background and previous work ………………………………………….. 3

1.3 brief background work on Asynchronous circuits …………………………………. 5

CHAPTER II. ASYCHRONOUS CIRCUIT ON FPGAS ……………………………………. 8

2.1 FPGA architecture ………………………………………………………………. 8

2.2 NCL circuit ………………………………………………………………….. 13

2.2.1 Completion criteria ……………………………………………………….. 13

2.2.2 Threshold gates with hysteresis ………………………………………… 15

2.2.3 NCL pipeline ……………………………………………………………… 18

2.2.4 NCL registers ……………………………………………………………… 19

2.3 FPGA implementation circuit ………………………………………………… 20

CHAPTER III. SOFT ERROR MODES IN FPGAS AND EXPERIMENTAL FAULT INJECTION …………………………………………………………………………...…….. 24
3.1 soft error models in FPGAs ……………………………………………………... 24
3.1.1 Transient error …………………………………………………………………. 25
3.1.2 Permanent error ………………………………………………………………... 27
3.1.2.1 Routing error ………………………………………………………………. 27
3.1.2.2 Bit-flip on look up table configuration bits ……………………………… 29
3.1.2.3 Bit-flip on control/clocking bits …………………………………………. 29
3.2 fault injection method …………………………………………………………….. 29
CHAPTER IV. SOFT ERROR IN FPGA-BASED ASYCHRONOUS CIRCUIT ………….. 35
4.1 soft error generation and simulation in single Threshold gate ………………………. 35
4.2 soft errors propagation and simulation in NCL pipelines …………………………… 42
CHAPTER V. PROPOSED SOFT ERROR DETECTION SCHEME AND SIMULATION ..49
5.1 soft error detection schemes ……………………………………………………… 49
5.2 soft error detection scheme simulations ……………………………………….... 51
CHAPTER VI CONCLUSION AND FUTURE WORK ……………………………………… 53
REFERENCES ………………………………………………………………….………….. 54
APPENDIX A…………………………………………………………………………….. 57
APPENDIX B…………………………………………………………………………….. 69
BIOGRAPHICAL SKETCH …………………………………………………………………82

LIST OF TABLES

Page

Table 2.2.1: Dual-rail encoding …………………………………………………………………. 14

Table 2.3: Truth table of set LUT ………………………………………………………………. 23

Table 4.1: Soft error of TH34w2 ………………………………………………………………. 41

Table 4.2: Soft errors in NCL pipeline …………………………………………………………. 47

LIST OF FIGURES

Page

Fig 1.2.1: Mechanism of soft errors in MOSFET ………………………………………………….. 4

Fig 1.2.2: Induced current ………………………………………………………………………... 4

Fig 2.1.1: Xilinx FPGA structure ………………………………………………………………… 9

Fig 2.1.2: Logic elements (LEs) ………………………………………………………………….. 9

Fig 2.1.3: The architecture of Cyclone FPGAs ……………………………………………………. 11

Fig 2.1.4: Architecture of SRAM-based 4-input look-up table (LUT) ………………………… 11

Fig 2.1.5: SRAM-FPGA architecture model ……………………………………………………… 12

Fig 2.1.6: PIPs control in switch block ……………………………………………………………. 13

Fig 2.2.1.1: Weak conditions for NCL completeness of input ………………………………… 15

Fig 2.2.2.1: Schematic and symbol of TH23 …………………………………………………… 16

Fig 2.2.2.2: Schematic and symbol of TH23W2 ………………………………………………….. 16

Fig 2.2.2.3: Symbol examples of threshold gates ……………………………………………… 18

Fig 2.2.3: Basic NCL pipeline structure …………………………………………………………... 18

Fig 2.2.4: N-bit completion component ………………………………………………………… 19

Fig 2.3.1: Threshold gate implemented on FPGAs ……………………………………………... 21

Fig 2.3.2: TH34w2 gates on Altera Cyclone II ……………………………………………………. 21

Fig 3.1.1.1: An SEU affects inverter gates and makes a bit-flip error ……………………….. 26

Fig 3.1.1.2: SEU hit on FF and Block RAM in FPGA ………………………………………. 26

Fig 3.1.2.1.1: Open errors in switch blocks ….………………………………………….. 28

Fig 3.1.2.1.2: Short errors in switch blocks ………………………………………….. 28

Fig 3.1.2.2: SEU hits on SRAM cell ………………………………………………. 29

Fig 3.2.1: Quartus II design flow ………………………………………………….. 31

Fig 3.2.2: Netlist viewer in tool button ……………………………………………. 32

Fig 3.2.3: Technology Map Viewer of TH34w2 gate ……………………………... 34

Fig 3.2.4: Resource property editor for fault injection ……………………………. 34

Fig 4.1.1: TH34w2 implemented on FPGA ………………………………………….. 36

Fig 4.1.2: No error appears at output Z (soft error setting set LUT: 0000 (0→ 1)) ………….. 36

Fig 4.1.3: Premature fire appears at output Z (soft error setting set LUT: 0011 (0→ 1)) ……. 37

Fig 4.1.4: No fire appears at output Z (soft error setting set LUT: 0111 (1→ 0)) …………… 37

Fig 4.1.5: No return to 0 appears at output Z (soft error setting Reset LUT:0000 (0→ 1)) ……. 38

Fig 4.1.6: Early return to 0 appears at output Z (soft error setting Reset LUT: 0001 (1→ 0)) 38

Fig 4.1.7: No fire appears at output Z (soft error setting Reset LUT: 0111 (1→ 0)) ………… 39

Fig 4.1.8: Oscillating appears at output Z (soft error setting hold LUT: 111 (1→ 0)) ………. 39

Fig 4.1.9: No return to 0 appears at output Z (soft error setting hold LUT 001(0 → 1)) ……. 40

Fig 4.1.10: Premature fire appears at output Z (soft error setting hold LUT 010 (0→ 1)) …... 40

Fig 4.1.11: Early return to 0 appears at output Z (soft error setting hold LUT 011 (1→ 0)) … 40

Fig 4.1.12: Summarized simulation results of TH34w2 gate with different soft errors ……….. 42

Fig 4.2.1: Normal simulated circuit pipeline ……………………………………………. 43

Fig 4.2.2: Circuit simulated ……………………………………………………………... 43

Fig 4.2.3: Invalid “11” appears at the output sum (soft error setting: Set LUT 0110 (0→1) in G3 gate in full adder) ……………………………………………………………………. 45

Fig 4.2.4: Deadlock in the pipeline when no fire at sum (soft error setting: Set LUT 1001 (1→0) in G3 gate in full adder) ………………………………………………………………….. 45

Fig 4.2.5: Deadlock in the pipeline when no return to 0 at sum (soft error setting: hold LUT 001(0->1) in G3 gate in full adder) ……………………………….……………………. 46

Fig 4.2.6: Invalid "11" appears at sum when G3 ocsillating (soft error setting: Hold LUT 000 (0→1) in G3 gate in full adder) ……………………………………………………………... 47

Fig 4.2.7: No error appears at the output when G3 ocsillating (soft error setting: Hold LUT 011(1→0)) in G3 gate in full adder) …………………………………………………………. 48

Fig 5.1.1: Scheme of soft error detection ………………………………………………….. 50

Fig 5.2.1: Simulation of proposed detected circuit ……………………………………... 51

Fig 5.2.2: The detect_no_fire output is 1 when no fire appears at sum (soft error setting: Set LUT 1001 (1->0) in G3 gate) …………………………………………………………………... 51

Fig 5.2.3: The detect_ no_return0 output is 1 when no return to 0 appears at sum (soft error setting: hold LUT 001(0->1) in G3 gate) ……………………………………………………. 52

CHAPTER I

INTRODUCTION

Soft error is called transient faults or single-event upsets (SEUs). SEUs are caused due to electrical noise or external radiation rather than design or manufacturing defects. As CMOS device sizes decrease, they are more easily affected by the low energy particles resulting from collisions between cosmic rays and particles in the atmosphere, potentially leading to a much higher rate of soft errors [3]. A Field Programmable Gate Array (FPGA) is an integrated semiconductor device designed to be programmed or configured any number of times using a schematic design or a source code in HDL (hardware description language) that describe the user's hardware design [1]. FPGAs can be configured with dense logic and have very high logic capacity. However, sensitive to Single Event Upsets (SEU) limit their widespread use in mission-critical application. The circuit implemented in FPGAs may be changed functionality by SEUs. Different from synchronous circuits, the asynchronous circuits use handshaking protocols to communicate between modules or parts of the circuits for the operations to be down in sequence, so that they are not governed by a clock circuit or global clock signal, but instead need only wait for the signals that indicate completion of instructions and operations. Since the asynchronous circuit [2] is a circuit in which the parts are largely autonomous, the effect of SEU hitting on Asynchronous circuits implemented FPGA should be explored.

1.1 Overview of modern FPGA

The FPGA industry sprouted from programmable read-only memory (PROM) and programmable logic devices (PLDs). Both PROMs and PLDs had the option of being programmed in batches in the field (field programmable), however， programmable logic was hard-wired between logic gates [1]. In 1970s, the mask programmable logic arrays (MPGA) were invented and used to implement application-specific integrated circuits (ASICs). The MPGAs make up of an array of pre-fabricated transistors that can be customized into the logic circuit. The specifying metal interconnect perform customization during chip fabrication, it means that in order to use an MPGA a large setup cost is involved and manufacturing time is long. MPGA gave motivation to the design of FPGAs. In the earlier 1980s, the monolithic memories Inc （MMI） programmable array logic (PAL) was introduced by AMD. The MMI PALs were commercially very successful. The PALs consist of an array of programmable AND gate, which link to array of programmable OR gate, thus, the output can be produced conditionally complemented. However, the PAL is only a single level that wired AND plane that feeds fixed OR-gates.

Nowadays, FPGAs are widely known in many applications such as industrial productions, spacecraft and embedded applications, according to their high performance, no non-refundable-engineering cost and fast time response. The strengths of modern FPGAs are quick prototyping and time-to-market, reprogramability, relatively easy to use. The weaknesses are, cost, density, and speed.

The FPGAs are handy thing to have on the workbench; it means they can be used for rapid prototyping. With the obvious advantages that compare with ASIC, FPGAs are used in critical applications and are replacing ASICs on a regular basis. The last decade, FPGAs are ever

increasingly collect attention by most researchers in areas of DSP applications, automotive applications, space applications, robotics, computer security, and reconfigurable computing.

1.2 Soft error background and previous work

Soft errors, also called transient errors, are intermittent malfunctions of hardware that are not reproducible [2], arise from Single Event Upset (SEU). Soft error was first discovered in memory element like Dynamic random-access memory (DRAMs) in 1970s [3]. Since then DRAMs were the focus of soft error also because it occupies most of the susceptible surface area. DRAMs of 256 KB with 1980s technology had flips of five to six bits from single alpha particle [5]. The present devices employ more flips for the same alpha particle.

These SEUs are caused by energetic neutrons originating from cosmic rays or alpha particles coming from radioactive contaminants in the package material hitting the surface of silicon devices. Device scaling significantly affects the susceptibility of integrated circuits to soft error [4]. The cause factors of intensity of these soft errors lie in energy of hitting particles, the location of the device, geometry of the impact, and design of the logic circuit. The Soft Error Rate (SER) is measured by Failures In Time (FIT): one FIT is one error per billion hours of operation. Alternative unit is mean time between failures (MTBF). EX: 1 year MTBF= $10^9/(365 \times 24) = 114{,}115$ FIT. The Fig.1.2.1 shows the mechanism of soft errors in a Metal Oxide Semiconductor Field Effect Transistor (MOSFET).

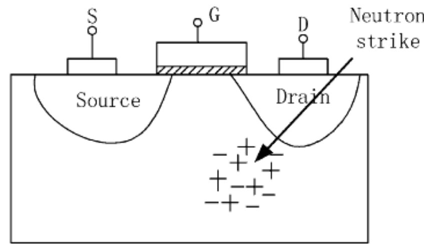

Fig.1.2.1 mechanism of soft errors in MOSFET

Electron-hole pairs with a very high carrier concentration are generated as the particle loses energy in silicon when a particle hits the drain of the MOSFET, and the resulting charges can be rapidly collected by the electric field to create a large transient current at that node [8]. The transient current can be modeled as [8]

$$I(t) = \frac{2Q}{T\sqrt{\pi}}\sqrt{\frac{t}{T}} \cdot \exp\left(-\frac{t}{T}\right)$$

where Q is the amount of collected charge, and T is a process technology-dependent time constant. The detailed discussion about this model and related parameters can be found in [9]. The Fig 1.2.2 shows the induced current caused by soft error.

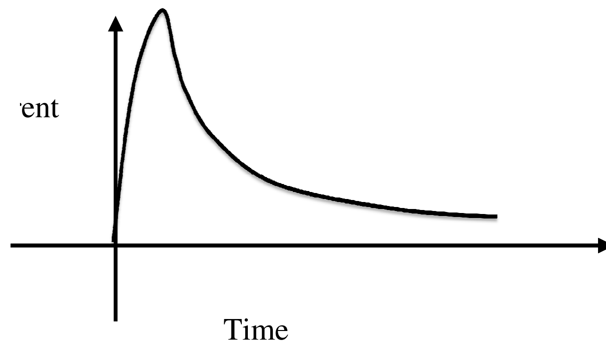

Fig.1.2.2 Induced current

The case of the current is injected into or drawn from the node depends on the type of victim drain of transistor. For example, a current is injected into the node if particle hit occurs at a p-type drain, so that it increases the node voltage. If this node of logic circuit is transmitting

logic 0 under the current injected to the node, a transient positive glitch (0-1-0) may occur at the node. Similarly, the negative transient glitch (1-0-1) may be generated if an n-type drain is hit.

Since FPGAs are vulnerable to single event upsets (SEUs) [7], an SEU with sufficient energy changes the logic state of memory element, producing a soft error [8]. For sake of SEU mitigation of FPGAs, the SEU Mitigation Techniques are therefore desirable. Many studies have focused on solution either at device level or architecture level. At device level, one solution is to use radiation-hardened FPGA devices. However, these devices are much more expensive than Commercial-Off-The-Shelf (COTS) FPGAs [7]. At architecture level, redundancy designs such as Triple Module Redundancy (TMR) are explored to protect FPGAs from soft error [10]. TMR-based mitigation techniques impose more than 200% overhead in terms of area and power. Scrubbing, i.e., the periodic refresh of the configuration memory, is another effective approach, especially when used in conjunction with TMR. Others Mitigation Techniques are multiple redundancy with voting, error detection and correction codes (EDAC), and FPGA-specific methods, such as reconfiguration, partial configuration, rerouting design.

1.3 Brief background work on asynchronous circuit

For last three decades, most circuits designed and fabricated are “synchronous”. However, as lock rates have significantly increased while feature size has decreased, clock skew has become a major problem. Most of performance chips must dedicate increasingly larger portions for clock drivers to generate acceptable skew, causing these chips to dissipate increasingly higher power, especially at the clock edge, when switching is most prevalent. With the process of these trends, the clock is becoming more and more difficult to manage, while clocked circuits’ inherent power

inefficiencies are emerging as the dominant factor hindering increased performance. These concerns caused renewed interest in asynchronous circuit. Asynchronous, clockless circuits require less power, generate less noise, and produce less electro-magnetic interference (EMI), compare to synchronous circuits, without degrading performance. Furthermore, these delay-insensitive (DI) asynchronous circuits have some additional advantages, especially when designing complex circuit, such as system-on-a-chip (SoCs), including substantially reduced crosstalk between analog and digital circuits, ease of integrating multi-rate circuits, and facilitation of component reuse. Recently, companies such as ARM, Phillips, Intel, and others are incorporating asynchronous logic into some of their products using their own proprietary tools.

Asynchronous circuits can be classified into two main categories: bounded-delay and delay-insensitive models. Bounded-delay models, such as micropipelines [12], assumed that delays in both gates and wires are bounded. Delays are added based on worse-case scenarios to avoid hazard conditions. This leads to extensive timing analysis of worse-case behavior to ensure correct circuit operation. Delay-insensitive circuits assume delays in both logic elements and interconnects to be unbounded, although they assume that wire forks within basic components, such as full adder, are isochronic, meaning that the wire delays within a component are much less than the logic element delays within the component, which is a valid assumption even in future nanometer technologies. Wires connecting components do not have to adhere to the isochronic fork assumption. This implies the ability to operate in the presence of indefinite arrival times for the reception of inputs. Completion detection of the output signals allows for handshaking controlling input wavefronts. Delay-intensive design styles therefore require very little, if any, timing analysis to ensure correct operation (i.e., they are correct by construction),

and also yield averge-case performance rather than the worse-case performance of bounded-delay and traditional synchronous paradigms.

CHAPTER II

ASYNCHRONOUS CIRCUIT ON FPGAS

The modern FPGAs use schematic design or a resource code in HDL (hardware description language) to program and configure circuit. The modern CAD tool can implement both synchronous and asynchronous circuits by using HDL. The previous soft error mitigation method are more focus on synchronous circuit implemented on FPGA, however, the asynchronous circuits, especially Null Convention Logic (NCL), implemented on FPGA have much easier detection signal when the soft error injected in asynchronous circuit implemented FPGAs. In order to analyze behavior of soft error injected in asynchronous circuit implemented FPGAs, the FPGA architecture, NCL circuit, and FPGA implementation NCL circuit are necessary understood and introduced.

2.1 FPGA architecture

FPGA is logic devices that contain basic components LUT (look-up-table), flip-flops, multiplexors, I/O blocks, programmable switching matrices, interconnect, clocks. These basic elements compose the functional blocks in FPGAs, such as configurable logic blocks, I/O blocks, and programmable interconnects. The FPGA structure is shown in Fig2.1.1 [13]. The two-dimensional arrays of generic logic elements (LEs) as shown in Fig.2.1.2 [13], and programmable switches are main elements specifying the functionality of circuit mapped into the FPGAs.

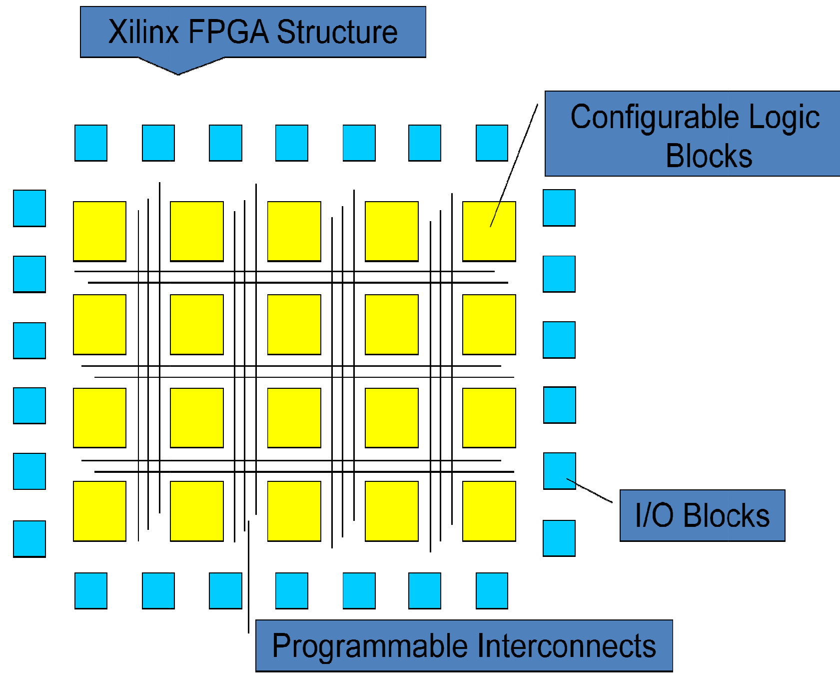

Fig 2.1.1 Xilinx FPGA structure [11]

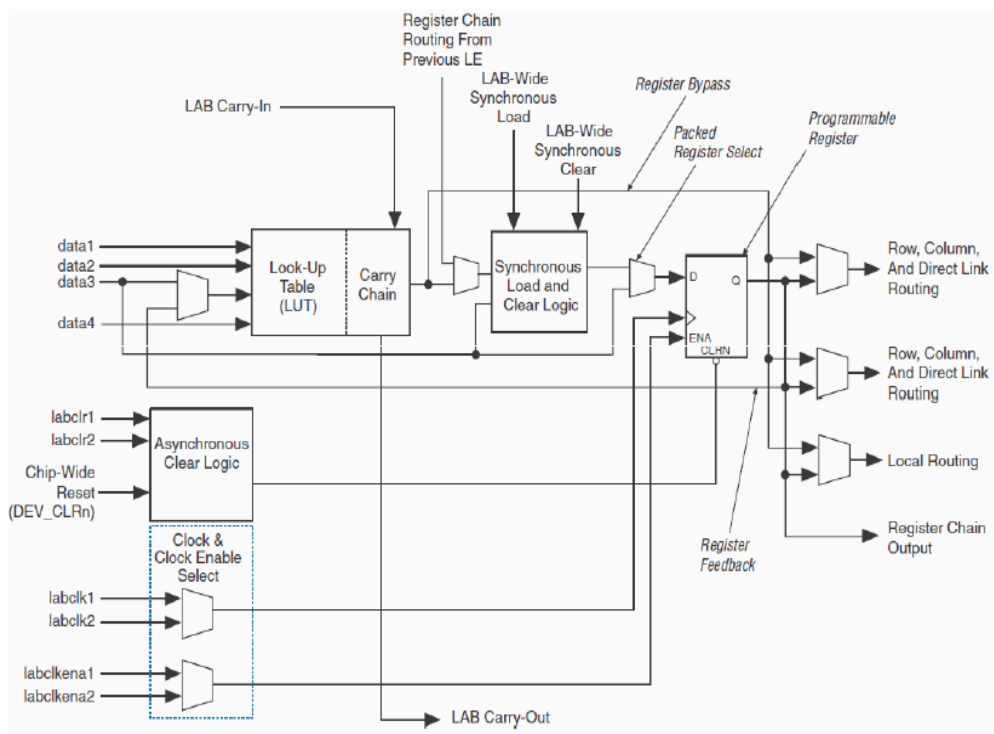

Fig 2.1.2 Logic Elements (LEs) [13]

A logic element can be configured (i.e., programmed) to perform a simple function, and a programmable switch can be customized to provide interconnections among the logic elements. A custom design can be implemented by specifying the function of each logic element and selectively setting the connection of each logic element and selectively setting the connection of each programmable switch. A logic element usually contains a programmable look-up table (LUT), programmable interconnects, and flip-flops (FF). An *n-input* look-up table is typically implemented by a static random access memory (SRAM), and is used to implement any *n-input* combinational function. The flip-flops can be selectively used to implement sequential circuits. Most FPGA devices also embed certain macro cells, such as BlockRAMs, dedicated multipliers, clock managers, and I/O interface circuits. Logic elements are usually grouped into logic array blocks (LABs).

For example, the architecture of Altera Cyclone II FPGAs is shown in Fig.2.1.3 [14]. The logic array consists of Logic Array Blocks (LABs), with 16 LEs in each LAB. Each LE includes a four-input LUT, a D-flip-flop and connections. The phase-locked loops (PLLs) provide general-purpose clocking with clock synthesis and phase shifting as well as external outputs for high-speed differential I/O support. The input/output elements (IOEs) contain a bidirectional I/O buffer and three registers for complete embedded bidirectional single data rate transfer. The devices also contain embedded memory and multipliers.

PLL	IOEs							PLL
IOEs	Logic array	Memory	Logic array	multipliers	Logic array	Memory	Logic array	IOEs
PLL	IOEs							PLL

Fig 2.1.3 The architecture of Cyclone FPGAs [14]

The architecture of an SRAM-based 4-input look-up table, implementing a logic function $f(D,C,B,A)$, is illustrated by Fig.2.1.4. The truth table of $f(D,C,B,A)$ is stored in the memory cells. For instance, binary data 1110_1000_1110_1000 is stored in the memory cells as shown in Fig.2.1.4, to implement logic function is $f(D,C,B,A) = AB + BC + AC$.

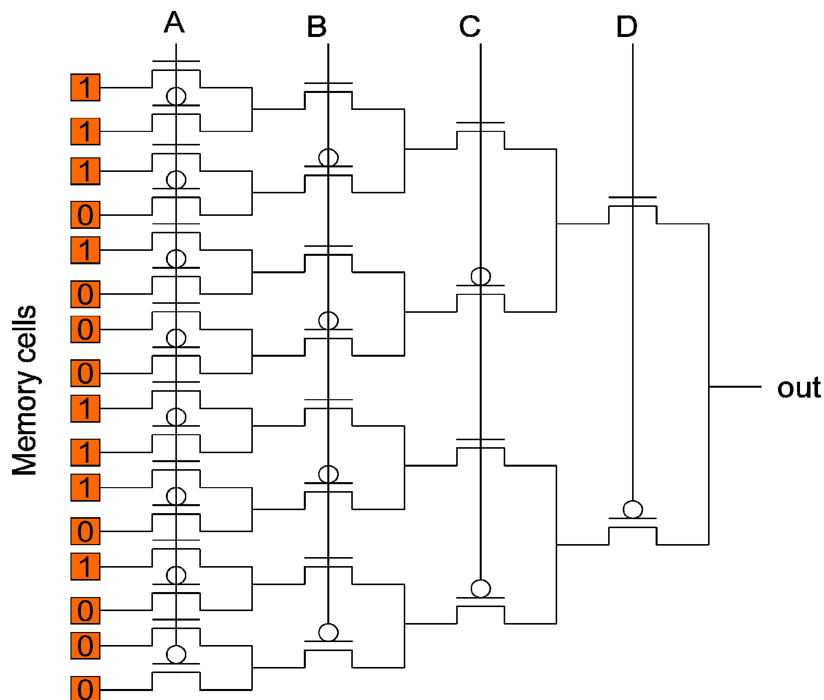

Fig 2.1.4 Architecture of SRAM-based 4-input look-up table (LUT)

These SRAM-based 4-input LUTs are used to implement digital circuits while switch modules provide the interconnection between resources, shown in Fig.2.1.5. The switch module consists of several switch blocks. The programmable switch blocks provide the selective connectivity of horizontal as well as vertical routing channels placed between logic blocks which consist of nets of different length [17]. Inside switch block, a pass transistor controlled by a user-SRAM cell play a role as programmable switches. Each of this architecture with pass transistor and user-SRAM cell is called Programmable interconnect points (PIPs). Consider the matrix shown in Fig. 2.1.6, three different nets are PIPs (W1, N1), (W2, S1), (S2, E2). The PIPs can control the net open or close. The light line shows the possible connection nets, while the PIP is closed.

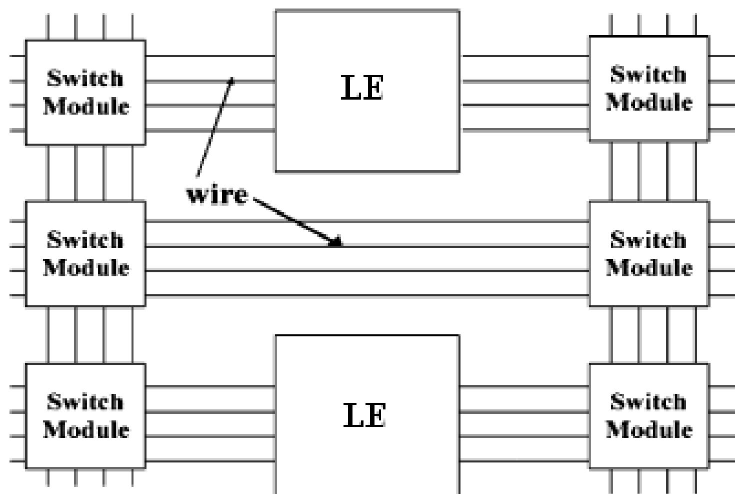

Fig 2.1.5 SRAM-FPGA architecture model [22]

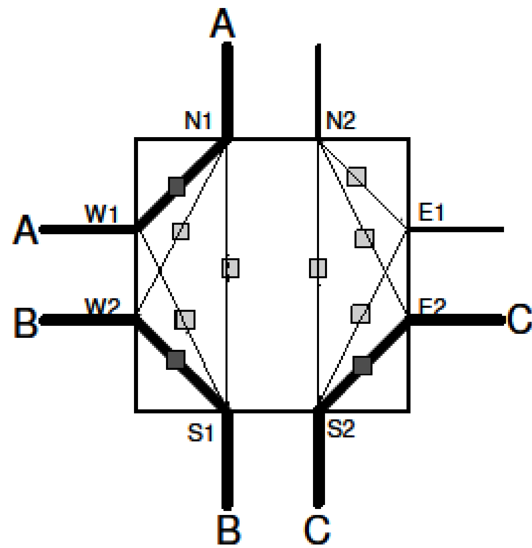

Fig 2.1.6 PIPs control in switch block [23]

2.2 Null Convention Logic (NCL) circuit

Null Convention Logic (NCL) [16] is a delay-insensitive asynchronous paradigm meaning that NCL circuits will operate correctly regardless of delay of components and wires. NCL circuits utilize dual-rail or quad-rail logic to achieve delay-insensitivity. For this thesis, the designs employ the dual-rail logic.

2.2.1 Completion Criteria

NCL uses two completeness criteria to achieve its delay-insensitive behavior: symbolic completeness of expression and completeness of input. A symbolically complete expression is defined as an expression that only depends on relationships of the symbols presented in the expression. Dual-rail signals state logic value (NULL, DATA0, and DATA1) achieve symbolic completeness of expression. A dual-rail signal D is encoded by two wires D^1, D^0, as shown in Table 2.2.1.

Table 2.2.1 dual-rail encoding

Dual-rail encoding (D^1, D^0)	**Logic value**
(0,0)	NULL
(0,1)	DATA0
(1,0)	DATA1
(1,1)	Invalid

The DATA0 state ($D^1 = 0, D^0 = 1$) corresponds to a Boolean logic 0, the DATA1 state ($D^1 = 1, D^0 = 0$) corresponds to a Boolean logic1, and the NULL state ($D^1 = 0, D^0 = 0$) corresponds to empty set meaning that the value of D is not available. Two rails are mutually exclusive, such that both rails can never be asserted simultaneously, this state is defined as an illegal state.

The second criterion is completeness of input for NCL combinational circuit, 1) the output may not transition from NULL to a complete set of DATA until the input values are completely DATA, 2) the output may not transition from DATA to a complete set of NULL values until the input values are completely NULL. The criterion, equivalent to Seitz's "weak condition" [17], is shown in Fig 2.2.1.1.

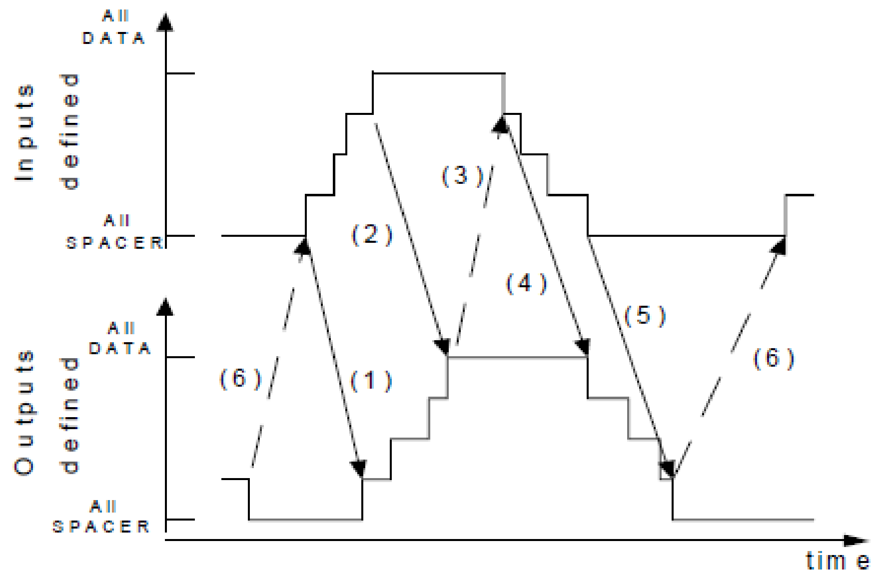

Fig 2.2.1.1 weak conditions for NCL completeness of input

(1) Some inputs become DATA before some outputs become DATA.

(2) Some inputs become NULL before some outputs become NULL.

(3) All inputs become DATA before all outputs become DATA.

(4) All outputs become DATA before some inputs become NULL.

(5) All inputs become NULL before all outputs become NULL.

(6) All outputs become NULL before some inputs become DATA.

2.2.2 Threshold Gates with Hysteresis

NCL uses a special type of gates, namely threshold gates with hysteresis [18], as basic units to build NCL circuits. The format that describes threshold gate is $\mathrm{thmnWn_1n_2n_3 \ldots . n_w}$, where 'th' is stand for threshold gate, m is the threshold, n is the number of inputs, W means the following 'n_1', ' n_2'… 'n_w' are weights of the first 'W' inputs and the weights of other inputs are one by default. The following figures are schematic and symbol of TH23 and TH23w2.

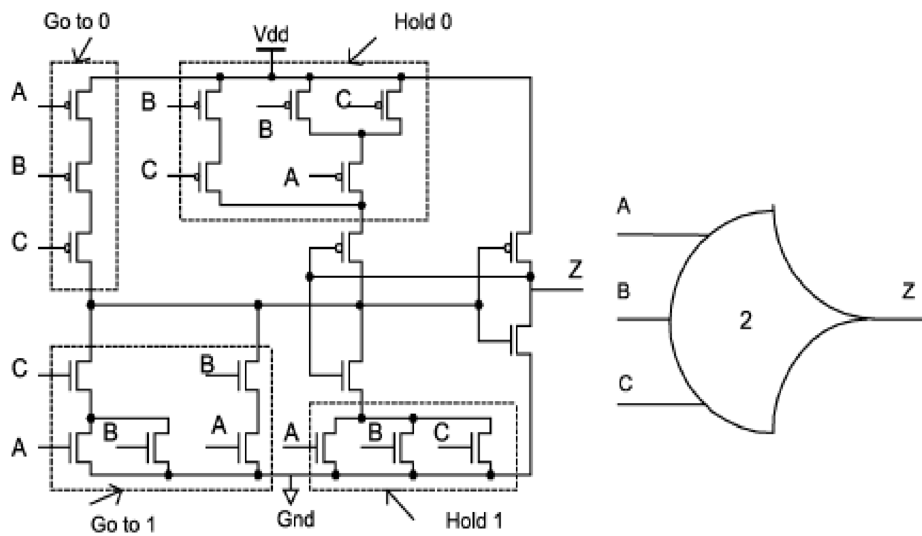

Fig 2.2.2.1 Schematic and symbol of TH23

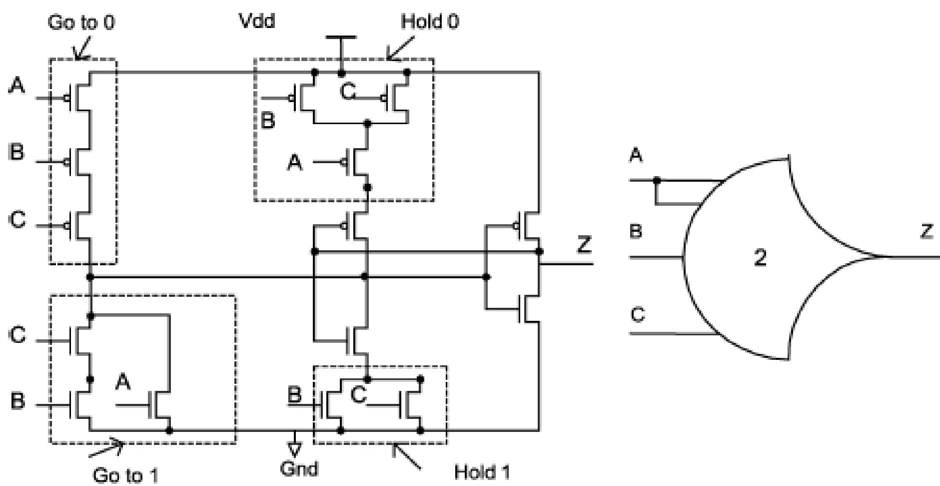

Fig 2.2.2.2 Schematic and symbol of TH23W2

NCL circuits are comprised of a family of threshold gates with hysteresis. The primary type of threshold gate is TH*mn* gate where *n* is the number of inputs, *m* is the threshold, and $1 \leq m \leq n$. A THmn gate will set its output high when any *m* inputs have gone high and it will reset its output low when all *n* inputs are low. A more general type of threshold gate with hysteresis is referred to as a weighted threshold gate, denoted as TH*mn*W$w_1w_2...w_R$, where *n* is the number of inputs,

m is the threshold, $w_1, w_2, \ldots w_R$ $(1 < w_i \leq m, 1 \leq R < n)$ are the integer weights of *input 1*, *input 2*, … *input R*, respectively. For example, TH*34* has 4 inputs (A, B, C, D) and a threshold of 3, as shown in Fig 2.2.2.3 (a). When any three inputs go high, its output will be asserted to high. Only when all inputs are low, the output will be reset to low. For all other input patterns, the output will remain unchanged. A weighted gate TH*34*W*22* has the same number of inputs (4) and threshold (3) as TH34 gate, but there is a weight 2 applied to each of the first two inputs (A and B), as shown in Fig 2.2.2.3 (b). For the gate TH*34*W*22,* the output is asserted only when either input A is high along with any other input, or input B is high along with any other input. The output is deasserted only when all inputs are low. NCL threshold gates may also include a reset input to initialize the output. Either a *d* or an *n* is attached at the end of the gate name to designate these gates, such as TH*22n* shown in Fig 2.2.2.3 (c). *d* denotes the gate as being reset to high while *n* to low. These resettable gates are used in the design of registers. A bubble attached at the output denotes an inverter connected at the output, as shown in Fig 2.2.2.3 (d). The principle of transistor-level threshold gate design can be found in [18]. The design of computational blocks, registers and completion detection blocks using threshold gates is available in [18].

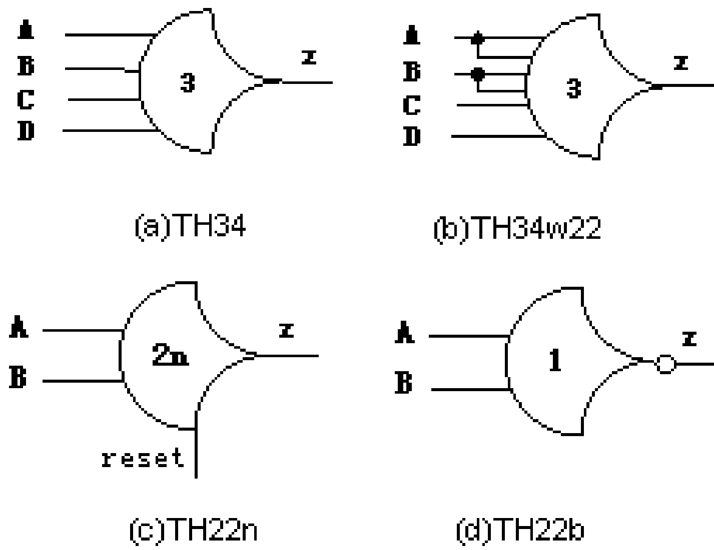

Fig 2.2.2.3. Symbol examples of threshold gates

2.2.3 NCL Pipeline

The framework for NCL systems consist of delay-insensitive combinational logic sandwiched between delay-insensitive registers. This combination of NCL registers along with completion detection circuitry and combinational logic is called NCL pipeline [17]. A typical NCL pipeline architecture consists of NCL registers, completion detection circuitry and NCL combinational logic like exor, full-adder,etc. the following Fig 2.2.3 shows basic NCL pipeline structure.

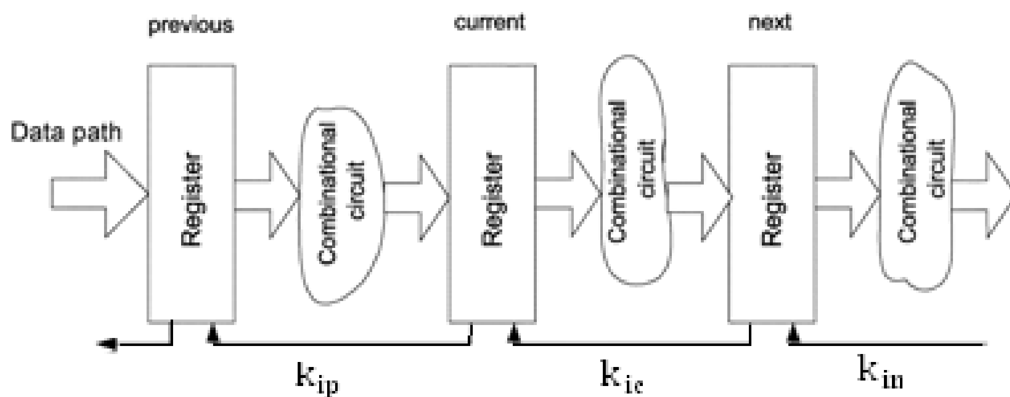

Fig 2.2.3 Basic NCL pipeline structure

The NCL allows DATA and Null pass through the NCL pipeline alternately. The input request for each register gate comes from the detection gate of the next register. Two adjacent register stages interact through their request and acknowledge signals $k_i, k_o,$ respectively, to prevent the current DATA wavefront from overwriting the previous DATA wavefront, by ensuring that the two DATA wavefronts are always separated by a NULL wavefront. For example, assume that all the circuits are in a NULL state and that input request signals of the current register (k_{ic}) and the next register (k_{in}) are requesting DATA, at same time, the previous register is transiting a complete DATA to its combinational circuit. As the wavefront propagates through the previous combinational circuit to the current register, the current register passes the data since its control line is requesting DATA. When a complete data is received by current detection circuitry, it gives control line (k_{ip}) to the previous register to indicating that the current register has received and stored the data wavefront, thus, the previous registers can pass NULL. The requested NULL by previous register may arrive at the current register, however, the NULL wavefront will be blocked at current register until control line (k_{ic}) is requesting NULL, and the current register will maintain presentation DATA values to the current combinational circuit. When the next register receives and stores the DATA wavefront, the DATA set no longer needs to be maintained by the current register. Since the next completion detection circuit detects the complete DATA set and transitions its acknowledge line (k_{ic}) to request NULL indicating that it has received DATA wavefront and the current register can allow a NULL wavefront.

2.2.4 NCL register

An N-bit register stage, comprised of N single-bit dual-bit NCL registers, requires N completion signals, one for each bit. The NCL completion component, shown in Fig 2.2.4, uses these Nk_o lines to detect complete DATA and NULL sets at the output of every register stage

and request the next NULL and DATA set, respectively. In full-world completion, the single-bit output of the completion component is connected to all k_i lines of the previous register stage. Since the maximum input threshold gate is the TH44 gate, the number of logic levels in the completion component for N-bit register is given by $[\log_4 N]$. Likewise, the completion component for an N-bit quad-rail registration stage requires $\frac{N}{2}$ inputs, and can be realized in a similar fashion using TH44.

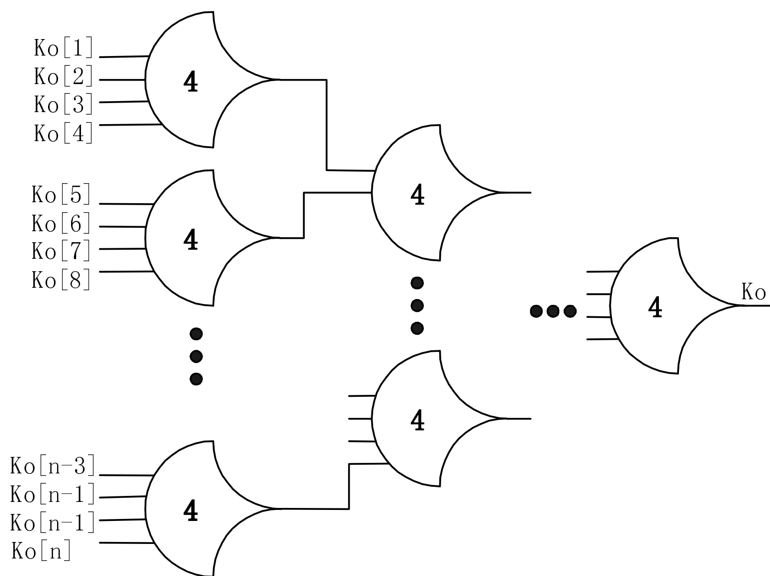

Fig 2.2.4 N-bit completion component

2.3 FPGA implementation of NCL circuits

Generally, a threshold gate is synthesized into three logic elements in FPGAs, as shown in Fig 2.3.1 Each logic element, utilizing an SRAM LUT, performs *Set, Reset, and Hold* respectively. The function of *Set LUT* is defined as this: its output t1 is "1" when the number of inputs equal to "1" reaches (or more than) the threshold *m*; otherwise the output is "0". For example, the *Set* function of TH23 is $t1 = AB + BC + AC$, and the *Set* function of TH34w2 (A has a weight of 2) is $t1 = A(B + C + D) + BCD$. The function of *Reset LUT* for all threshold gates is an OR gate

delivering “0” when all inputs are “0”. The *Hold LUT* delivers the final output with hysteresis using a feedback. It is easy to find out that its function is $z = t2 \cdot (t1 + z)$ for all threshold gates.

The Quartus II provides the way to look at schematic of design. By looking at the schematic of design at software, it shows three LUT to implement Threshold gate. Each LUT may restore the truth table to generate function of threshold gate. The concept in Fig 2.3.1 has been verified by implementing TH23 and TH34w2 gates on Altera Cyclone II EP2C35F672C6N chip, as shown in Fig 2.3.2.

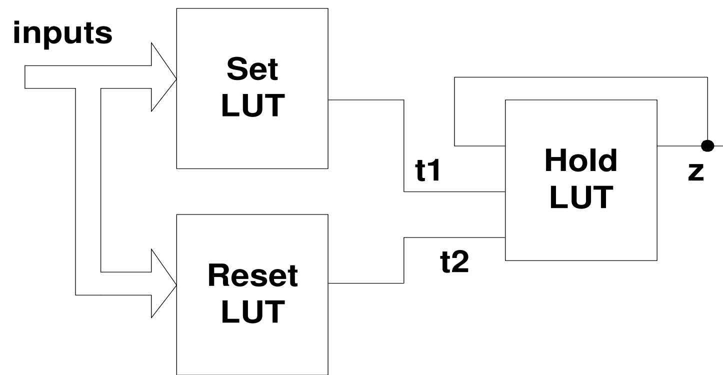

Fig 2.3.1 Threshold gate implemented on FPGAs

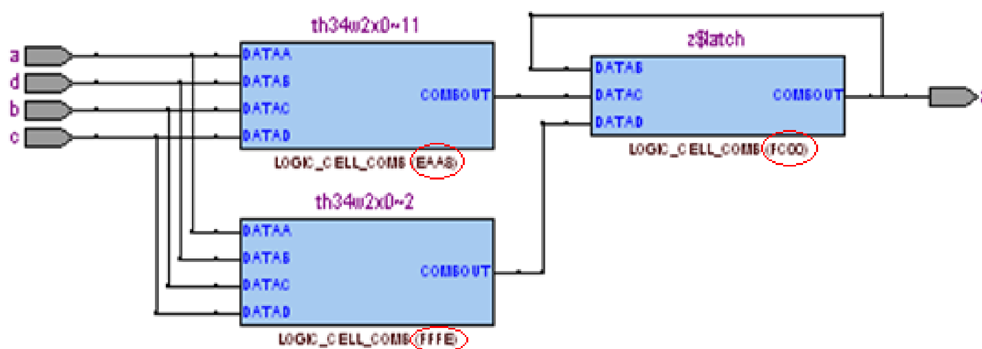

Fig 2.3.2. TH34w2 gates on Altera Cyclone II

The Fig 2.3.2 shows Th34w2 gates implemented in Altera Cyclone II FPGA by Quartus II software. It has 4 inputs going to Set LUT (th34w2x0~11) and Reset LUT (th34w2x0~2). The Hold LUT (z$latch) delivers final output with hysteresis using output feedback and output of set LUT and reset LUT. The red circle is the value of sum LUT mask. The sum LUT mask is simply the hexadecimal representation of the LUT output. Take set LUT for example: the function equation of set LUT is $t1 = A(B + C + D) + BCD$, so that the truth table of set LUT can be described in Table 2.3:

Table 2.3 truth table of set LUT

D	C	B	A	OUTPUT
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

From this truth table the set LUT function can be represented by following binary sting: 1110_1010_1010_1000, the sum LUT mask in hexadecimal for this binary string is : EAA8. Similarly, the reset LUT and hold LUT mask can be found by this way too.

CHAPTER III

SOFT ERROR MODELS AND EXPERIMENTAL FAULT INJECTION

In the previous chapter, basic works on asynchronous circuit implemented in FPGAs have been done. With help of these basic works, effects of soft error injecting asynchronous circuit implemented FPGA are obvious to observe. Therefore, the soft error models and experimental fault injection method are concluded as part of this research and explained in CHAPTER 3. This chapter explains soft error models in FPGAs and possible experimental fault injection method are described.

3.1 Soft error models in FPGAs

The effects of single event upsets (SEUs) on digital circuits can be classified into 1) transient and 2) permanent errors. The transient errors are caused by SEUs hitting on combinational logic components, which can be propagated and captured in flip-flops. These errors are called transient errors because they may be corrected using detection techniques. These errors could be occurred and found in memory elements and caches, register files and flip-flops (FFs). The permanent errors are caused by SEUs hitting on memory bits, and these errors alter the contents of configuration bits. This case may keep erroneous until the new configuration is downloaded and rewrited into the FPGA.

The research and analysis of transient errors have been described in [18]. They inject faults into the simulation or emulation models of the design to investigate the behavior of combination logic circuit. The alteration of memory elements such as data-path registers and control-unit registers are injected by faults with faults injection techniques [20].

The research of permanent errors that changes contents of configuration bits are more complex to analyze, since the simple bit flip fault model cannot be exploited. An SEU can changes interconnect inside configurable logic block (CLBs) and also the routing signals between different CLBs. Moreover, a SEU may change the content of look-up-tables (LUTs).

There are two major types of memory resources in FPGAs: user bits and configuration bits. An SEU on user bits cause a transient error, and an SEU on configuration bits leads to a permanent error.

3.1.1 Transient errors

The transient errors do not change the content of SRAM configuration bits, but affect user-defined logic and flip-flops. An SEU affecting a combination part makes transient error on the combinational part inside CLBs. These errors can be propagated to next part to make a bit-flip error. The Fig 3.1.1.1 describes SEU makes bit-flip error in a flip-flop. It has been shown that in ASIC designs, combinational logic is less susceptible to soft error than memory elements [21]. This is because the combinational logic provides some natural resistance to soft errors [22].

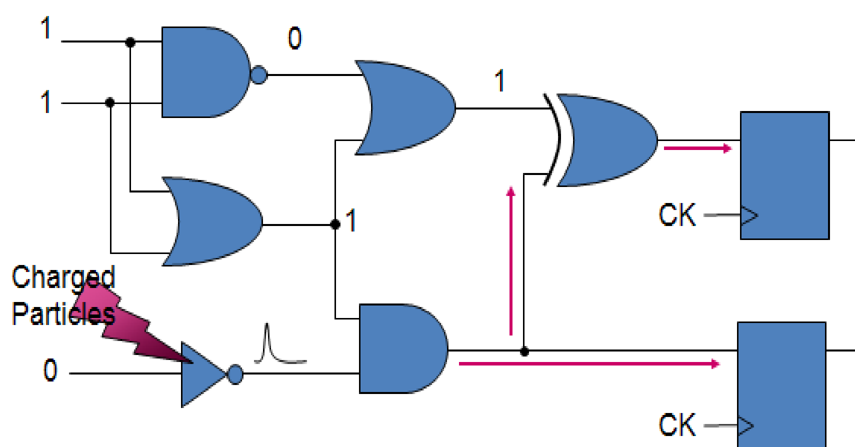

Fig 3.1.1.1 An SEU affects inverter gates and makes a bit-flip error

An SEU may also affect the contents of flip-flop and memory. The content of the Flip-Flops (FFs) and memory will keep erroneous until another data write to FFs or corrected by detecting circuit. The Fig 3.1.1.2 is shown the SEU hit on FF and Block RAM.

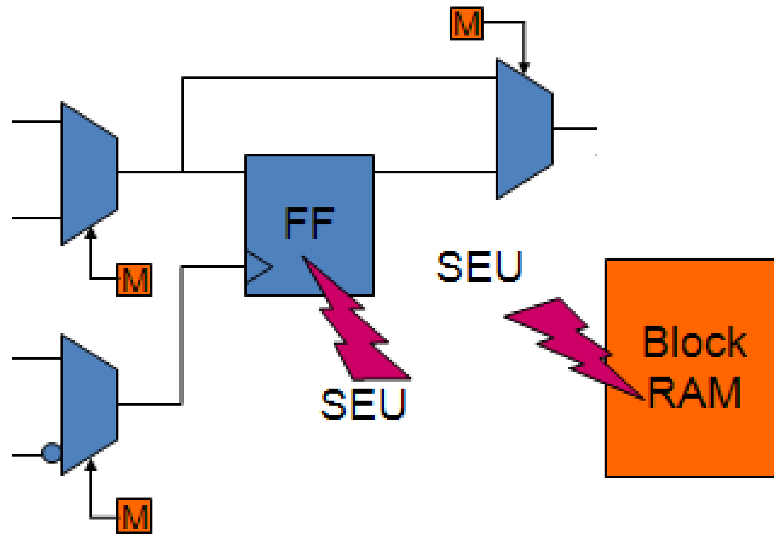

Fig 3.1.1.2. SEU hit on FF and Block RAM in FPGA

3.1.2 Permanent error

A single event upset (SEU) induced by a particle strike in a user bit causes a transient error, whereas an SEU in a configuration bit would lead to a permanent error which remains in the FPGA until the next reconfiguration of a new design. This permanent error may result in a logic error or routing error depending on which part of the configuration memory is affected. A logic error may lead to complement one of the entries of the LUT modifying the functionality of the mapped logic function. A routing error may lead to a signal getting misrouted or disconnected [23]. This type of error is the major error type in FPGAs because the number of SRAM cells

dominates user-defined memory elements. Typically, the amount of SRAM configuration cells is more than 98% of all memory elements inside an FPGA [23].

There are two types configuration memory bits, are sensitive and non-sensitive bits, based on their vulnerabilities to SEUs. When the SEU hit at the sensitive configuration bits, it affects the functionality of the circuit which mapped into FPGA. On the other hand, the non-sensitive configuration bits play role as “don’t care” for the design mapped into FPGA.

Since a transient error is not lasting ever and easy to be detected by detection correcting circuit, permanent error is more difficult to correct. Our research is more focus on permanent error. Permanent error can be classified into routing errors, LUT bit-flips, and control/clocking bit-flips.

3.1.2.1 Routing error

The programmable routing network of FPGA consist of Programmable interconnect points (PIPs), multiplexers and buffers. An SEU changing a configuration routing bit causes PIPs switch open, switch short. Normally, the PIPs employ the NMOS transistor as switch. Since the SRAM cell control the NMOS transistor with logic 1 close, the changing a PIP control from 1 to 0 by SEU will cause a switch open in this nets resulting in an open error in the gate-level netlist, is shown in Fig 3.1.2.1.1. It’s easy to see the close connection nets break out by the changing of PIP control cell from 1 to 0. However, the short error is a little bit different with open error. An SEU (0 to 1) on unused PIP (W1,S1) or (N1,S1) causes a bridging error between nets A and B. it’s called short error. On the other hand some unused PIP (N2, E1) not cause any bridging errors since no nets are adjacent to this PIP. Therefore, we may define PIP as two categories, sensitive PIP and non-sensitive PIP. The Fig 3.1.2.1.2 shows the sensitive PIP and non-sensitive PIP in short error case.

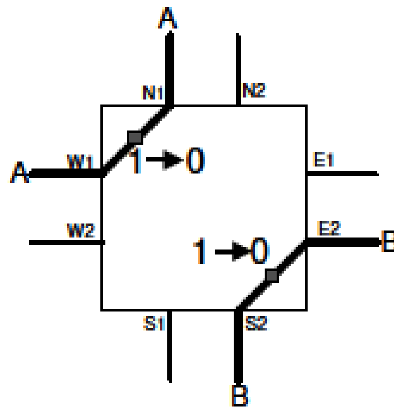

Fig 3.1.2.1.1 open errors in switch blocks

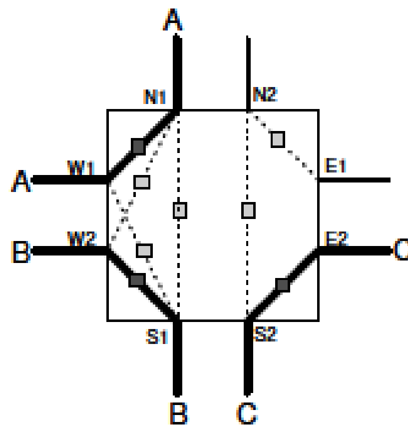

Fig 3.1.2.1.2 short errors in switch blocks

3.1.2.2 Bit-flip on Look up Table (LUT) configuration bits

As the previous chapter's illustration, a logic function can be implemented by storing all values for the truth table in LUT. The look up table (LUT) consists of SRAM cell. A SEU may lead to complement one of the entries of the LUT modifying the functionality of the mapped logic function permanently. Fig 3.1.2.2 the SRAM cell changes its stored value by SEU.

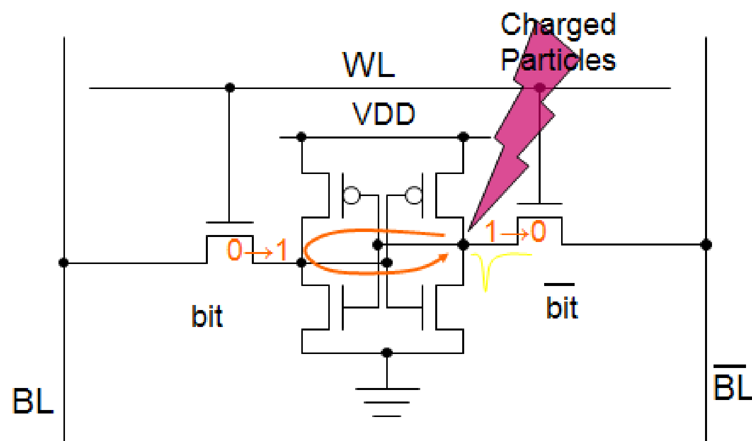

Fig 3.1.2.2 SEU hits on SRAM cell

3.1.2.3 Bit-flip on control/clocking bits

There are some control bits in CLBs and I/0 blocks to determine miscellaneous functionalities. For example, the control bits may determine whether the LUT performs as a look up table, a dual-ported RAM, or a programmable shift register. Furthermore, some SRAM cells also works for clock signal routing throughout the circuit. In summary, bit-flip on the control/clock bits affects the functionality of the mapped design drastically [24].

3.2 **Fault injection method**

The soft error propagation both in signal NCL gate and NCL pipeline has been derived in next chapter. In order to verify our analysis and conclusion of soft error, the simulation with soft error injected environment must be generated, so that the fault injection techniques are employed to reach our aim. In modern society, the approaches to fault injection have been classified by three categories:

- Hardware fault injection
- Software fault injection
- Simulation fault injection

The hardware fault injection cause faults in circuit physically. The advantages of hardware fault injection are realistic good fault coverage, good for low latency faults, and low perturbation. On other hand, it's harder to control and expensive to build up system in safety limit their widespread use. The hardware injection with contact uses pin-level injection such as, probes, socket insertion, but without contact uses heavy ion radiation and electromagnetic interference. The common used hardware tools are Messaline, FIST and MARS. The software fault injection injects the soft error by software programmatically. It has compile-time injection and runtime injection. The software injection is cheaper, controllable and more targeted and flexible, but it's more perturbation and limited coverage. The common software tools are Ferrari, Ftage, Xception. The simulation injection injects faults during the simulation time. It's not closed to real environment.

The fault injection in this thesis is performed by Quartus II software properties resource editor. It also can be called software fault injection. Quartus II provides an easy way to change the content of a desired LUT SRAM cell after compiling the VHDL files. This change is equivalent to a fault injection to the cell. After saving the change, it can simulate the behavior of the circuit with a fault injection. The Fig 3.2.1 shows the Quartus II design flow with fault injection.

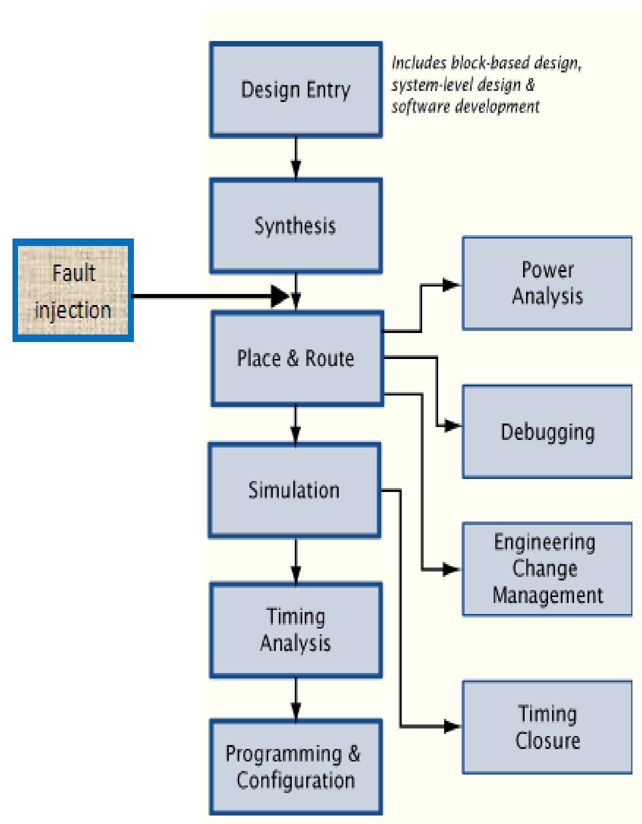

Fig 3.2.1 Quartus II design flow

From Quartus II tools, it provides lots of powerful and useful tools, such as in system content memory editor, in system source and probes editor and so on. One of them is called net viewers allow you to view schematic representations of internal structure of your designs. Fig 3.2.2 shows netlist viewers in tool box.

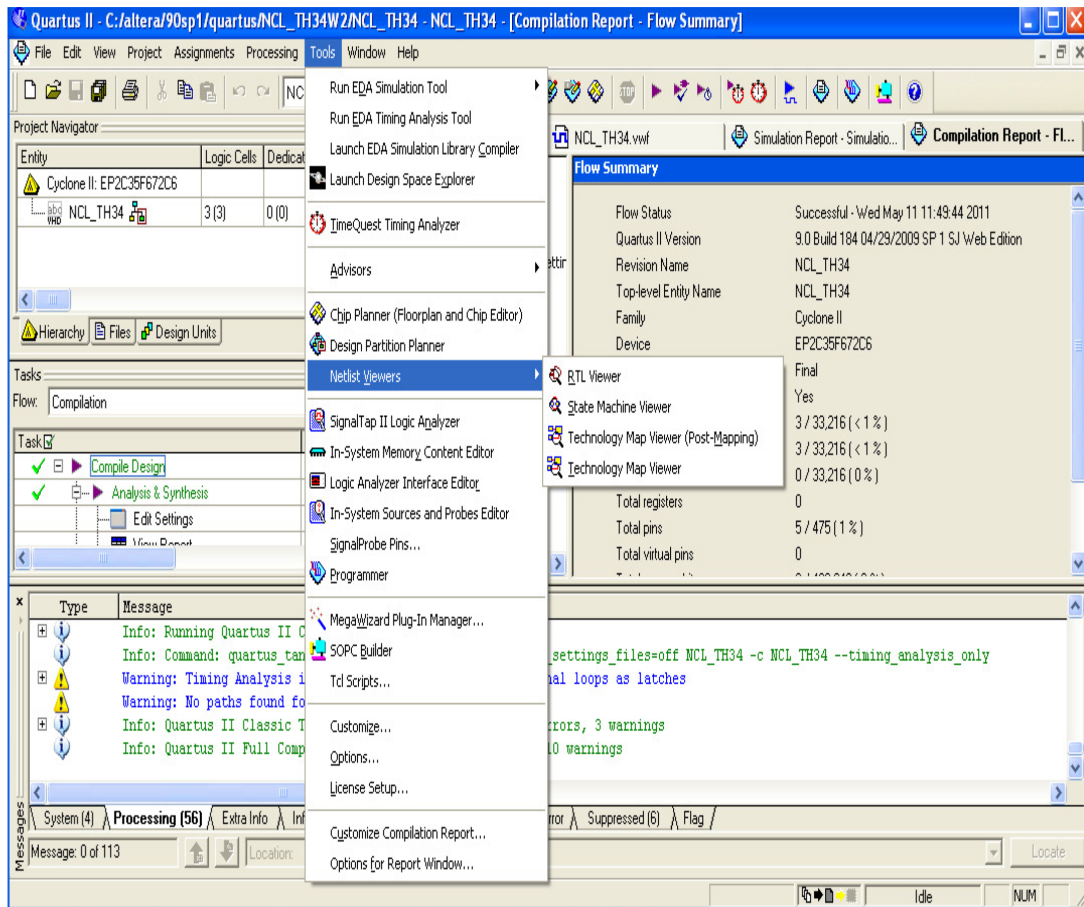

Fig 3.2.2: Netlist viewer in tool button

If you are ready find out the netlist viewer button as shown in Fig 3.2.2, the software will provide four different functional views ask you choose. The RTL viewer can view a schematic of the design netlist after Analysis & Elaboration and netlist extraction, but before Quartus II synthesis and fitting optimizations. This view is not the final structure of the design, because not all optimizations are included; instead it is the closest possible view to the original design. If the design uses integrated synthesis, this view shows how the Quartus II software interprets the design files; if you are using a third-party EDA synthesis tool, this view shows the netlist as written by the EDA synthesis tool. The Technology Map Viewer can view a low-level, technology-specific schematic of the design netlist after fitting or after Analysis & Synthesis.

You can access the post-fitting view of the schematic with the Technology Map Viewer; or you can access the post-mapping view of the schematic, regardless of the synthesis tool you use, with the Technology Map Viewer (Post-Mapping). When opened from a timing path in the Timing Analyzer report, the Technology Map Viewer also displays detailed timing delay information for the timing path. The State Machine Viewer provides a high-level view of finite state machines in the design and displays the internal structure of state machines in the design, including a more detailed view of the fan-in and fan-out of individual state nodes. The State Machine Viewer also displays the node transitions in table format. We use the technology map viewer to find out schematic of our design. With help of this function we can find out the TH34w2 is implemented on three LUTs: Set LUT, Reset LUT, and Hold LUT, as shown in Fig.3.2.3. The K-map, truth table, and logic function are available for each LUT in *Properties* pop-up window. The "Resource property editor" can be used to change the logic function of any LUT so that the K-map and truth table are changed accordingly as desired is shown as Fig 3.2.4. The resource property editor has a block to show and edit equation provides possibility to change the truth table by editing sum equation, is shown as Fig 3.2.5, thus, we can achieve fault injection by changing the truth table.

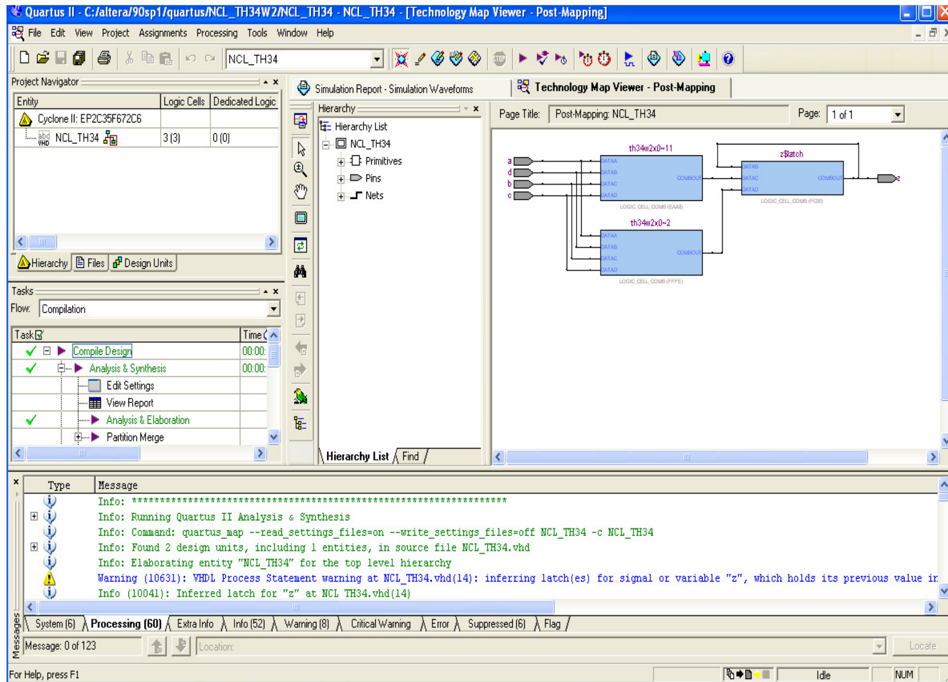

Fig 3.2.3 Technology Map Viewer of TH34w2 gate

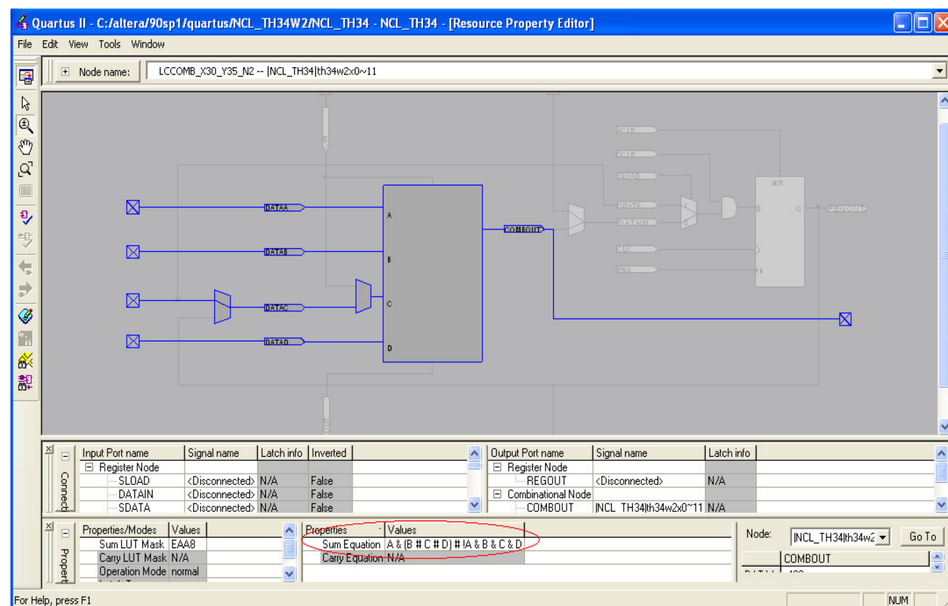

Fig 3.2.4 Resource property editor for fault injection

CHAPTER IV
SOFT ERROR IN FPGA-BASED ASYRCHRONOUS CIRCUIT

The FPGAs are widely used in many applications, however, the sensitivity of soft error limit their used in mission-critical application. When soft error is generated by alpha particles hitting FPGAs, the digital circuit which is implemented on FPGAs has totally changed. Most previous research on soft error of synchronous circuit implemented FPGA presents effect of soft error infected FPGA and also gives hints to possible detection method for soft error in FPGA [23]. As the previous chapter describes NCL circuits implemented in FPGA are different with synchronous circuit, soft error in FPGA-based asynchronous circuit has to be investigated. In this chapter, we investigate the mechanism of soft error generation and propagation in asynchronous circuits which are implemented on FPGAs

4.1 Soft error generations and simulation in a single threshold gate

As previous chapter explained, there are so many soft errors models; we only focus on the permanent logic error due to bit-flips in LUT configuration bits. It is assumed that a particle strike leads to the change of a configuration bit in LUT. Under this assumption, the implemented circuit such as a threshold gate has been changed into an undesired circuit. The TH34w2 gate is used as an example to show how and what kind of soft errors may be generated at the output of a threshold gate. TH34w2 is mapped into a 4-input *Set LUT*, a 4-input *Reset LUT* and a 3-input *Hold LUT* with feedback, as shown in Fig.4.1.1. Each entry in LUT is associated with a cell in SRAM. A particle strike may randomly lead to the data flip in a cell.

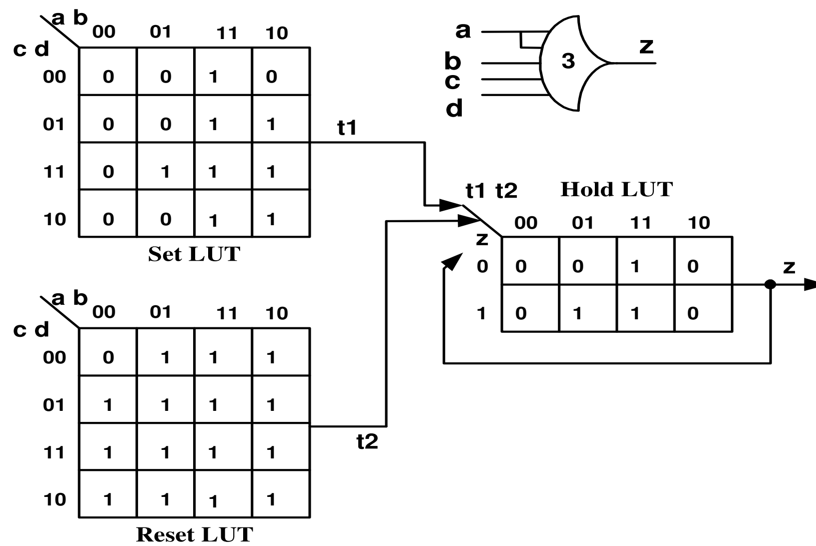

Fig. 4.1.1 TH34w2 implemented on FPGA

The function set LUT is bringing 'fire' under the condition that number of inputs equal to '1' is greater than or equal to the threshold *m,* otherwise there is no fire at output. Once the SEU generates at set LUT, the output may not fire correctly. The effect of soft error at set LUT is defined as these:

- **No error**, the SEU happened at set LUT: 0000 (0→1), the error signal transmits to hold LUT. Because of reset LUT gives reset signal ("0") to hold LUT, the SEU will not lead to any error. The Fig 4.1.2 shows simulation results of no error at set LUT 0000.

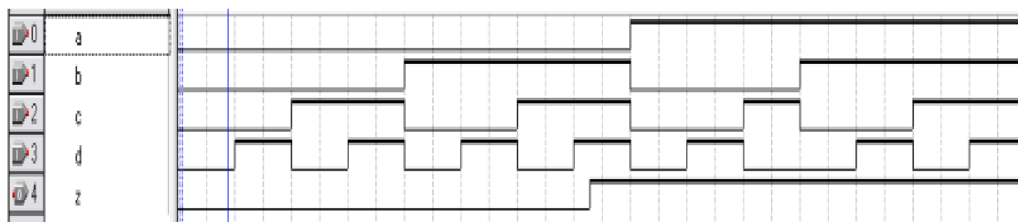

Fig 4.1.2 no error appears at output Z (soft error setting set LUT: 0000 (0→ 1))

- **Premature fire**, some SEUs, such as set LUT: 0001,0010,0011,0100,0101,0110,1000 (0→1) will generate fire ('1') to hold LUT, despite the number of inputs does not satisfy the threshold *m*. These SEUs may lead to premature fire, as shown in Fig4.1.3

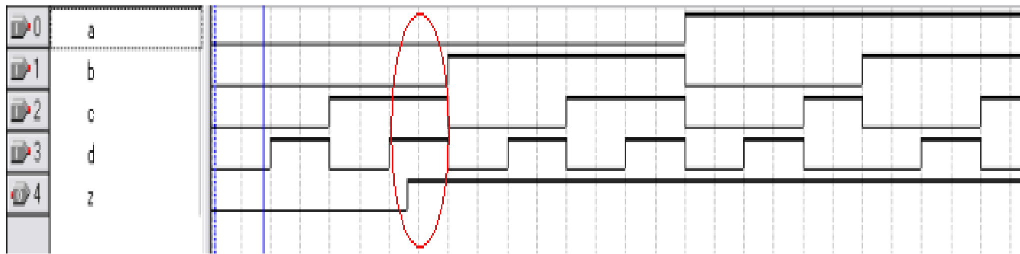

Fig 4.1.3 Premature fire appears at output Z (soft error setting set LUT: 0011 (0→ 1))

- **No fire**, on opposite side, others SEUs in set LUT (1→0) hold the fire. These may lead to no fire, despite the number of inputs satisfies the threshold *m*. The simulation results are shown in Fig 4.1.4.

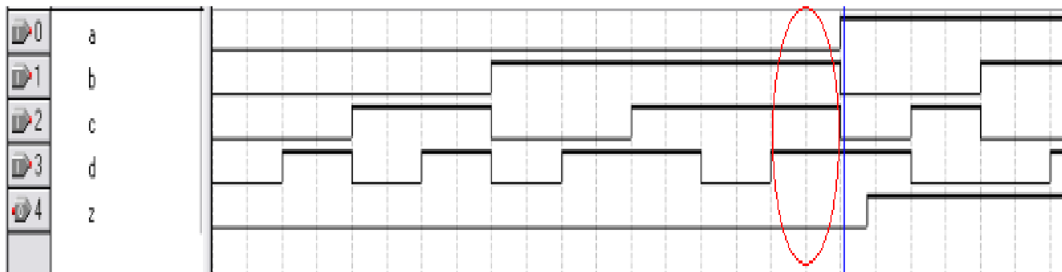

Fig 4.1.4 No fire appears at output Z (soft error setting set LUT: 0111 (1→ 0))

The reset LUT functionalizes OR gate to achieve reset process. It delivers "0" when all inputs are "0". The SEUs at reset LUT may generate wrong reset signal to hold LUT. The effect of soft error at set LUT is defined as these:

- **No return to 0**, the SEU of reset LUT: 0000 (0→ 1) changes function of reset LUT to a constant "1". The reset LUT keeps "1", even during the process of reset delivering "0". This SEU leads to no return to 0 is shown as Fig 4.1.5.

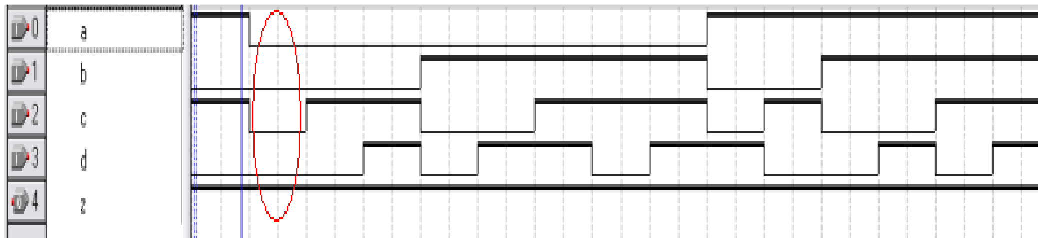

Fig 4.1.5 No return to 0 appears at output Z (soft error setting Reset LUT: 0000 (0→ 1))

- **Early return to 0**, some SEUs, such as reset LUT: 0001,0010,0011,0100, 0101,0110,1000 (1→0) give wrong signal delivering "0" to hold LUT, though the only one condition of reset process delivering "0" is not satisfied, as shown in Fig 4.1.6.

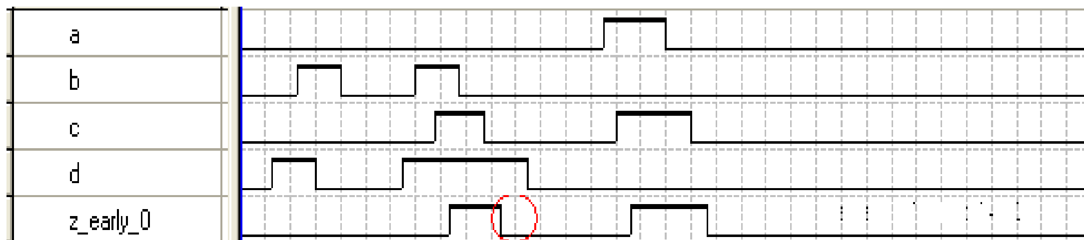

Fig 4.1.6 Early return to 0 appears at output Z (soft error setting Reset LUT: 0001 (1→ 0))

- **No fire**, SEUs at others reset LUT cause no fire at output. At this case, the TH gate default the 0000 signal already arrives reset LUT startup the reset process and the fire keep on set LUT may be considered as time-delay case too. These SEUs may lead to no fire that is totally different from no fire at set LUT. The simulation is shown as fig 4.1.6

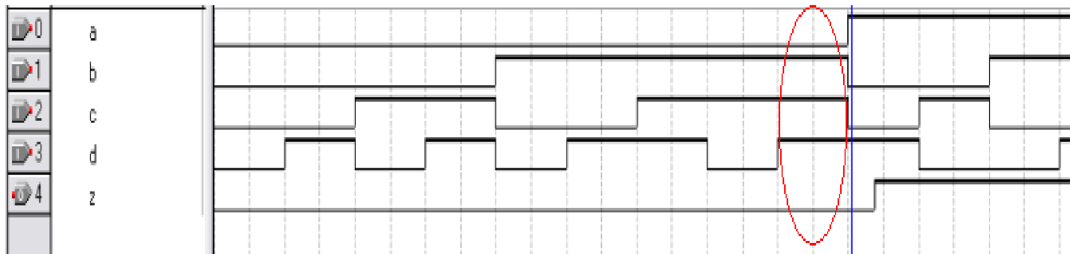

Fig 4.1.7 No fire appears at output Z (soft error setting Reset LUT: 0111 (1→ 0))

Hold LUT is more difficult to analyze compared to set and reset LUT. The reason is that hold LUT employs output of system as its input to achieve delay-insensitivity. The variable effects of SEU at hold LUT are defined as following:

- **Oscillating**, the SEU happened at hold LUT: 000 (0→1) 111 (1→ 0), the soft error propagates to output and comes back as its input, thus, the output could get value which is opposite with current output from LUT. When the new value comes to output again, it leads to output oscillating, is shown as Fig 4.1.7

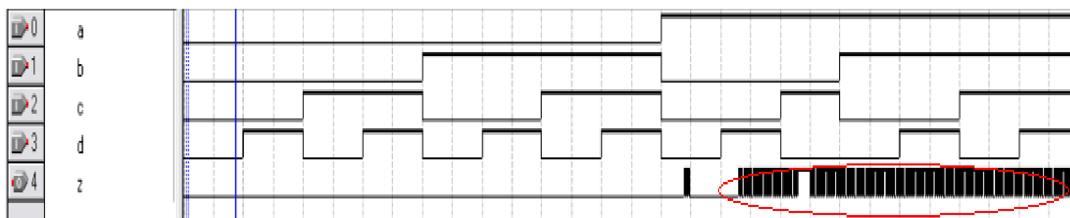

Fig. 4.1.8 oscillating appears at output Z (soft error setting hold LUT: 111 (1→ 0))

- **No return to 0**, SEU of hold LUT 001 (0 → 1) leads to no return to 0 with full satisfied condition of reset process which is that set and reset LUT all stay at “0”. The Fig 4.1.8 shows simulation.

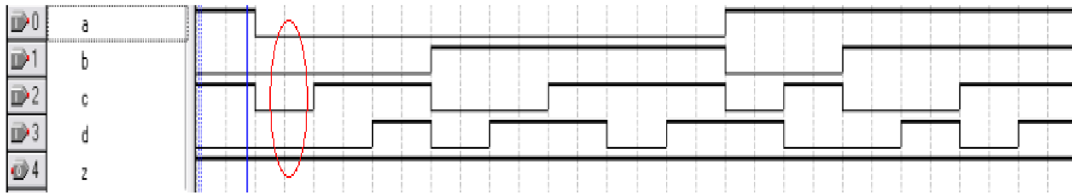

Fig 4.1.9 No return to 0 appears at output Z (soft error setting hold LUT 001(0 → 1)

- **Premature fire**, SEU at hold LUT 010 (0→ 1) cause the fire at condition of number of inputs does not satisfy the threshold *m* in set LUT. The simulation is shown as Fig 4.1.9

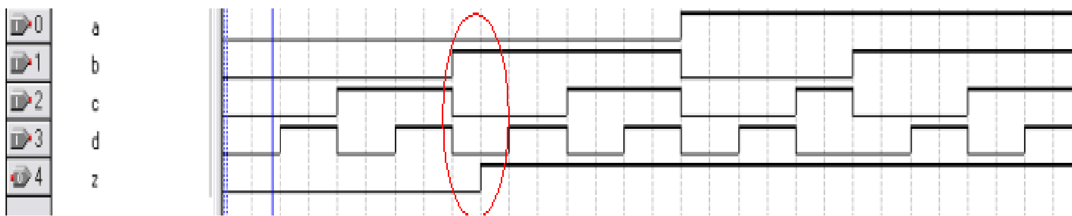

Fig 4.1.10 Premature fire appears at output Z (soft error setting hold LUT 010 (0→ 1))

- **Early return to 0**, hold LUT 011 has changes (1→ 0), so that the TH gate runs reset process not wait until reset LUT delivering "0". It may lead to early return to 0 at wrong condition. The Fig 4.1.10 shows the simulation results.

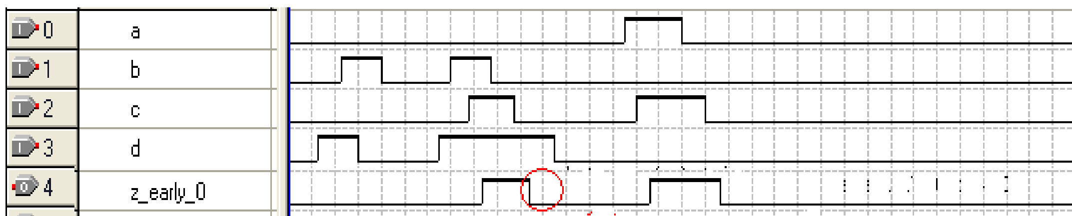

Fig 4.1.11 Early return to 0 appears at output Z (soft error setting hold LUT 011 (1→ 0))

- **No error**, the hold LUT 100,101 change from 0→ 1, these blocks are designed for inputs signal delay arriving. The hold LUT block 100 is used for a process that changes from null to data, the beginning status is reset, and all inputs keep "0". When inputs of set LUT are equal or more than the threshold *m*, the output of set LUT which is also the input of hold LUT changes to 1, so that the hold LUT 100 is active for keeping current output. Similarly,

the hold LUT block 101 is used for a process that changes from data to null, the beginning status is set, and all inputs keep 1. When inputs of reset LUT receives null ("0"), the output of reset LUT which is also inputs of hold LUT changes to 0, so that hold LUT 101 is active for changing current output. Once the SEU (0→ 1) happened at these hold LUT block, short lasting time of these process determine the effect can be ignored.

Table 4.1 Soft error of TH34w2

SEU location	Soft error at Z (worst case)
Set LUT:0000 (0→1)	No error
Set LUT: 0001,0010,0011,0100, 0101,0110,1000 (0→1)	0→1 (premature fire)
Set LUT: others (1→0)	1→0 (No fire)
Reset LUT: 0000 (0→1)	0→1 (No return to 0)
Reset LUT: 0001,0010,0011,0100 0101,0110,1000 (1→0)	1→0 (Early return to 0)
Reset LUT: others (1→0)	1→0 (No fire)
Hold LUT: 000 (0→1)	Oscillating (when abcd = 0000)
Hold LUT: 001 (0→1)	0→1 (No return to 0)
Hold LUT: 010 (0→1)	0→1 (premature fire)
Hold LUT: 011 (1→0)	1→0 (Early return to 0)
Hold LUT: 100,101 (0→1)	No error
Hold LUT: 110 (1→0)	1→0 (No fire)
Hold LUT: 111 (1→0)	Oscillating (when abcd ≥ threshold 3)

In summary, Table 4.1 lists all possible soft errors of TH34w2 associated with different data-flip (SEU) locations. The index of each box in K-map (for example, Fig 4.1.1) is used to represent the location of LUT SRAM cell. Some SEUs, such as Set LUT 0000 and Hold LUT 100 or 101, will not lead to any error. Other SEUs result in four types of soft errors: 1) premature

fire, i.e., the output is erroneously equal to 1 when the inputs do not reach the threshold), 2) no fire, i.e., the output is equal to 0 when inputs reach the threshold, 3) no return to 0, i.e., the output is still 1 even when all inputs are 0, and 4) oscillating. "*Early return to 0*" usually does not lead to malfunction under reasonable delay-timing assumption, therefore being ignored here. The fig 4.1.11 shows the summarized simulation results.

Other threshold gates can be implemented on FPGAs in the same way illustrated in Fig 4.1.1, except that different threshold gates have different content in Set LUT. Therefore, the above statement about four types of soft errors is generally true for all threshold gates.

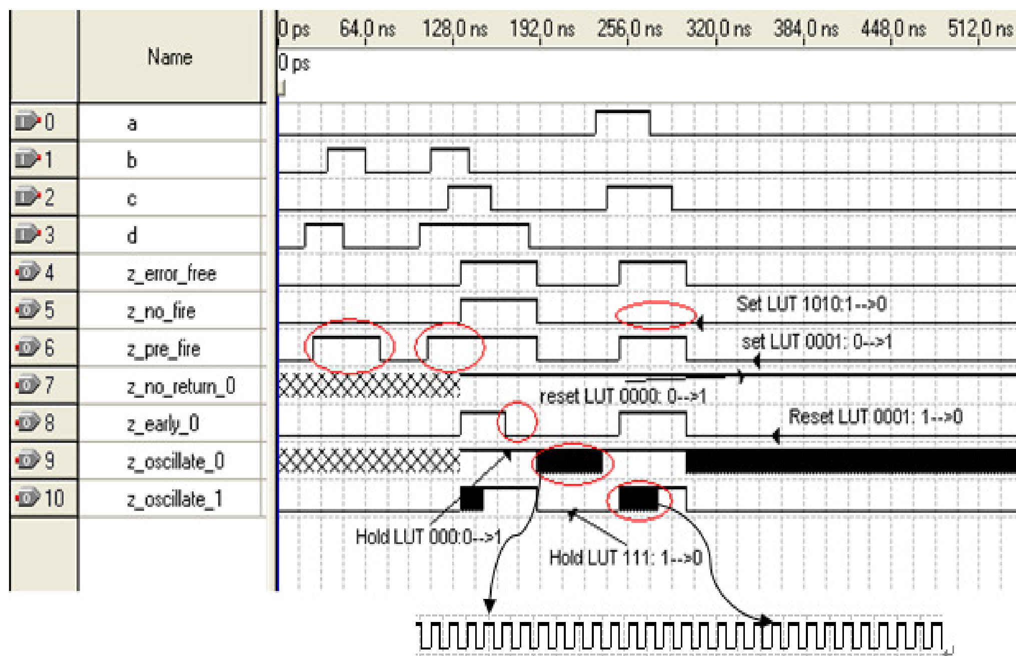

Fig 4.1.12 Summarized simulation results of TH34w2 gate with different soft errors

4.2 Soft Error Propagation and simulation verified in NCL Pipelines

The behavior of any computational block in Fig.4.1.11 has a monotonic property that does not exist in traditional logic. Specifically, during the computation, i.e., transition from NULL to complete DATA, the number of asserted gate-level nodes monotonically increases. On the other

hand, during returning to all NULL from complete DATA, the number of asserted gate-level nodes monotonically decreases to zero. The simulation of normal simulated circuit is shown as Fig4.2.1. It is assumed that an SEU occur only in a computational block in Fig.4.1 because computational blocks consume the major resource of FPGAs. The behavior of an NCL pipeline in the presence of SEU can be delivered by considering: 1) dual-rail encoding, 2) monotonic property, 3) weak conditions, and 4) handshake protocol of pipeline.

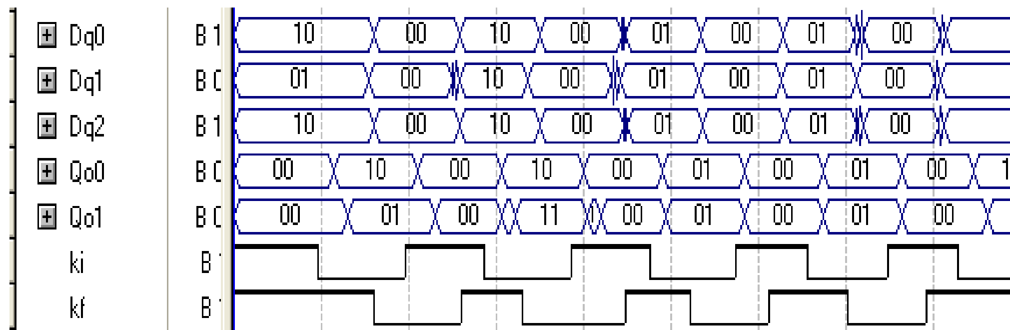

Fig 4.2.1: Normal simulated circuit pipeline.

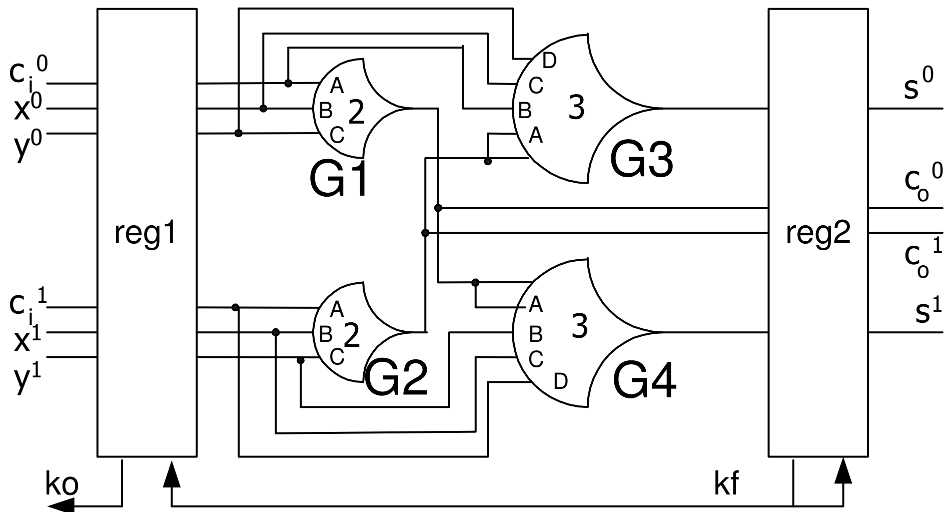

Fig 4.2.2 Simulated NCL full-adder circuit

In order to investigate soft error at RTL level, a simple pipeline in Fig.4.2.2 is simulated and implemented on Altera FPGA Cyclone II EP2C35F672C6N with various soft errors injected into the computational block. Without the loss of generality, soft errors are injected into the different LUT SRAM cells in TH34w2 gate G3 that delivers the output of sum rail0. The effect of soft error on NCL pipeline can be analyzed based on the soft error effect on signal TH gate. The effect of signal TH gate's soft error is analyzed as following:

1) **Premature fire**: some SEUs changing "0" to "1" at output of TH34w2 gate lead to the premature fire. Since the output of TH34w2 Gate G3 delivers the output of sum rail0 (s0), the s0 is the only one infected output by soft error at TH34w2 gate G3. The NCL dual-rail encoding uses two wires to present null, data0, data1, invalid logic value. The NCL pipeline without soft error injection should operate only through the logic value of data1, data0, null. If the premature fire soft error happened at TH34w2 gate G3 and changes its orginal logic value "0" to logic value "1", the output will appear "11" invalid data at system output based on the NCL encoding. For the handshake protocol, the NCL register can detect data full arrives and give completion signal to previous stage NCL register to request null. In premature fire case, the feedback signal of regsiter2 (kf) runs normally, because the "11" invalid data can still satisfy the weak condition of NCL register detection gate. The fig 4.2.3 shows simulation for premature fire case.

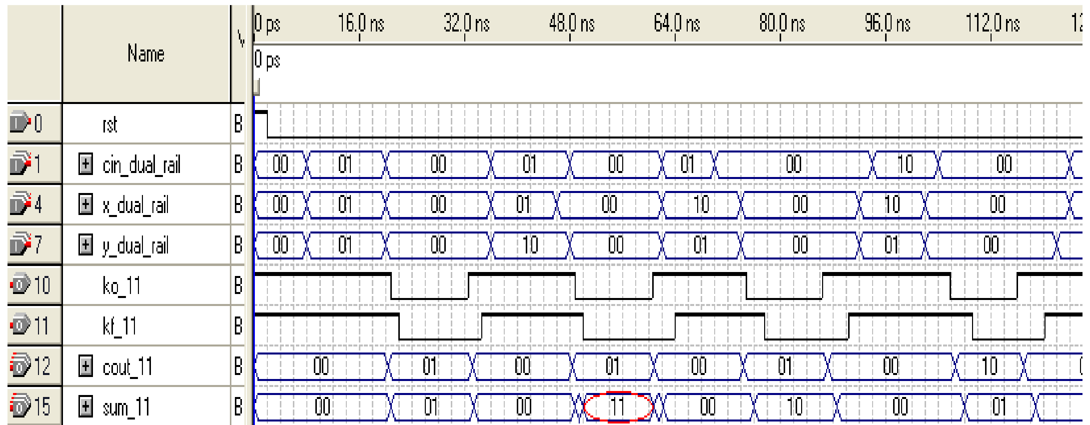

Fig 4.2.3 Invalid "11" appears at the output sum (soft error setting: Set LUT 0110 (0→1) in G3 gate in full adder)

2) **No fire**: some SEUs at LUT (1→0) cause the no fire. The no fire lead to "0" instead of "1", despite the number of inputs satisfies the threshold *m* at TH gate. In the NCL pipeline, no fire may cause "00", since gate G3 which is also called sum rail0 changing from 0→ 1. By delivering "00" to NCL register2, the NCL register2 can't generate feedback of completion data to requesting null to previous stage register. So that the system will stop by requesting data, is shown as Fig4.2.4.

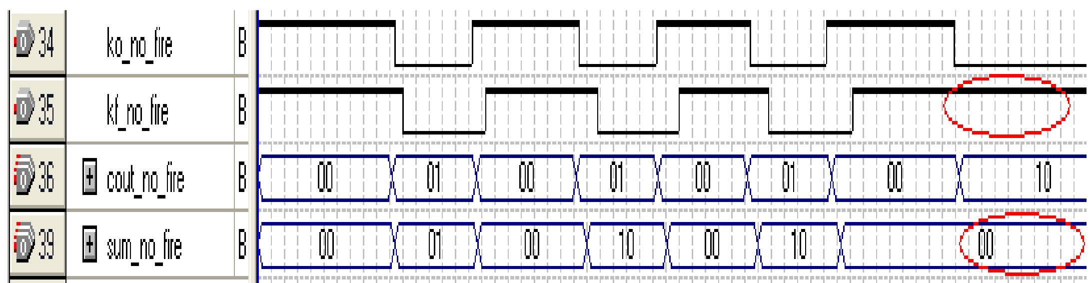

Fig 4.2.4 Deadlock in the pipeline when no fire at sum (soft error setting: Set LUT 1001 (1→0) in G3 gate in full adder)

3) **No return to 0**: some SEUs happened on reset or hold LUT changes SRAM memory from 0→ 1. These may lead to no return to 0. The LUT lose the function of reset makes the TH gate will not delivering "0", through the inputs are ready set up null, thus, the system cannot complete null at output and NCL register cannot generate feedback signal of completion null to requesting data to previous stage register. The effect of no return to 0 on NCL pipeline is stopping the system and keeping requesting null, as shown in Fig4.2.5.

Fig 4.2.5 Deadlock in the pipeline when no return to 0 at sum (soft error setting: hold LUT 001(0->1) in G3 gate in full adder).

4) **Oscillating:** some SEUs happened at hold LUT: 000 (0→1) 111 (1→ 0), the soft error propagates to output and comes back as its input, thus, the output could get value which is opposite with current output from LUT. When the new value comes to output again, it leads to output oscillating. However, in NCL pipeline two different changes have two obvious output, as shown in Fig 4.2.6.

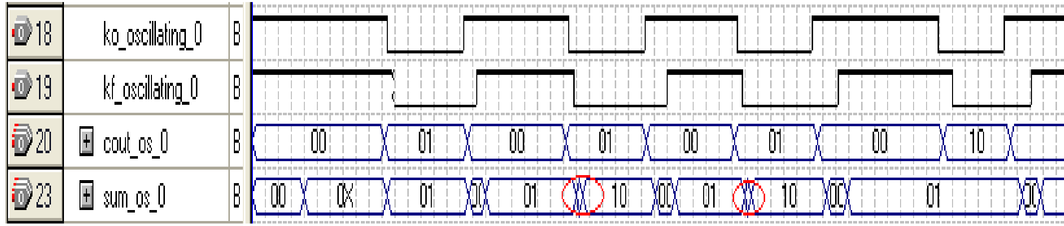

Fig 4.2.6 Invalid “11” appears at sum when G3 ocsillating (soft error setting: Hold LUT 000 (0→1) in G3 gate in full adder)

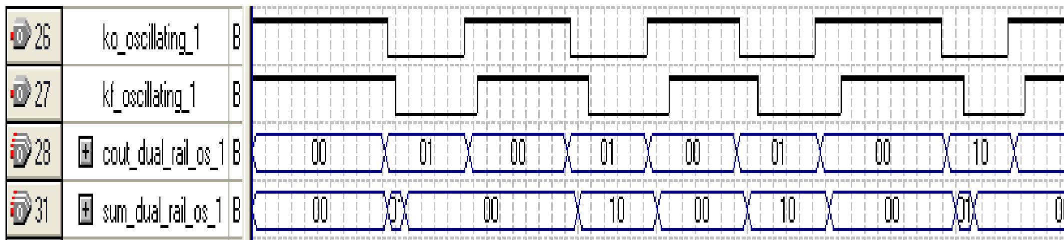

Fig 4.2.7 No error appears at the output when G3 ocsillating (soft error setting: Hold LUT 111(1→0)) in G3 gate in full adder)

Table 4.2 soft errors in NCL pipeline

Soft error at TH gate	Soft error at computational output	Behavior of pipeline
Premature fire	“11”dual-rail code	**Invalid DATA**
No fire	Never to complete DATA	**Deadlock**
No return to 0	Never to complete NULL	**Deadlock**
Oscillating	“11”dual-rail code	**Invalid DATA**

In summary, Table 4.2 lists all possible soft errors at the output of computational block and pipeline behavior originating from SEU in the computational block. *Premature fire* will eventually generate an invalid dual-rail code “11”. *No fire* makes the computation process

everlasting long while *No return to 0* makes the reset process unlimited long. *Oscillating* may lead to an invalid dual-rail code "

CHAPTER V

SOFT ERROR DETECTION SCHEME

This chapter proposed a detection scheme for asynchronous circuits on FPGAs. In the previous chapter, the obvious behavior of soft error injected NCL pipeline on FPGAs are classified by invalid data and deadlock. Unlike the synchronous circuit, the asynchronous circuits do not hold the global clock to control circuit. Furthermore, the delay-insensitivity of asynchronous circuits may cause unbounded delay in both logic elements and interconnects. Therefore, it's hard to separate between normal operation delay and deadlock. The deadlock would never be detected without sacrificing delay-insensitivity.

5.1 Soft error detection scheme.

As explanation of previous chapter, the soft errors in NCL pipeline have been summarized. Table 4.2 gives us the hint to detect SEU in NCL pipelines. The invalid data "11" can be easily detected by ANDing two rails for each bit at output of computational block. Deadlock can be detected by monitoring the request signal Kf between two neighboring registers, shown in Fig 5.1.1. If any of T1, T2, or T3 is approaching to infinite, it is said that deadlock occurs. Since the deadlock would never be detected without sacrificing delay-insensitivity, a worst-case delay should be assumed. We assume that a NULL (or Data) propagation delay is not more than k times of previous Data (or NULL) delay. For example,

k=2 is used in this thesis, that is, T2 < 2T1, T3 < 2T2, otherwise, the occurrence of a soft error is reported. The bigger k, the less false alarm rate.

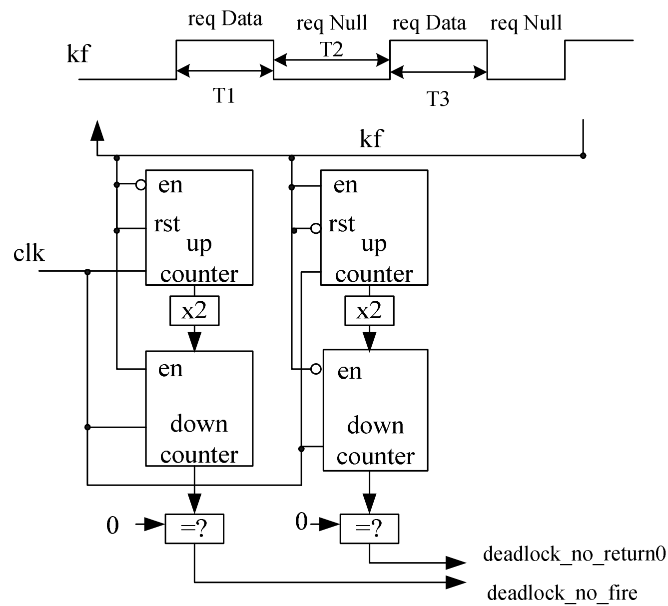

Fig 5.1.1 Scheme of soft error detection

In Fig 5.1.1, the right up-counter and down-counter are used to detect no_return_0 deadlock while the left up and down counters are used to detect no_fire deadlock. In the right side, the up-counter is used to measure the time of Kf request data (T1) and reset to 0 when Kf=0. At the falling edge of Kf, the result of up-counter is doubled and then used as the initial value of the down-counter, and the down-counter starts to reduce 1 every clock cycle. If the output of the down-counter is equal to 0 at one moment, it implies that T2 is equal or more than 2T1, and that a no-return_0 deadlock is detected. To achieve enough accuracy, the frequency of the external clk is relatively much higher than Kf frequency. The size of counters should be enough to avoid overflow. Similarly, the left side up and down counters detect whether T3 is more than twice of T2, i.e., whether a no_fire deadlock occurs. A high-level of output signals from detection circuit will automatically trigger the FPGA programmer to re-program the FPGA. Because of the Kf

gives count up or down signal to both deadlock no return to 0 detection system and deadlock no fire detection system, the all detection systems high-level output will generate sequentially.

5.2 Soft error detection scheme simulations

To demonstrate the effectiveness of the proposed soft error detection scheme, the circuit is attached in Fig 5.2.2 to Fig 5.2.3 to monitor Kf for deadlock detection, and also attach an AND-OR tree circuit to the output of the full adder in Fig 5.2.1 to detect invalid code "11". Furthermore, the up-down counter is all counters are 4-bit. The "11" invalid data detection circuit gives 1 once the invalid data happened. The up-down counter measure the regular data time to determine deadlock case the deadlock_no_return0 and deadlock _no_fire will turn on when the deadlock soft error happened.

Fig 5.2.1 The detect output is 1 when invalid "11" appears at sum (soft error setting: set LUT 0010(0->1))

Fig 5.2.2 The detect_no_fire output is 1 when no fire appears at sum (soft error setting: Set LUT 1001 (1->0) in G3 gate)

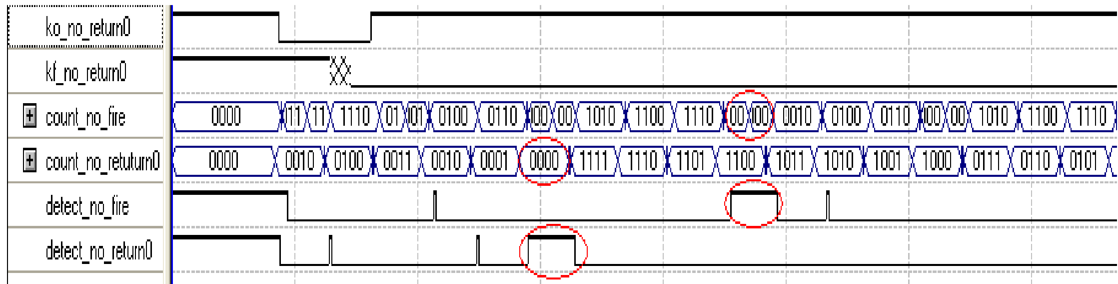

Fig 5.2.3 The detect_ no_return0 output is 1 when no return to 0 appears at sum (soft error setting: hold LUT 001(0->1) in G3 gate)

CHAPTER VI

CONCLUSION

Null Convention Logic Circuit (NCL) can be implemented through either full custom design at transistor level or FPGAs, but the effects of SEU on the two implementation styles are different. In the full-custom implementation, the soft errors are transient and no deadlock happens. The corresponding soft error detection and correction scheme was proposed in [11]. Unfortunately, soft errors in FPGA implementations are permanent. To remove the errors, the FPGA has to be reprogrammed.

This research investigated FPGA-based implementations of asynchronous circuits and their behavior in the presence of SEU in FPGA SRAM cells. The preliminary results and simulations show that dual-rail asynchronous circuits have obvious advantage for SEU detection. The proposed detected circuit can detect all possible soft errors occurs in NCL pipelines which implemented in FPGA.

Future work should focus on the presence of the routing soft error in Asynchronous circuit implemented on FPGAs, and the SEU estimation for Asynchronous circuit implemented on FPGAs. To expatiate the behavior of soft error affected Asynchronous circuit implemented on FPGAs roundly, the multiple or combination of different types of soft error in Asynchronous circuit implemented on FPGAs will be considered too.

REFERENCES

[1] Field-programmable gate array-wikipedia
http://en.wikipedia.org/wiki/Field-programmable_gate_array

[2] Karlsson, J.,Liden, P.,Dahlgren, P., Johansson, R.,Gunneflo, U. “Using heavy-ion Radiation to validate fault-handling mechanisms” IEEE Micro, 14 (1) pp8-23, Feb.1994.

[3] Shivakumar, P. ,Kistler, M. ,Keckler, S.W. ,Burger, D. , Alvisi, L. “Modeling the effect of technology trends on the soft error rate of combinational logic” Proceedings. International Conference, Washigton DC, June 2002.

[4] Soft error- wikipedia http://en.wikipedia.org/wiki/Soft_error

[5] P.Hazucha,C.Svensson “Impact of CMOS Technology Scaling on the Atmospheric Neutron Soft Error Rate,” IEEE Transactions On Nuclear Science, Vol. 47, No. 6, December, 2000.

[6] Paul Graham, Michael Caffrey, “consquences and categories of SRAM FPGA configuration SEUs,” Military and Aerospace Programmable Logic Devices International Conference, Washington DC, 2003.

[7] Weidong Kuang, P. Zhao, J.S.Yuan, and R. DeMara, "Design of asynchronous circuits for high soft error tolerance in deep submicron CMOS circuits," IEEE Trans. on VLSI systems, vol.18, no.3, March 2010, pp.410-422.

[8]Ghazanfar-Hossein Asadi, Mehdi Baradaran Tahoori, “Soft Error Mitigation for SRAM-Based FPGAs” VLSI Test Symposium, IEEE, pp. 207-212, May 2005.

[9] P. Hazucha and C. Svensson, “Impact of CMOS technology scaling on the atmospheric neutron soft error rate,” IEEE Trans. Nucl. Sci., vol. 47, no. 6, pp. 2586–2594, Dec. 2000.

[10] Carl Carmichael, Earl Fuller, Phil Blain, Michael Caffrey, “SEU Mitigation Techniques for Virtex FPGAs in Space Applications” MAPLD, 1999 ppB2 1-B2.11.

[11] I. E. Sutherland, “Micropipelines,” in Communications of the ACM, vol. 32, pp. 720–738,

June 1989
[12]Carl Lebsack, “FPGA technology Overview”, Sep 16, 2003

[13]Weidong Kuang, Yu Bai, “soft error in FPGA-Implemented Asynchronous circuit” IEEE Southern Programmable Logic Conference (SPL), April, 2011.

[14] Virtex™-E 1.8 V. Field Programmable Gate Arrays. DS022-2 (v2.4) July 17, 2002

[15] K. M. Fant and S. A. Brandt, “Null Convention Logic: A complete and consistent logic for asynchronous digital circuit synthesis,” in Int. Conf. Application-Specific Systems, Architecture and Processors, pp. 261-273, 1996.

[16] C. L. Seitz, “system timing,” in Introduction to VLSI Systems, Addison-Wesley, 1980, pp. 218-262[16] G. E. Sobelman, and K. Fant, “CMOS circuit design of threshold gates with hysteresis,” in Proc. Int. Symp. Circuits and Systems, pp. 61-64, 1998

[17] S. C. Smith, R. F. DeMara, J. S. Yuan, M. Hagedorn, and D. Ferguson, "Delay-Insensitive Gate-Level Pipelining," Elsevier's Integration, the VLSI Journal, vol. 30/2, pp. 103-131, October 2001.

[18] L. Antoni, R. Leveugle, and B. Fehér, "Using Run-Time Reconfiguration for Fault Injection in Hardware Prototypes", in Proc. DFT, 2002, pp.245-253.

[19] M. Rebaudengo, M. Sonza Reorda, M. Violante, “Simulation-based analysis of SEU effects on SRAM-based FPGAs” Nuclear Science, IEEE Transactions, Vol: 51 Issue:6 pp: 3354 - 3359 Dec. 2004

[20] P. Liden, P. Dahlgren, R. Johansson, and J. Karlsson, “On Latching Probability of Particle Induced Transients in Combinational Networks,” Proceedings of the 24th Symposium on Fault-Tolerant Computing (FTCS-24), pp. 340-349, 1994.

[21] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, L. Alvisi, “Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic,” Proc. of the International Conference on Dependable Systems and Networks (DSN’02), Washington D.C., June 2002.

[22] H. R. Zarandi, S. G. Miremadi, D. K. Pradhan and J. Mathew, "Soft Error Mitigation in Switch Modules of SRAM-based FPGAs", In the Proceedings of IEEE International Symposium on Circuits and Systems, vol.27, pp. 141-144, 2007.

[23] Ghazanfar Asadi, Mehdi Baradaran Tahoori: Soft error rate estimation and mitigation for SRAM-based FPGAs. FPGA 2005: 149-160.

[24] T. C. May and M. H. Woods. Alpha-particle-induced soft errors in dynamic memories. IEEE Trans. on Electron Devices, 26(1):2–9, 1979.

APPENDIX A

APPENDIX A

VHDL CODE TO SIMULATE NCL CIRCUIT AND DETECTION SCHEME

1. TH34W2 gate

```
library ieee;
   use ieee.std_logic_1164.all;
   entity NCL_TH34 is
         port(a: in std_logic; -- weight 2
                  b: in std_logic;
                  c: in std_logic;
                  d: in std_logic;
                  z: out std_logic );
   end NCL_TH34;
   architecture archth34w2x0 of NCL_TH34 is
   begin
       th34w2x0: process(a, b, c, d)
       begin
           if (a= '0' and b= '0' and c= '0' and d = '0') then
                z <= '0';
           elsif    (a = '1' and b = '1')
                          or (a = '1' and c = '1')
                          or (a = '1' and d = '1')
                          or (b = '1' and c = '1' and d = '1') then
                z <= '1';
           end if; -- else NULL
       end process;
   end archth34w2x0;
```

2. Asynchronous fulladder

```
    Library IEEE;
    Use IEEE.std_logic_1164.all;
    Use work.ncl_signals.all;

    entity asynfulladder is
       port (
                 rst : in std_logic;
                 Qo : out dual_rail_logic_vector(1 to 2);
                 Dq : out dual_rail_logic_vector(1 to 3);
                 ki,kf: out std_logic);
               end asynfulladder;
    architecture behavioral of asynfulladder is
    signal Q1 : dual_rail_logic_vector(1 to 3);
    signal sc : dual_rail_logic_vector(1 to 2);
    signal k1,k2: std_logic;
    signal D : dual_rail_logic_vector(1 to 3);
    component event_countern is
    port ( clk: in std_logic;Di : out dual_rail_logic_vector(1 to 3));
    end component;
    component initreg is
          port ( D : in dual_rail_logic_vector(1 to 3);
                   ki : in std_logic;
```

```
                    rst : in std_logic;
                    Q : out dual_rail_logic_vector(1 to 3);
                    ko : out std_logic);
      end component;
      component finalreg is
           port ( D : in dual_rail_logic_vector(1 to 2);
                    ki : in std_logic;
                    rst : in std_logic;
                    Q : out dual_rail_logic_vector(1 to 2);
                    ko : out std_logic);
      end component;
      component fulladder is
           port ( a : in dual_rail_logic_vector(1 to 3);
                    s : out dual_rail_logic_vector(1 to 2));
      end component;
      begin
      c1:event_countern port map (k1,D);
      Dq<=D;
      reg1 : initreg port map(D,k2,rst,Q1,k1);
      ki<=k1;
      combi1 : fulladder port map(Q1,sc);
      reg2 : finalreg port map(sc,k2,rst,Qo,k2);
      kf<=k2;
      end behavioral;
```

3. Event counter

```
      library ieee;
      use ieee.std_logic_1164.all;
use work.ncl_signals.all;
entity event_countern is
port ( clk: in std_logic;
              Di : out dual_rail_logic_vector(1 to 3));
              end event_countern;
architecture event_counter of event_countern is
signal digit1,digit2 : integer range 0 to 8;
signal digit : integer range 0 to 15;
begin
rising: process(clk)
variable temp1: integer range 0 to 8;
begin
if (clk'event and clk='1') then
temp1:=temp1+1;
if (temp1=8) then temp1:=0;
end if;
end if;
digit1 <= temp1;
end process rising;
falling: process(clk)
variable temp2: integer range 0 to 9;
begin
if (clk'event and clk='0') then
temp2 :=temp2+1;
if (temp2=8) then temp2:=0;
end if;
end if;
digit2 <=temp2;
```

```
end process falling;
digit <=digit1+digit2;
process (digit)
  begin
case digit is
when 0 =>
  Di(1).rail0<='1';
  Di(1).rail1<='0';
  Di(2).rail0<='1';
  Di(2).rail1<='0';
  Di(3).rail0<='1';
  Di(3).rail1<='0';
    when 1 =>
  Di(1).rail0<='0';
  Di(1).rail1<='0';
  Di(2).rail0<='0';
  Di(2).rail1<='0';
  Di(3).rail0<='0';
  Di(3).rail1<='0';
  when 2 =>
  Di(1).rail0<='1';
  Di(1).rail1<='0';
  Di(2).rail0<='1';
  Di(2).rail1<='0';
  Di(3).rail0<='0';
  Di(3).rail1<='1';
  when 3 =>
  Di(1).rail0<='0';
  Di(1).rail1<='0';
  Di(2).rail0<='0';
  Di(2).rail1<='0';
  Di(3).rail0<='0';
  Di(3).rail1<='0';
  when 4=>
  Di(1).rail0<='1';
  Di(1).rail1<='0';
  Di(2).rail0<='0';
  Di(2).rail1<='1';
  Di(3).rail0<='1';
  Di(3).rail1<='0';
    when 5 =>
  Di(1).rail0<='0';
  Di(1).rail1<='0';
  Di(2).rail0<='0';
  Di(2).rail1<='0';
  Di(3).rail0<='0';
  Di(3).rail1<='0';
    when 6 =>
  Di(1).rail0<='1';
  Di(1).rail1<='0';
  Di(2).rail0<='0';
  Di(2).rail1<='1';
  Di(3).rail0<='0';
  Di(3).rail1<='1';
    when 7 =>
```

```
Di(1).rail0<='0';
Di(1).rail1<='0';
Di(2).rail0<='0';
Di(2).rail1<='0';
Di(3).rail0<='0';
Di(3).rail1<='0';
  when 8 =>
Di(1).rail0<='0';
Di(1).rail1<='1';
Di(2).rail0<='1';
Di(2).rail1<='0';
Di(3).rail0<='1';
Di(3).rail1<='0';
when 9 =>
Di(1).rail0<='0';
Di(1).rail1<='0';
Di(2).rail0<='0';
Di(2).rail1<='0';
Di(3).rail0<='0';
Di(3).rail1<='0';
when 10 =>
Di(1).rail0<='0';
Di(1).rail1<='1';
Di(2).rail0<='1';
Di(2).rail1<='0';
Di(3).rail0<='0';
Di(3).rail1<='1';
    when 11 =>
Di(1).rail0<='0';
Di(1).rail1<='0';
Di(2).rail0<='0';
Di(2).rail1<='0';
Di(3).rail0<='0';
Di(3).rail1<='0';
  when 12 =>
Di(1).rail0<='0';
Di(1).rail1<='1';
Di(2).rail0<='0';
Di(2).rail1<='1';
Di(3).rail0<='1';
Di(3).rail1<='0';
  when 13 =>
Di(1).rail0<='0';
Di(1).rail1<='0';
Di(2).rail0<='0';
Di(2).rail1<='0';
Di(3).rail0<='0';
Di(3).rail1<='0';
  when 14 =>
Di(1).rail0<='0';
Di(1).rail1<='1';
Di(2).rail0<='0';
Di(2).rail1<='1';
Di(3).rail0<='0';
Di(3).rail1<='1';
```

```
when others =>
  Di(1).rail0<='0';
  Di(1).rail1<='0';
  Di(2).rail0<='0';
  Di(2).rail1<='0';
  Di(3).rail0<='0';
  Di(3).rail1<='0';
  end case;
end process;
end event_counter;
```

4. Initial register

```
Library IEEE;
Use IEEE.std_logic_1164.all;
Use work.ncl_signals.all;

entity initreg is
     port ( D : in dual_rail_logic_vector(1 to 3);
                ki : in std_logic;
                rst : in std_logic;
                Q : out dual_rail_logic_vector(1 to 3);
                ko : out std_logic);
end initreg;

architecture behavioral of initreg is
signal ao : std_logic_vector(1 to 3);
component ncl_register_D
       generic(width : integer;initial_value: integer); -- 1=DATA1, 0=DATA0, -4=NULL
       port(D: in dual_rail_logic_vector(width-1 downto 0);
              ki: in std_logic;
              rst: in std_logic;
              Q: out dual_rail_logic_vector(width-1 downto 0);
               ko : out    std_logic_vector(width-1 downto 0));
end component;
component th33x0
     port(a: in std_logic;
            b: in std_logic;
            c: in std_logic;
            z: out std_logic);
end component;
begin
regi : ncl_register_D generic map(width=>3,initial_value=>-4)
                                port map(D,ki,rst,Q,ao);
cdi : th33x0 port map(ao(3),ao(1),ao(2),ko);
end behavioral;
```

5. Final register

```
Library IEEE;
Use IEEE.std_logic_1164.all;
Use work.ncl_signals.all;
entity finalreg is
     port ( D : in dual_rail_logic_vector(1 to 2);
                ki : in std_logic;
                rst : in std_logic;
                Q : out dual_rail_logic_vector(1 to 2);
                ko : out std_logic);
```

```
end finalreg;
architecture behavioral of finalreg is
signal ao : std_logic_vector(1 to 2);
component ncl_register_D
    generic(width : integer;initial_value: integer); -- 1=DATA1, 0=DATA0, -4=NULL
    port(D: in dual_rail_logic_vector(width-1 downto 0);
         ki: in std_logic;
         rst: in std_logic;
         Q: out dual_rail_logic_vector(width-1 downto 0);
         ko : out    std_logic_vector(width-1 downto 0));
end component;
component th22x0
   port(a: in std_logic;
        b: in std_logic;
        z: out std_logic);
end component;
begin
regi : ncl_register_D generic map(width=>2,initial_value=>-4)
                          port map(D,ki,rst,Q,ao);
cdi : th22x0 port map(ao(2),ao(1),ko);
end behavioral;
```

6. Signal generator

```
Library IEEE;
Use IEEE.std_logic_1164.all;
Use work.ncl_signals.all;
 entity signal_gen1 is
port (
      clk : in std_logic;
      Di : out dual_rail_logic_vector(1 to 3));
 end signal_gen1;
 architecture behavior of signal_gen1 is
 begin
  process (clk)
 variable count_value: natural:=0 ;
 begin
 if (clk='1')then count_value := (count_value+1) mod 16;
 end if;
  case count_value is
 when 1 =>
 Di(1).rail0<='1';
 Di(1).rail1<='0';
 Di(2).rail0<='1';
 Di(2).rail1<='0';
 Di(3).rail0<='1';
 Di(3).rail1<='0';
 when 2 =>
 Di(1).rail0<='0';
 Di(1).rail1<='0';
 Di(2).rail0<='0';
 Di(2).rail1<='0';
 Di(3).rail0<='0';
 Di(3).rail1<='0';
 when 3 =>
 Di(1).rail0<='1';
 Di(1).rail1<='0';
```

```
Di(2).rail0<='1';
Di(2).rail1<='0';
Di(3).rail0<='0';
Di(3).rail1<='1';
when 4 =>
Di(1).rail0<='0';
Di(1).rail1<='0';
Di(2).rail0<='0';
Di(2).rail1<='0';
Di(3).rail0<='0';
Di(3).rail1<='0';
when 5 =>
Di(1).rail0<='1';
Di(1).rail1<='0';
Di(2).rail0<='0';
Di(2).rail1<='1';
Di(3).rail0<='1';
Di(3).rail1<='0';
when 6 =>
Di(1).rail0<='0';
Di(1).rail1<='0';
Di(2).rail0<='0';
Di(2).rail1<='0';
Di(3).rail0<='0';
Di(3).rail1<='0';
when 7 =>
Di(1).rail0<='1';
Di(1).rail1<='0';
Di(2).rail0<='0';
Di(2).rail1<='1';
Di(3).rail0<='0';
Di(3).rail1<='1';
when 8 =>
Di(1).rail0<='0';
Di(1).rail1<='0';
Di(2).rail0<='0';
Di(2).rail1<='0';
Di(3).rail0<='0';
Di(3).rail1<='0';
when 9 =>
Di(1).rail0<='0';
Di(1).rail1<='1';
Di(2).rail0<='1';
Di(2).rail1<='0';
Di(3).rail0<='1';
Di(3).rail1<='0';
when 10 =>
Di(1).rail0<='0';
Di(1).rail1<='0';
Di(2).rail0<='0';
Di(2).rail1<='0';
Di(3).rail0<='0';
Di(3).rail1<='0';
when 11 =>
Di(1).rail0<='0';
```

```
  Di(1).rail1<='1';
  Di(2).rail0<='1';
  Di(2).rail1<='0';
  Di(3).rail0<='0';
  Di(3).rail1<='1';
    when 12 =>
  Di(1).rail0<='0';
  Di(1).rail1<='0';
  Di(2).rail0<='0';
  Di(2).rail1<='0';
  Di(3).rail0<='0';
  Di(3).rail1<='0';
  when 13 =>
  Di(1).rail0<='0';
  Di(1).rail1<='1';
  Di(2).rail0<='0';
  Di(2).rail1<='1';
  Di(3).rail0<='1';
  Di(3).rail1<='0';
  when 14 =>
  Di(1).rail0<='0';
  Di(1).rail1<='0';
  Di(2).rail0<='0';
  Di(2).rail1<='0';
  Di(3).rail0<='0';
  Di(3).rail1<='0';
    when 15 =>
  Di(1).rail0<='0';
  Di(1).rail1<='1';
  Di(2).rail0<='0';
  Di(2).rail1<='1';
  Di(3).rail0<='0';
  Di(3).rail1<='1';
when others =>
  Di(1).rail0<='0';
  Di(1).rail1<='0';
  Di(2).rail0<='0';
  Di(2).rail1<='0';
  Di(3).rail0<='0';
  Di(3).rail1<='0';
  end case;
end process;
end behavior;
```

7. NCL Full adder

```
Library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.ncl_signals.all;
entity fulladder is
      port ( a : in dual_rail_logic_vector(1 to 3);
                  s : out dual_rail_logic_vector(1 to 2)
                  );
end fulladder;
architecture beh of fulladder is
signal c0,c1: std_logic;
component th23x0 is
```

```
      port( a: in std_logic;
              b: in std_logic;
              c: in std_logic;
              z: out std_logic );
end component;
component th34w2x0 is
          port(a: in std_logic; -- weight 2
                 b: in std_logic;
                 c: in std_logic;
                 d: in std_logic;
                 z: out std_logic );
end component;
begin
g1 : th23x0 port map(a(1).rail0,a(2).rail0,a(3).rail0,c0);
g2 : th23x0 port map(a(1).rail1,a(2).rail1,a(3).rail1,c1);
g3 : th34w2x0 port map(c1,a(1).rail0,a(2).rail0,a(3).rail0,s(2).rail0);
g4 : th34w2x0 port map(c0,a(1).rail1,a(2).rail1,a(3).rail1,s(2).rail1);
s(1).rail0<=c0;s(1).rail1<=c1;
end beh;
```

8. Detection scheme for Invalid data “11”

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.all;
Use work.ncl_signals.all;
entity dect_logic is
port( input:in dual_rail_logic_vector(1 to 2);
          output: out std_logic);
  end dect_logic;
  architecture behavior of dect_logic is
component AND_ent is
port ( A:in std_logic;
             B: in std_logic;
             F1: out std_logic);
end component;
COMPONENT OR_GATE is
port (a:in std_logic;
          b: in std_logic;
          F2: out std_logic);
end component;
signal w1,w2: std_logic;
begin
gate1:AND_ent port map (A=>input(1).rail0,B=>input(1).rail1,F1=>w1);
gate2:AND_ent port map (A=>input(2).rail0,B=>input(2).rail1,F1=>w2);
gate3:OR_GATE port map (a=>w1,b=>w2,F2=>output);
end behavior;
```

9. Detection scheme for Deadlock

```
       library IEEE;
       use IEEE.STD_LOGIC_1164.ALL;
       USE IEEE.STD_LOGIC_UNSIGNED.ALL;
       entity clear_gen is
       port (cectionlk:in std_logic;
di:IN STD_LOGIC;
```

```
Si:out std_logic;
        COUNT_OUT:OUT STD_LOGIC_VECTOR (3 DOWNTO 0));
        END clear_gen;
        architecture behavioral of clear_gen is
        signal Q: STD_LOGIC_VECTOR (3 DOWNTO 0);
                component upcounter_1 is
        port (clk:in std_logic;
        di:IN STD_LOGIC;
        COUNT_OUT:OUT STD_LOGIC_VECTOR (3 DOWNTO 0));
        END component;
                component downcounter_1 is
        port (clk:in std_logic;
        di:IN STD_LOGIC;
        P:OUT STD_LOGIC;
        COUNT:in STD_LOGIC_VECTOR (3 DOWNTO 0);
        COUNT_OUT:OUT STD_LOGIC_VECTOR (3 DOWNTO 0));
        end component;
                begin
                c1: upcounter_1 port map (clk,di,Q);
        c2: downcounter_1 port map (clk,di,SI,Q,count_out);
                end behavioral;
```

A. Down counter

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
entiy downcounter_1 is
port (clk:in std_logic;
        di:IN STD_LOGIC;
        P: OUT STD_LOGIC;
        COUNT:in STD_LOGIC_VECTOR (3 DOWNTO 0);
        COUNT_OUT:OUT STD_LOGIC_VECTOR (3 DOWNTO 0));
        END downcounter_1;
        architecture behavioral of downcounter_1 is
        signal count_int:std_logic_vector(3 downto 0);
        signal B:integer range 0 to 15;
        begin
        process (clk)
        begin
        if clk'event and clk ='1' then
        if di='0'then
        count_int<=count_int-1;
        else
        count_int<=count+count;
        end if;
        end if;
        end process;
        count_out<=count_int;
        B<=conv_integer(count_int);
        process(B)
        begin
        case B is
        when 0=>
        P<='1';
        when others =>
        P<='0';
```

```
        end case;
        end process;
        end behavioral;
```

B. Up counter

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
entity upcounter_1 is
port (clk:in std_logic;
        di:IN STD_LOGIC;
        COUNT_OUT:OUT STD_LOGIC_VECTOR (3 DOWNTO 0));

END upcounter_1;
architecture behavioral of upcounter_1 is
signal count_int:std_logic_vector(3 downto 0):="0001";
begin
process (clk)
begin
if clk'event and clk ='1' then
if di='1'then
count_int<=count_int+1;
else
count_int<="0001";
end if;
end if;
end process;
count_out<=count_int;
end behavioral;
```

APPENDIX B

APPENDIX B

SIGNAL AND THRESHOLD GATE LIBRARY

1. Completion component

```
-- Package used for Completion Component
Library IEEE;
use IEEE.std_logic_1164.all;
package tree_funcs is
function log_u(L: integer; R: integer) return integer; -- ceiling of Log base R of L
function level_number(width, level, base: integer) return integer; -- bits to be combined on level of tree of width
using base input gates
end tree_funcs;
package body tree_funcs is
function log_u(L: integer; R: integer) return integer is
variable temp: integer := 1;
variable level: integer := 0;
begin
        if L = 1 then
                return 0;
        end if;
        while temp < L loop
                temp := temp * R;
                level := level + 1;
        end loop;
        return level;
end;
function level_number(width, level, base: integer) return integer is
variable num: integer := width;
begin
    if level /= 0 then
    for i in 1 to level loop
        if (log_u((num / base) + (num rem base), base) + i) = log_u(width, base) then
            num := (num / base) + (num rem base);
        else
            num := (num / base) + 1;
        end if;
    end loop;
    end if;
    return num;
end;
end tree_funcs;
-- Generic Completion Component --width = 120--
library ieee;
use ieee.std_logic_1164.all;
use work.tree_funcs.all;
```

```
entity comp is
     generic(width : integer);-- := 120);
     port(a: IN std_logic_vector(width-1 downto 0);
              ko: OUT std_logic);
end comp;
architecture arch of comp is
       type completion is array(log_u(width, 4) downto 0, width-1 downto 0) of std_logic;
       signal comp_array: completion;
              component th22x0
                     port(a: in std_logic;
                            b: in std_logic;
                            z: out std_logic);
              end component;
              component th33x0
                     port(a: in std_logic;
                            b: in std_logic;
                            c: in std_logic;
                            z: out std_logic);
              end component;
              component th44x0
                     port(a: in std_logic;
                            b: in std_logic;
                            c: in std_logic;
                            d: in std_logic;
                            z: out std_logic);
                    end component;
begin
       RENAME: for i in 0 to width-1 generate
              comp_array(0, i) <= a(i);
       end generate;
       STRUCTURE: for k in 0 to log_u(width, 4)-1 generate
       begin
            NOT_LAST: if level_number(width, k, 4) > 4 generate
            begin
              PRINCIPLE: for j in 0 to (level_number(width, k, 4) / 4)-1 generate
                     G4: th44x0
                            port map(comp_array(k, j*4), comp_array(k, j*4+1), comp_array(k, j*4+2), comp_array(k,
j*4+3),
                                   comp_array(k+1, j));
              end generate;
              LEFT_OVER_GATE: if log_u((level_number(width, k, 4) / 4) + (level_number(width, k, 4) rem 4), 4) +
k + 1
                                   /= log_u(width, 4) generate
              begin
                     NEED22: if (level_number(width, k, 4) rem 4) = 2 generate
                            G2: th22x0
                                                 port map(comp_array(k, level_number(width, k, 4)-2), comp_array(k,
level_number(width, k, 4)-1),
                                          comp_array(k+1, (level_number(width, k, 4) / 4)));
                     end generate;
                     NEED33: if (level_number(width, k, 4) rem 4) = 3 generate
                                                 G3: th33x0
```

```
                                    port map(comp_array(k, level_number(width, k, 4)-3),
comp_array(k, level_number(width, k, 4)-2),
                          comp_array(k, level_number(width, k, 4)-1), comp_array(k+1, (level_number(width,
k, 4) / 4)));
                         end generate;
                  end generate;
                 LEFT_OVER_SIGNALS: if (log_u((level_number(width, k, 4) / 4) + (level_number(width, k, 4)
rem 4), 4) + k + 1
                                    = log_u(width, 4)) and ((level_number(width, k, 4) rem 4) /= 0)
generate
                 begin
              RENAME_SIGNALS: for h in 0 to (level_number(width, k, 4) rem 4)-1 generate
                  comp_array(k+1, (level_number(width, k, 4) / 4)+h) <= comp_array(k, level_number(width, k,
4)-1-h);
                         end generate;
                 end generate;
      end generate;
      LAST22: if level_number(width, k, 4) = 2 generate
        G2F: th22x0
                         port map(comp_array(k, 0), comp_array(k, 1), ko);
          end generate;
          LAST33: if level_number(width, k, 4) = 3 generate
                 G3F: th33x0
                         port map(comp_array(k, 0), comp_array(k, 1), comp_array(k, 2), ko);
          end generate;
         LAST44: if level_number(width, k, 4) = 4 generate
                 G4F: th44x0
                         port map(comp_array(k, 0), comp_array(k, 1), comp_array(k, 2), comp_array(k, 3),
ko);
          end generate;
    end generate;
end arch;
-- Generic Completion Component width = 64
--library ieee;
--use ieee.std_logic_1164.all;
--use work.tree_funcs.all;
--entity finalcompdet is
   -- generic(width: in integer := 120);
   -- port(a: IN std_logic_vector(width-1 downto 0);
      --       ko: OUT std_logic);
--end finalcompdet;
--architecture arch of finalcompdet is
    --type completion is array(log_u(width, 4) downto 0, width-1 downto 0) of std_logic;
    --signal comp_array: completion;
       -- component th22x0
         --       port(a: in std_logic;
       --              b: in std_logic;
         --            z: out std_logic);
       -- end component;
      --   component th33x0
           --   port(a: in std_logic;
             --        b: in std_logic;
             --        c: in std_logic;
```

```
            --          z: out std_logic);
        -- end component;
        -- component th44x0
            --    port(a: in std_logic;
                --      b: in std_logic;
                --      c: in std_logic;
              --        d: in std_logic;
              --        z: out std_logic);
      --    end component;
--begin
    --RENAME: for i in 0 to width-1 generate
    --    comp_array(0, i) <= a(i);
    --end generate;
    --STRUCTURE: for k in 0 to log_u(width, 4)-1 generate
    --begin
    --     NOT_LAST: if level_number(width, k, 4) > 4 generate
      --   begin
    --     PRINCIPLE: for j in 0 to (level_number(width, k, 4) / 4)-1 generate
    --          G4: th44x0
    --                port map(comp_array(k, j*4), comp_array(k, j*4+1), comp_array(k, j*4+2), comp_array(k,
j*4+3),
    --                          comp_array(k+1, j));
    --     end generate;
    --     LEFT_OVER_GATE: if log_u((level_number(width, k, 4) / 4) + (level_number(width, k, 4) rem 4), 4) +
k + 1
--                              /= log_u(width, 4) generate
    --     begin
    --          NEED22: if (level_number(width, k, 4) rem 4) = 2 generate
    --                G2: th22x0
        --                                  port map(comp_array(k, level_number(width, k, 4)-2), comp_array(k,
level_number(width, k, 4)-1),
    --                              comp_array(k+1, (level_number(width, k, 4) / 4)));
    --          end generate;
    --          NEED33: if (level_number(width, k, 4) rem 4) = 3 generate
          --                                G3: th33x0
          --                                        port map(comp_array(k, level_number(width, k, 4)-3),
comp_array(k, level_number(width, k, 4)-2),
    --                              comp_array(k, level_number(width, k, 4)-1), comp_array(k+1, (level_number(width,
k, 4) / 4)));
       --                           end generate;
        --              end generate;
          --              LEFT_OVER_SIGNALS: if (log_u((level_number(width, k, 4) / 4) + (level_number(width, k,
4) rem 4), 4) + k + 1
          --                                        = log_u(width, 4)) and ((level_number(width, k, 4) rem 4) /= 0)
generate
            --          begin
           --      RENAME_SIGNALS: for h in 0 to (level_number(width, k, 4) rem 4)-1 generate
                --      comp_array(k+1, (level_number(width, k, 4) / 4)+h) <= comp_array(k, level_number(width, k,
4)-1-h);
                     --           end generate;
                --      end generate;
       -- end generate;
      --    LAST22: if level_number(width, k, 4) = 2 generate
```

```
--     G2F: th22x0
            --                    port map(comp_array(k, 0), comp_array(k, 1), ko);
            --   end generate;
            --   LAST33: if level_number(width, k, 4) = 3 generate
            --          G3F: th33x0
                 --               port map(comp_array(k, 0), comp_array(k, 1), comp_array(k, 2), ko);
           --    end generate;
            --   LAST44: if level_number(width, k, 4) = 4 generate
              --        G4F: th44x0
                --                port map(comp_array(k, 0), comp_array(k, 1), comp_array(k, 2), comp_array(k, 3),
ko);
            --   end generate;
    --end generate;
--end arch;
-- 1-bit Dual-Rail Register
use work.ncl_signals.all;
library ieee;
use ieee.std_logic_1164.all;
entity ncl_register_D1 is
    generic(initial_value: integer := -4); -- 1=DATA1, 0=DATA0, -4=NULL
    port(D: in dual_rail_logic;
        ki: in std_logic;
        rst: in std_logic;
        Q: out dual_rail_logic;
        ko: out std_logic);
end ncl_register_D1;
architecture arch of ncl_register_D1 is
signal Qbuf: dual_rail_logic;
component th22nx0
    port (a, b, rst: IN    std_logic;
          z: OUT std_logic);
end component;
component th22dx0
    port (a, b, rst: IN    std_logic;
          z: OUT std_logic);
end component;
component th12bx0
    port (a, b: IN    std_logic;
          zb: OUT std_logic);
end component;
begin
    RstN: if initial_value = -4 generate
        R0: th22nx0
            port map(D.rail0, ki, rst, Qbuf.rail0);

        R1: th22nx0
            port map(D.rail1, ki, rst, Qbuf.rail1);
    end generate;
    Rst1: if initial_value = 1 generate
        R0: th22nx0
            port map(D.rail0, ki, rst, Qbuf.rail0);

        R1: th22dx0
```

```
                port map(D.rail1, ki, rst, Qbuf.rail1);
        end generate;
        Rst0: if initial_value = 0 generate
            R0: th22dx0
                port map(D.rail0, ki, rst, Qbuf.rail0);

            R1: th22nx0
                port map(D.rail1, ki, rst, Qbuf.rail1);
        end generate;
        Q <= Qbuf;
        COMP: th12bx0
            port map(Qbuf.rail0, Qbuf.rail1, ko);
end;
-- Generic Length Dual-Rail Register width=64
use work.ncl_signals.all;
library ieee;
use ieee.std_logic_1164.all;
entity ncl_register_D is
        generic(width: integer;
                initial_value: integer := -4); -- 1=DATA1, 0=DATA0, -4=NULL
        port(D: in dual_rail_logic_vector(width-1 downto 0);
            ki: in std_logic;
            rst: in std_logic;
            Q: out dual_rail_logic_vector(width-1 downto 0);
            ko: out std_logic_vector(width-1 downto 0));
end ncl_register_D;
architecture gen of ncl_register_D is
component ncl_register_D1
        generic(initial_value: integer := -4); -- 1=DATA1, 0=DATA0, -4=NULL
        port(D: in dual_rail_logic;
            ki: in std_logic;
            rst: in std_logic;
            Q: out dual_rail_logic;
            ko: out std_logic);
end component;
begin
        gen_reg: for i in 0 to D'length-1 generate
            REGi: ncl_register_D1
                generic map(initial_value)
                port map(D(i), ki, rst, Q(i), ko(i));
            end generate;
end;
-- 1-bit initreg
use work.ncl_signals.all;
library ieee;
use ieee.std_logic_1164.all;
entity ncl_register_D11 is
        generic(initial_value: integer := -4); -- 1=DATA1, 0=DATA0, -4=NULL
        port(D: in dual_rail_logic;
            ki: in std_logic;
            rst: in std_logic;
            Q: out dual_rail_logic);
end ncl_register_D11;
```

```
architecture arch of ncl_register_D11 is
component th22nx0
	port (a, b, rst: IN	std_logic;
		z: OUT std_logic);
end component;
component th22dx0
	port (a, b, rst: IN	std_logic;
		z: OUT std_logic);
end component;
begin
	RstN: if initial_value = -4 generate
		R0: th22nx0
			port map(D.rail0, ki, rst, Q.rail0);

		R1: th22nx0
			port map(D.rail1, ki, rst, Q.rail1);
	end generate;
	Rst1: if initial_value = 1 generate
		R0: th22nx0
			port map(D.rail0, ki, rst, Q.rail0);
					R1: th22dx0
			port map(D.rail1, ki, rst, Q.rail1);
	end generate;
	Rst0: if initial_value = 0 generate
		R0: th22dx0
			port map(D.rail0, ki, rst, Q.rail0);
					R1: th22nx0
			port map(D.rail1, ki, rst, Q.rail1);
	end generate;
end;
-- Generic Length initial register
use work.ncl_signals.all;
library ieee;
use ieee.std_logic_1164.all;
entity ncl_reg_Dinit is
	generic(width: integer;
		initial_value: integer ); -- 1=DATA1, 0=DATA0, -4=NULL
	port(D: in dual_rail_logic_vector(width-1 downto 0);
		ki: in std_logic;
		rst: in std_logic;
		Q: out dual_rail_logic_vector(width-1 downto 0));
end ncl_reg_Dinit;
architecture gen of ncl_reg_Dinit is
component ncl_register_D11
	generic(initial_value: integer := -4); -- 1=DATA1, 0=DATA0, -4=NULL
	port(D: in dual_rail_logic;
		ki: in std_logic;
		rst: in std_logic;
		Q: out dual_rail_logic);
end component;
begin
	gen_reg: for i in 0 to D'length-1 generate
		REGi: ncl_register_D11
```

```
                    generic map(initial_value)
                    port map(D(i), ki, rst, Q(i));
               end generate;
end;
-- Generic Length Dual-Rail Register width=32
--use work.ncl_signals.all;
--library ieee;
--use ieee.std_logic_1164.all;
--entity ncl_register_D32 is
     -- generic(width: positive := 32 ;
        --          initial_value: integer := -4); -- 1=DATA1, 0=DATA0, -4=NULL
     -- port(D: in dual_rail_logic_vector(width-1 downto 0);
        --     ki: in std_logic;
          --   rst: in std_logic;
          --   Q: out dual_rail_logic_vector(width-1 downto 0);
          --   ko: out std_logic_vector(width-1 downto 0));
--end ncl_register_D32;
--architecture gen of ncl_register_D is
--component ncl_register_D1
       --generic(initial_value: integer := -4); -- 1=DATA1, 0=DATA0, -4=NULL
       --port(D: in dual_rail_logic;
          --   ki: in std_logic;
          --   rst: in std_logic;
          --   Q: out dual_rail_logic;
          --   ko: out std_logic);
--end component;
--begin
       --gen_reg: for i in 0 to D'length-1 generate
       --     REGi: ncl_register_D1
              --     generic map(initial_value)
              --     port map(D(i), ki, rst, Q(i), ko(i));
              --end generate;
--end;
-- Generic Length Dual-Rail Register width=56
--use work.ncl_signals.all;
--library ieee;
--use ieee.std_logic_1164.all;
--entity ncl_register_D56 is
     -- generic(width: positive := 56 ;
          --         initial_value: integer := -4); -- 1=DATA1, 0=DATA0, -4=NULL
       --port(D: in dual_rail_logic_vector(width-1 downto 0);
        --     ki: in std_logic;
          --   rst: in std_logic;
            -- Q: out dual_rail_logic_vector(width-1 downto 0);
          --   ko: out std_logic_vector(width-1 downto 0));
--end ncl_register_D56;
--architecture gen of ncl_register_D is
--component ncl_register_D1
 --      generic(initial_value: integer := -4); -- 1=DATA1, 0=DATA0, -4=NULL
   --    port(D: in dual_rail_logic;
        --     ki: in std_logic;
             --rst: in std_logic;
            -- Q: out dual_rail_logic;
```

```
            -- ko: out std_logic);
--end component;
--begin
        --gen_reg: for i in 0 to D'length-1 generate
                --REGi: ncl_register_D1
                --      generic map(initial_value)
                --      port map(D(i), ki, rst, Q(i), ko(i));
                --end generate;
--end;
```

2. Threshold gate

```
    -------------------------------------
    -- invx0
    -------------------------------------
    library ieee;
    use ieee.std_logic_1164.all;
    entity invx0 is
            port(i: in std_logic;
                    zb: out std_logic);
    end invx0;
    architecture archinvx0 of invx0 is
    begin
            invx0: process(i)
            begin
               if i = '0' then
                    zb <= '1';
               elsif i = '1' then
                    zb <= '0';
               else
                    zb <= not i;
               end if;
            end process;
    end archinvx0;
    ------------------------------------
    -- th12bx0
    -------------------------------------
    library ieee;
    use ieee.std_logic_1164.all;
    entity th12bx0 is
            port(a: in std_logic;
                    b: in std_logic;
                    zb: out std_logic);
    end th12bx0;
    architecture archth12bx0 of th12bx0 is
    begin
          th12bx0: process(a, b)
          begin
               if a = '0' and b = '0' then
                    zb <= '1';
               elsif a = '1' or b = '1' then
                    zb <= '0';
             -- else
                    --zb <= a nor b;
               end if;
```

```
end process;
end archth12bx0;
---------------------------------------------------------
  -- th22dx0
---------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity th22dx0 is
      port(a: in std_logic;
                b: in std_logic;
                rst: in std_logic;
                z: out std_logic );
end th22dx0;
architecture archth22dx0 of th22dx0 is
begin
        th22dx0: process(a, b, rst)
        begin
             if rst = '1' then -- reset
                  z <= '1';
             elsif (a= '1' and b= '1') then
                  z <= '1';
             elsif    (a= '0' and b= '0') then
                  z <= '0';
             end if;
end process;
end archth22dx0;
---------------------------------------------------------
-- th22nx0
---------------------------------------------------------
   library ieee;
   use ieee.std_logic_1164.all;
   entity th22nx0 is
          port(a: in std_logic;
                b: in std_logic;
                rst: in std_logic;
                z: out std_logic );
   end th22nx0;
   architecture archth22nx0 of th22nx0 is
   begin
        th22nx0: process(a, b, rst)
        begin
             if rst = '1' then -- reset
                  z <= '0';
             elsif (a= '1' and b= '1') then
                  z <= '1';
             elsif    (a= '0' and b= '0') then
                  z <= '0';
             end if;
        end process;
   end archth22nx0;
   ---------------------------------------------------------
   -- th22x0
   ---------------------------------------------------------
```

```
library ieee;
use ieee.std_logic_1164.all;
entity th22x0 is
        port(a: in std_logic;
             b: in std_logic;
             z: out std_logic );
end th22x0;
architecture archth22x0 of th22x0 is
begin
th22x0: process(a, b)
      begin
          if (a= '1' and b= '1') then
              z <= '1';
          elsif    (a= '0' and b= '0') then
              z <= '0';
          end if;
      end process;
end archth22x0;
----------------------------------------------------------
-- th23x0
----------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity th23x0 is
port(a: in std_logic;
     b: in std_logic;
     c: in std_logic;
     z: out std_logic );
end th23x0;
architecture archth23x0 of th23x0 is
begin
      th23x0: process(a, b, c)
      begin
          if (a= '0' and b= '0' and c= '0') then
              z <= '0';
          elsif    (a= '1' and b= '1') or (b= '1' and c= '1') or (c= '1' and a= '1') then
              z <= '1';
          end if;
      end process;

  end archth23x0;
----------------------------------------------------------
  -- th23w2x0
----------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity th23w2x0 is
        port(a: in std_logic; -- weight 2
             b: in std_logic;
             c: in std_logic;
             z: out std_logic );
end th23w2x0;
architecture archth23w2x0 of th23w2x0 is
```

```
begin
th23w2x0: process(a, b, c)
begin
        if (a= '0' and b= '0' and c= '0') then
        z <= '0';
        elsif    (a= '1' or (b= '1' and c= '1')) then
        z <= '1';
        end if; -- else NULL
        end process;
        end archth23w2x0;

----------------------------------------------------------
-- th33x0
----------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity th33x0 is
port(a: in std_logic;
      b: in std_logic;
      c: in std_logic;
      z: out std_logic );

end th33x0;
architecture archth33x0 of th33x0 is
begin
th33x0: process(a, b, c)
begin
if (a= '1' and b= '1' and c= '1') then
   z <= '1';
elsif    (a= '0' and b= '0' and c= '0') then
   z <= '0';
end if; -- else NULL
end process;
end archth33x0;
----------------------------------------------------------
--th34w2x0
----------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity th34w2x0 is
port(a: in std_logic; -- weight 2
      b: in std_logic;
      c: in std_logic;
      d: in std_logic;
      z: out std_logic );

end th34w2x0;
architecture archth34w2x0 of th34w2x0 is
begin
        th34w2x0: process(a, b, c, d)
               begin
                    if (a= '0' and b= '0' and c= '0' and d = '0') then
                         z <= '0';
                    elsif    (a = '1' and b = '1')
```

```
                              or (a = '1' and c = '1')
                              or (a = '1' and d = '1')
                              or (b = '1' and c = '1' and d = '1') then
                z <= '1';
            end if; -- else NULL
        end process;
    end archth34w2x0;
    ----------------------------------------------------------
    -- th44x0
    ----------------------------------------------------------
    library ieee;
    use ieee.std_logic_1164.all;
    entity th44x0 is
        port(a: in std_logic;
             b: in std_logic;
             c: in std_logic;
             d: in std_logic;
             z: out std_logic );
    end th44x0;
    architecture archth44x0 of th44x0 is
    begin
        th44x0: process(a, b, c, d)
        begin
            if (a= '1' and b= '1' and c= '1' and d= '1') then
                z <= '1';
            elsif    (a= '0' and b= '0' and c= '0' and d= '0') then
                z <= '0';
            end if; -- else NULL
        end process;
    end archth44x0;
```

3. NCL signal

```
Library IEEE;
use IEEE.std_logic_1164.all;
package ncl_signals is
type dual_rail_logic is
    record
        RAIL1 : std_logic;
        RAIL0 : std_logic;
    end record;
type dual_rail_logic_vector is array (NATURAL range <>) of dual_rail_logic;
end ncl_signals;
```

BIOGRAPHICAL SKETCH

Yu Bai was born in P.R. China in Oct.1984 as the son of Guimin Zhang and Shaogang Bai . He received his bachelor in automation and computer-integrated technology at Ukraine National Aviation University, Kiev, Ukraine in May 2008. He came to USA for pursuing his Master's degree in Electrical Engineering at University of Texas-Pan American in August 2008. His research interests include VLSI design, FPGA, soft error, digital circuit fabrication, and asynchronous circuit.