SOFT-ERROR MITIGATION AT THE ARCHITECTURE-LEVEL USING BERGER CODES FOR ERROR DETECTION

By

Edward J. Ossi

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Electrical Engineering

December 2011

Nashville, Tennessee

Approved:

Professor Bharat L. Bhuva

Professor William H. Robinson

ACKNOWLEGEMENTS

I would like to deeply thank Professor Bharat Bhuva for his work as my academic and research advisor. I am very grateful to him for his support and guidance that has helped me throughout graduate school. His suggestions and insights helped me complete my academic and career objectives.

I would also like to thank Professor William Robinson for his pivotal role in the development of my research. His inputs and advice during the architecture research group meetings were greatly appreciated. I also express my sincere gratitude to Professor Tim Holman, Professor Art Witulski, Professor Lloyd Massengill, Professor Ronald Schrimpf, Professor Robert Reed, Professor Robert Weller, and Professor Daniel Fleetwood for their prescient inputs during group meetings. Their advice during group meetings and instruction greatly advanced my academic career.

I could not have accomplished the work that comprises this thesis without the assistance of Daniel Limbrick. I also have nothing but good feelings about my fellow students in the Radiation Effects and Reliability Group. Their friendship and support made Vanderbilt a wonderful place to learn and work.

Finally, none of this would be possible without the love and encouragement from my parents, Edward and Kathy Ossi, my sister Elena Ossi, my grandmother Ana Maria Ossi, or my loving girlfriend Deborah Walden. I am eternally grateful to them for the support they gave that enabled me to finish this thesis and my graduate school career.

Table of Contents

LIST OF TABLES

Table

LIST OF FIGURES

Figure

CHAPTER I

I. INTRODUCTION

Soft errors are transient faults caused by energetic particles depositing enough charge into a circuit node to invert the logic state of a cell, latch, or gate. The errors are not permanent, but they do create a reliability challenge. Today's advanced integrated circuits (ICs) with reduced operating voltages and higher transistor densities exhibit increased sensitivity to soft errors. As such, soft errors have become a major reliability problem for military, space, and commercial electronic systems [1]. For older technologies, hardening against single events (SE) was achieved through process modifications to reduce the charge collected at a circuit node [2]. As the minimum feature sizes on ICs reached nanometer dimensions, such approaches became less cost-effective, and the circuit designs were used to mitigate the effects of single-events from the circuit [3-5]. Both of these approaches have been adequate to mitigate soft-errors for older technologies. However, in advanced technologies, these approaches have become ineffective against single events due to lower nodal capacitances, lower supply voltages, and close proximity of devices to each other [6]. Lower capacitances and supply voltages have resulted in very low charge requirements to cause an upset, while close proximity of devices can cause multiple devices to collect charge due to a single ion hit. These factors have resulted in a very complex response to single-events for advanced IC designs [7], necessitating higher, architecture-level approaches to manage the soft errors within a system.

Mitigation of soft errors has in CMOS has traditionally focused on a cells instead of combinational logic for two reasons. The first is that error detection and correction schemes, such as parity or error correcting codes (ECC) for memory are well known and their implementations very well understood. Also, caches and other memory structures make up a large part of the die area. Memory cells also feature less masking effects than combinational logic [8]. Strikes on combinational logic have been less investigated at the architecture level because the input and outputs are less easily mapped to code words. Also, a strike that causes a single event transient may not be stored as a single event upset if the transient does not get latched into memory.

This thesis presents several architecture-level error detection and correction strategies that target the Arithmetic Logic Units (ALU) within a microprocessor. The ALU was chosen because it is the heart of a microprocessor and the errors that affect it are unlike those that affect the rest of the microprocessor [9]. The strategies were developed and tested to determine if they could provide desired error coverage without the drastic power and area penalty associated with duplication/triplication of circuits. The proposed approaches seek to eliminate the incorrect data generated by the presence of the soft errors. The performance penalty for each of the techniques is presented in terms of cycles per instruction (CPI), clock frequency, power, and area.

This thesis is organized as follows. Chapter II presents a detailed background on the cause of soft errors and common mitigation strategies found in literature. Chapter III focuses on the hardware implementations on the ALU and Berger circuitry. Chapter IV deals with the two error detection and correction strategies. Chapter V states how a Register-Transfer Level (RTL) model of the circuits were built using VHDL code and

simulated. It also discusses how the designs were synthesized using the FreePDK library for area, speed, and power calculations. Finally, Chapter VI summarizes the results of this thesis.

CHAPTER II.

II. BACKGROUND

1.) SOFT ERRORS

When an energetic ion strikes bulk silicon (or any other semiconductor material), it is of no consequence. The particle will impart its energy to bound electrons, excite the electrons to the conduction band and create a hole, but they will eventually recombine. However, when the particle strikes a p-n junction, it will generate electron-hole pairs; the electrons will be swept to the n-region and the holes to the p-region because of the presence of the electric field at the p-n junction. The path of the ion will also create a field funnel that will extend the depletion region along the ion track and collect additional charge. Fig. 1 (a) shows the ion path and the funnel effect, while Fig. 1 (b) shows the initial current created by the drift process (where charge moves under the influence of an electric field) and the 'tail' created by the funnel and diffusion (where charge moves due to differences in carrier concentrations). At the microarchitecture level, if the charge generated by the ion strike is at a critical node, then it will cause that circuit module to have an incorrect output. This fault may then be latched, resulting in an error caused by a transient voltage pulse at a circuit node.

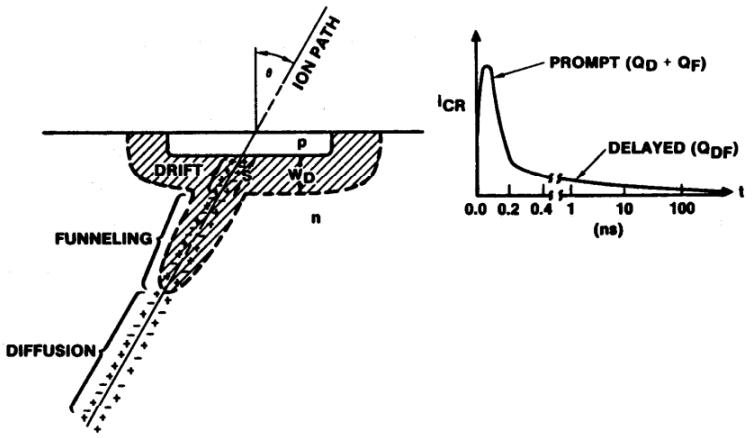

Figure 1. (a) Charge generation and collection at a p-n junction,
(b) resultant current pulse

2.) COMPUTER ABSTRACTION LEVELS

Achieving reliability at the architecture level first requires an understanding of what the architecture level is. Computer systems can be divided into different abstraction levels, from user interfaces down to atomic physics. These abstraction levels from highest to lowest are as follows: Application, Middleware, Operating System, Instruction Set Architecture (ISA), Microarchitecture, Circuits, and Device Physics. Faults present at each level must be handled or, if that prove too costly, propagated to the next higher level. For example, when working at the Microarchitecture level, one must deal with faults that could not be corrected at the Circuits level and unrecoverable errors can ascend to the ISA level and, if need be, levels above it. Fig. 2 shows the order of connectivity between these levels.

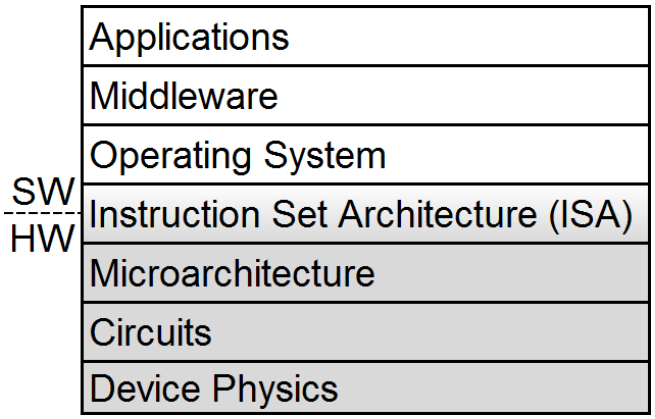

Figure 2. Levels of computer system abstraction [10]

Applications refer to computer programs that a user uses to perform a task. These can range from word processors, to graphics programs to media players. Middleware provides a link between different applications. It allows multiple processes to interact. It was originally developed to link newer applications with older operating systems. The operating system manages hardware and provides services that are used by different applications. The operating system handles input and output and memory allocations. Microarchitecture is defined as the way an Instruction Set Architecture (ISA) is implemented in hardware. The ISA defines the instructions, opcodes, data types, and registers that the processor can use. The ISA does not define the Microarchitecture; different microarchitectures can implement the same ISA. Microarchitecture can be represented as the interconnections of registers and execution units, even logic gates. The circuit level consists of the individual transistors, resistors, capacitors, and inductors that are connected by wires and allow current to flow. In the case of a microprocessor, all of these are fabricated on the surface of a thin wafer of semiconductor material. Device Physics details the way electrons are transferred through the semiconductor

material, the material themselves, and the dimensions and distances. This thesis will focus on the Microarchitecture level, specifically the ALU.

3.) TYPES OF SOFT ERRORS [11]

There are several outcomes when a single event occurs. A single event will always cause a *fault*. A *fault* only becomes an *error* when it has been detected. When a fault causes erroneous output, but is not detected, is termed as *silent data corruption* (SDC). This is the dangerous type of fault, since there is no outward indication that anything is wrong. When an error can be detected but not corrected, it is classified as a *detected unrecoverable error* (DUE). These are further classified as true DUE events and false DUE events. The ability to detect an error without correcting it can lead to false DUE events because they would not have affected program execution but were flagged as an error. The outcomes are shown in Fig. 3.

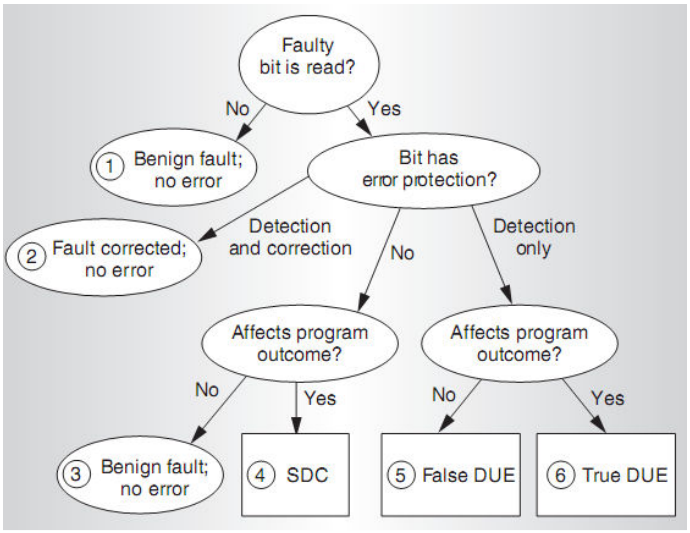

Figure 3. Classification of possible outcomes of single event [12]

4.) ARCHITECTURE LEVEL FAULT METRICS [13]

SDC and DUE rates are expressed in Failure-in-Time (FIT). One FIT is one error in a billion (10^9) device hours. The sum of SDC and DUE FIT rates gives the *soft error rate* (SER) of a chip. *Mean-time-to-failure* (MTTF) is inversely related to FIT and more intuitive. MTTF is the mean time between two faults or errors. If your system has an error every 2 years, then the MTTF would be 2 years.

A chip's FIT is determined by summing the FITs of each component. Each component's FIT is determined by its raw error rate and its *architectural vulnerability factor* (AVF). AVF is the probability that a state change in a component leads to a visible SDC or DUE. AVF varies based on a device's function and execution. An upset in a branch predictor would have an AVF of 0%, while an upset in a program counter would have an AVF of 100%. The raw error rate is the probability that it will experience a bit flip through computer simulation.

The error rate is then derated by the cell's *timing vulnerability factor* (TVF). TVF is the time percentage that a given cell is vulnerable to soft errors. For example, a RAM cell is vulnerable for the entire clock cycle, so its TVF is 100%. Latches are vulnerable for only 50% of the cycle. This is because the latch is holding data for 50% of the cycle, and for the other 50%, data is being driven through it [13]. An example of an architecture-level technique to reduce soft error rate involves flushing the instruction queue after a level-1 (L1) cache miss. After the flushing, the instruction would not be residing in memory while the data is retrieved from memory. This action reduces the TVF by reducing the amount of time the instruction was exposed to potential neutron and alpha strikes [13].

Combinational logic requires special attention. There are three different scenarios that can mask a single event from reaching a forward component that will capture the transient and cause an error. These masking effects are as follows:

- Logical masking occurs when the strike is on a portion of logic that is disconnected from a latch by the other inputs to later gates.
- Electrical masking occurs when the transient is reduced by passing through later gates and eventually its effect is negated. This is caused by circuit delays increasing the rise and fall time of the pulse and gates switching before the full amplitude of the pulse can be reached.
- Latch window masking occurs when the pulse reaches a latch, but at such a time that the latch is steady state. For an edge sensitive latch, the transient must arrive at the latch input during the setup-and-hold time window around the active clock edge, as shown in Fig. 4. Any transients outside this window will be masked (or will not cause an error).

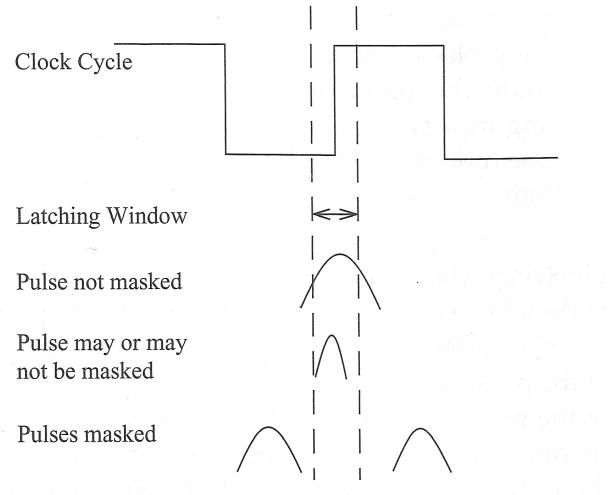

Figure 4. Latching Window Masking [13]

The error rate of combinational logic is therefore derated by a *propagation vulnerability factor* (PVF). PVF is a factor that represents all of the above masking factors.

5.) DUAL / TRIPLE MODULAR REDUNDANCY

Two standard mitigation strategies employed are dual modular redundancy (DMR) and triple modular redundancy (TMR). DMR identifies all instances of error when the two outputs do not match. TMR removes any single point of error, since all portions of the circuit are triplicated and all outputs are majority voted. There is a single point of failure in both strategies, however, that lies in the circuitry that compares the answer being computed. Since this circuit is usually very small compared to the original circuit, the probability of an error in the comparison circuit is very low. Another point of failure may occur if two copies show exactly identical errors. However, the probability of two upsets occurring during the same latch window on the same datapath is extremely low. As a result, these approaches provide excellent error coverage. These two implementations impose extreme area and power penalty, since doubling or tripling the combinational logic will typically cause a 2X or 3X increase in area and power. They do; however, suffer no performance penalty since the redundant copies of the circuit are run in parallel. DMR also has the limitation that it can only detect errors, and requires other circuitry to handle recovery after an error is detected.

CHAPTER III

III. CODE WORDS

1.) ERROR DETECTION FOR EXECUTION UNITS [13]

Error correcting codes can be used to detect and/or correct single or multi-bit upsets. These schemes attach *code bits* to the data word, creating a *code word*. The number of code bits is less than the number of bits in the original data word, creating a savings compared to full replication. This creates a code space of valid code words. When the data need to be read, the entire code word is decoded, and the code bits are read to determine if they belong to the code word space, or set of code words. If they do, then there was no error. Expanded code words can be used to determine the exact bit that was in error.

The number of bits that two code words differ is called the Hamming distance. Given a code word space, the minimum Hamming distance between two valid code words determines the number of error bits that a code scheme can detect. This is referred to as the minimum Hamming distance. For example, X = 00 and Y = 11 differ by two bits, therefore the Hamming distance between X and Y is 2. The minimum Hamming distance of a code word space is given by the following three rules.

- The minimum Hamming distance of a code word space must be ($\alpha + 1$) for it to *detect* α or fewer error bits
- The minimum Hamming distance of a code word space must ($2\beta + 1$) for it to *correct* β or fewer error bits

- The minimum Hamming distance of a code word space must ($\alpha + \beta + 1$), where $\alpha \geq \beta$ for it to *detect* α errors and *correct* β errors.

These types of codes are useful for memory systems; however, they are not applicable for arithmetic and ALU operations since the input data words are operated on by arithmetic and logical functions and result in a new code word. AN codes and Residue codes can be used for arithmetic operations, and parity prediction can be used for both arithmetic and logic operations. AN codes are formed by multiplying each data word N by a constant A. Residue codes use the modulus operation which determines the remainder of a division. The underlying principle is that (X + Y) mod M is equal to X mod M + Y Mod M. This is also true for subtraction. Parity prediction circuits compute the parity of the result and compare it to the parity of the source data words, the result, and the internal carries of the arithmetic operation. Each of these types of codes still has vulnerabilities. Parity schemes are vulnerable to errors which flip an even number of bits and residue codes to errors that result in the same modulo code word.

2.) BERGER CODES

In this thesis, the error detecting code used is the Berger code. Berger was chosen because it has been shown to be the least redundant systematic code for detecting single and multi-bit unidirectional errors [14]. Berger code also covers both logic and arithmetic operations, reducing the amount of code words needed to protect the ALU and reducing cost. The key to this type of correction scheme is that the data and code words are sent through asymmetric channels. The code words are then subjected to separate operations to compare to a code word on the output. This thesis will show that a

comparison of the input data words to the output based on Berger prediction is a cost-effective error correction strategy.

The original Berger code paper was published in 1961. It proposed two encoding schemes to compute the check symbol, B0 and B1. The check symbol B0 is the number of 0's in the data word represented as a binary number. The check symbol B1 represents the number of 1's in the data word. The check symbol length *k* is given by $k = \log_2(n+1)$, where *n* is the number of bits in the original data word [15]. For example, a 32-bit data word would require a 6-bit Berger check symbol. By comparing the check symbols, B0 or B1, of two data words, all unidirectional errors can be detected. Unidirectional errors only flip 0's into 1's or 1's into 0's. If both a 1 and 0 are flipped in the data word, the error will not be detected. This problem is inherent when using the Berger code as the primary error detection system.

Berger code words for ALUs are determined as follows. This example will use addition of two n-bit numbers, *X* and *Y*. $X = (x_n, x_{n-1}, \ldots, x_1, x_0)$ and $Y = (y_n, y_{n-1}, \ldots, y_1, y_0)$ are added to produce the sum $S = (s_n, s_{n-1}, \ldots, s_1, s_0)$ with internal carries $C = (c_n, c_{n-1}, \ldots, c_1, c_0)$, where x_i, y_i, s_i, c_i are $\in \{0, 1\}$. The addition of the i^{th} bit of the two operands can be described as:

$$x_i + y_i + c_{i-1} = 2c_i + s_i = (s_i + c_i) + c_i \quad (1)$$

Let N(X) denote the number of 1's in the binary representation of X. Then, N(xi) = xi and we have the following Lemma:

$$N(X) + N(Y) + c_{in} = N(S) + c_{out} + N(C) \quad (2)$$

Where c_{in} = carry input and c_{out} = carry output. The check symbol in our ALU is in the B0 encoding, so for an n-bit number X, the check symbol in B0 encoding is Xc = n –

N(X) or N(X) = n - Xc. Inserting this into the previous Lemma with some rearranging, we have

$$Sc = Xc + Yc - c_{in} + c_{out} - Cc \quad (3)$$

A similar analysis can be done for subtraction, logical, rotate and shift, and array multiplication operations. The operations that were implemented in our ALU are presented below in Table 1.

Table 1. Berger check algorithms

Operation	**Berger Check Algorithm**
ADD	$Sc = Xc + Yc - Cc - c_{in} + c_{out}$
SUB	$Sc = Xc - Yc - Cc - NOT(c_{in}) + c_{out} + n$
AND	$Sc = Xc + Yc - (X \text{ or } Y)c$
OR	$Sc = Xc + Yc - (X \text{ and } Y)c$
XOR	$Sc = Xc + Yc - 2(X \text{ and } Y)c + n$
ROTATE	$Sc = Xc$
LOGIC SHIFT	$Sc = Xc - c_{in} + c_{out}$
ARITHMETIC SHIFT RIGHT	$Sc = Xc - Xn + c_{out}$
ARITHMETIC SHIFT LEFT	$Sc = Xc + c_{out}$
IDENTITY	$Sc = Xc$

To further explain how Berger Code calculates the checksum, the sum of two actual numbers X and Y, will be shown. Let X=1001 and Y= 1010 with c_{in}=0. The sum is S=0011 with c_{out} = 1. The internal carry bits are C=1000. The number of zeroes for each is Sc=2, Xc=2, Yc=2, and Cc=3. The BCP formula for addition is $Sc=Xc + Yc - Cc - c_{in} + c_{out}$, or $Sc = 2 + 2 - 3 + 1$. This is equal to 2, which is also equal to the number of zeroes in the result of the addition operation [15].

CHAPTER IV

IV. CIRCUIT HARDWARE IMPLEMENTATIONS

1.) ALU

The fault tolerant requirement of the system required that the ALU's arithmetic and logical operations be partitioned, such that a single error in either the logic or arithmetic portion of the circuit cannot affect the other. This also allows for any type of adder sub-circuit to be chosen. For our circuit, a carry look-ahead adder was implemented by recursively expanding the carry term to each stage. Recursive expansion allows the *carry* expression for each individual stage to be implemented in a two-level *AND-OR* expression. This reduces the *carry* signal propagation delay (the limiting factor in a standard ripple carry adder) to produce a higher-performance addition circuit [15].

2.) THE BERGER CHECK PREDICTION (BCP) CALCULATOR [16]

The predictive schemes described in Chapter II protect both the logical and arithmetic data paths of the ALU. The ALU is controlled by four external control signals, A0, A1, and A2 and a carry_in signal. These signals, along with c_{in} and c_{out} from the ALU are translated to the BCP control signals through a programmable logic array (PLA). It decodes these inputs and performs the functions described in Table 2. The hardware implementation of the BCP is shown in Fig. 5.

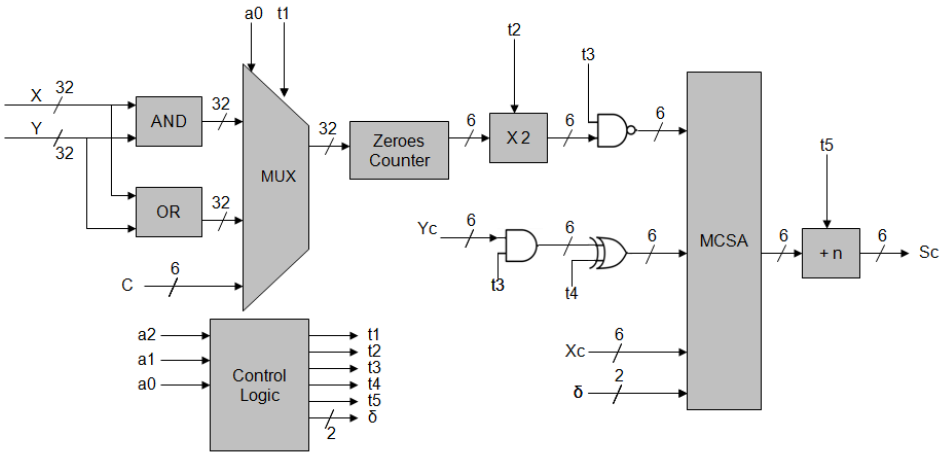

Figure 5. BCP Circuit

Table 2. Truth table for implementation of the Control Logic

PLA Inputs			Function	PLA Outputs					
A0	A1	A2		t1	t2	t3	t4	t5	δ
0	0	0	$S = X + Y + c_{in}$	0	0	1	0	0	$c_{out} - c_{in} + 1$
0	0	1	$S = X - Y - c_{in}$	0	0	1	1	1	$c_{out} - c_{in} + 2$
0	1	0	Rotate Left	0	0	0	0	0	0
0	1	1	Rotate Right	0	0	0	0	0	0
0	0	0	S = X AND Y	1	0	1	0	0	1
0	0	1	S = X OR Y	0	0	1	0	0	1
0	1	0	S = X XOR Y	0	1	1	0	1	1
0	1	1	S = X	0	0	0	0	0	0

The MUX is controlled by signals A0 and t1. When A0 = 0 and t1 = 0, the internal carries of the ALU are selected and routed the BCP circuit, when A0 = 0 and t1 = 1, X AND Y is routed, and when A0 = 1 and t1 = 1, X OR Y is routed to the BCP. The data path length for the BCP circuit for the 32-bit ALU is 6 bits.

To verify this circuit is correct, the arithmetic function "S = X + Y + c_{in}" will be used. The BCP check symbol for this is calculated as "Sc = Xc + Yc – Cc – c_{in} + c_{out}". The PLA sets t1-t2 and t4-5 as '0', t3 as '1', and δ as $c_{out} – c_{in} + 1$. The selection A0 = 1 and t1 = 0 feeds the internal carries to the zeros counter, producing Cc. The x2 operation is not needed, so t2 is '0'. The signal t3 is '1', inverting Cc as it passes through the NAND gate and allowing Yc to pass through the AND gate. The signal t4 is '0', allowing Yc to pass through the XOR gate. These are summed in the Modified Carry-Select Adder (MCSA) and finally signal t5 does not add n. δ is given as $c_{out} – c_{in} + 1$. This computes the symbol based on 2's complement subtraction, Sc = Xc + Yc + (Ccbar + 1) + $c_{out} – c_{in}$. The flow for logical operations is similar, with the MUX selected with X AND Y or X OR Y for the Cc input datapath.

3.) ZEROES COUNTER [14]

Three implementations of the zeroes counter were constructed in order to determine the most efficient way to implement these inputs to the BCP. A behavioral description was designed for the synthesizer to implement as efficiently as possible along with a simple adder tree. Finally a survey of literature found a suitable third implementation to test.

The first implementation was described in VHDL as a 32-stage 1-bit adder tree. Again, the input word was inverted to sum each 0 as a 1. Each bit of the input word is summed with every other bit. This seems grossly inefficient, but as a behavioral descriptionit takes advantage of the Carry-Save Addition transformation capability of the RTL compiler. This is discussed in a later section.

The second implementation is an adder tree. It consists of 5 stages, where there are sixteen 2-bit adders, eight 3-bit adders, four 4-bit adders, two 5-bit adders, and one 6-bit adder. This is designed to reduce the delay of the 32 1-bit add operations. Again, this design is also meant to take advantage of the CSA transformation optimization. This transformation is responsible for the advanced layout techniques that would be present in a modern fabrication of an ALU.

There are many different implementations described in literature proposing different 1's and 0's counters for the B0 and B1 encoding schemes. Some of these schemes include a symbol generator consisting of half-adder cells, half-adder and full-adder cells, and as a set of m-out-on-n codes. The most efficient of the designs found is described as follows. The building block of this scheme is a 4-bit 1's counter. It is used by inverting the input data word, thus providing a representation of the number of 0's by using a 1's counter. The four input 1's counter outputs a 3-bit representation of the number of 1's. The input data word is then partitioned into 4-bit slices and the outputs of these slices are fed to an adder tree [14]. This was the third zeros counter constructed.

CHAPTER V

V. FAULT SECURE ALU IMPLEMENTATIONS

At the architecture-level, the principle difficulty is in developing a technique to eliminate the soft error once it is detected. The approaches to error correction mostly involve overwrite or recalculation. For overwrite, corrupted bit(s) are identified, and correct values are overwritten over the incorrect bits. This can be accomplished by using complicated check codes that require large computation overhead and are costly, but can automatically correct the data. Another approach for overwrite is to use three copies of the hardware in parallel and vote on incorrect data. Approaches for recalculation essentially recalculate the incorrect data assuming that recalculated data will be correct. Recalculation approaches require very little overhead and can be as robust as overwrite approaches. For this thesis, the recalculation approach with two different implementations is investigated. These two implementations differ in their basic approach to error correction. The first approach only intervenes when an error is detected and repeats the instruction; the second approach always repeats the instruction without any additional penalty to the operating frequency. The first approach is expected to be better suited to an environment where the number of errors expected is very low, while the second approach will be better for an environment where the number of soft errors expected is high. Details of both the implementations are given below.

1.) REPEAT-AS-NEEDED IMPLEMENTATION

This implementation repeats the instruction during which the soft error occurred. The block diagram is shown in Fig. 6. The Berger Check Calculator block is responsible for raising a flag whenever a soft error is detected. Upon detection of the soft error, the clock to the entire system is suspended for one clock cycle. This results in all registers holding their values for an additional clock cycle. This effectively repeats the previous instruction until the Single Event Transient (SET) pulse has dissipated. If the SET pulse is longer than one clock cycle, then the clock suspension must last as long as the SET pulse affects the output data.

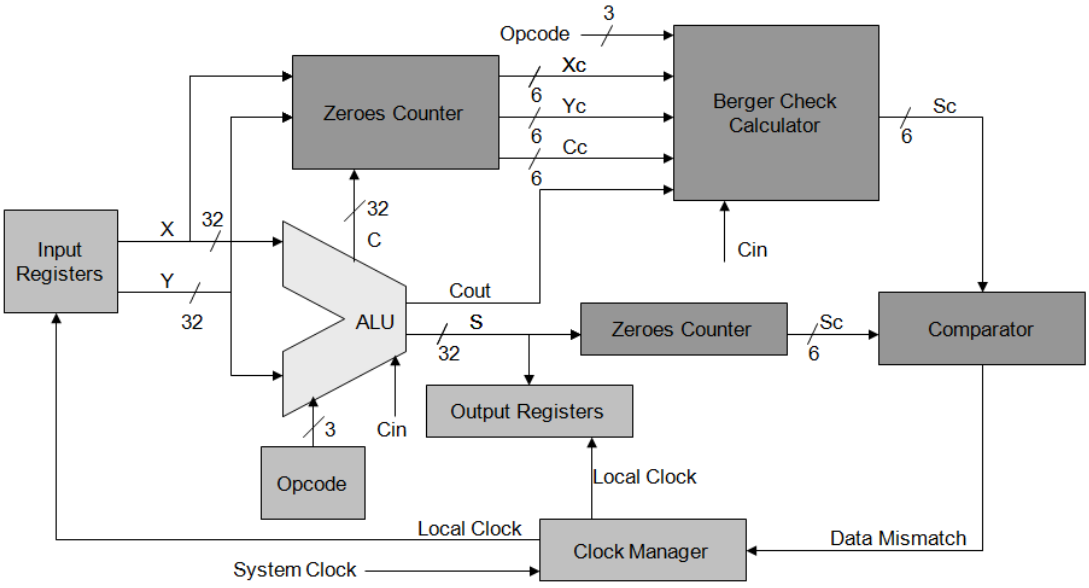

Figure 6. Block diagram of Repeat-As-Needed Implementation

The Clock Manager block receives the system clock and provides local clock signals to the sub-circuit being hardened. For the present case, the sub-circuit is the ALU and all the associated input and output registers. The Clock Manager holds the clock only

when a soft error is detected, resulting in a penalty on performance that is proportional to the number of soft errors.

2.) REPEAT-ALWAYS IMPLEMENTATION

This implementation exploits temporal redundancy by running every ALU instruction twice in one clock cycle as shown in Fig. 7. The instruction executes once on the positive edge and once on the negative edge of the clock. In the first half clock cycle, the ALU performs the operation concurrently with the Berger Check Calculator and determines whether the operation executed as intended (absent an error). In the second half of the clock cycle, the ALU repeats the same operation and stores the result. If an error occurs in the first half of the clock cycle, the circuit will latch the result from the second half of the clock cycle. In all other cases, the ALU stores the result from the first half of the clock cycle. The basic assumption is that the soft error causing transients are shorter than half the clock period. Recent papers have shown that the number of short single-event pulses is orders of magnitude higher than that for longer ones for radiation exposure [17, 18]. For such cases, most of the short errors will be detected and corrected. However, errors due to SET pulse longer than half a clock cycle may still get through the system.

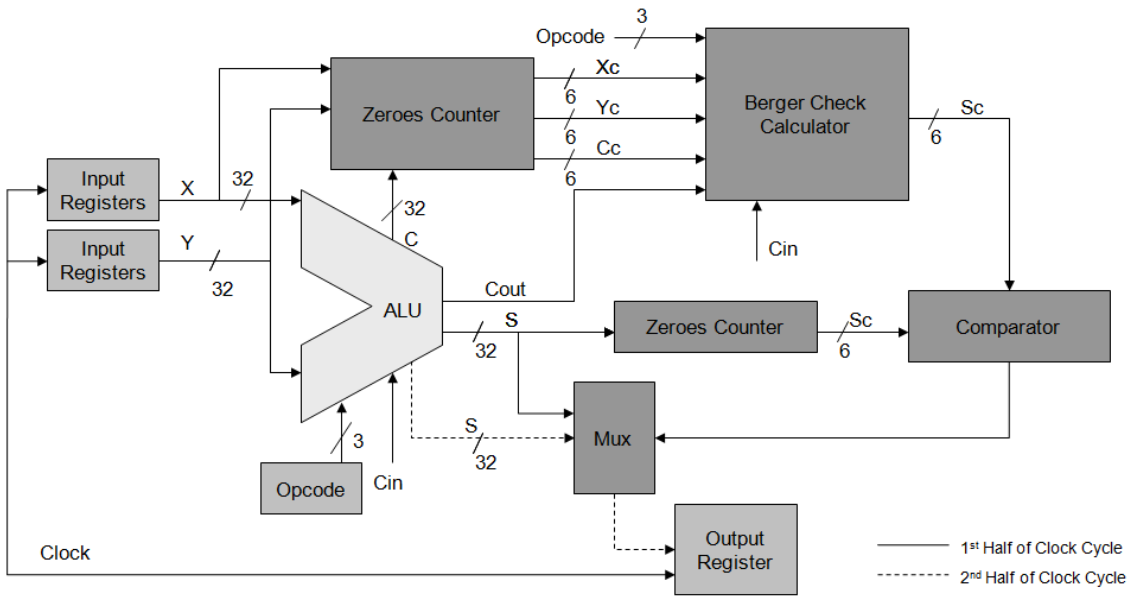

Figure 7. Block diagram of Always-Intervention Implementation

The overriding circuitry was designed with a multiplexer and a series of registers. In order to run an instruction twice, two sets of input registers and output registers were used; one loads on the positive edge and the other loads on negative edge. The multiplexer determines which output register to use as the ALU output. The detection of the error is done using the same comparator design as the Repeat-As-Needed implementation.

CHAPTER VI

VI. SIMULATION METHODOLOGY

1.) RTL MODEL

A VHDL register transfer level (RTL) model was created for both designs and implemented in an Altera DE2 Development and Education Board, which uses a Cyclone II Field Programmable Gate Array (FPGA). The two BCP ALU designs were synthesized using Quartus II version 9.1, and the device targeted is EP2C35F672C6 from the Cyclone II family [19].

2.) FAULT INJECTION METHODOLOGY

The fault injection methodology was based on [20]. It involves inserting faults into particular nodes in the system, and then monitoring the output to determine its behavior in the presence of a fault. This methodology was chosen since it is well suited for early in the design process, when an RTL model of system is all that has been designed. This also allows for faults to be injected at sensitive nodes of the design concurrent with the execution of whatever program or test bench is running. Our implementation differs from [20] in that we have chosen 32 specific nodes to inject faults into, and our fault injection block varies temporally, but not spatially within the design. The faults are injected into the registers that store data or the in-between functional units (FUs). This does not allow for faults to be injected into individual nodes with the combinational logic.

The fault injection mask consists of a 32-bit shift register. It is initialized with the Least Significant Bit (LSB) as 1, with all others 0. After each injection, the value 1 rotates to the left. Each bit in the registers corresponds to a fault injection location. Faults were injected one at a time into each node for every possible instruction. The nodes were chosen based on their transparency to the output to attempt to eliminate logical masking. Logic masking is the failure of an SET to cause an upset because it does not have a logical path the output. This seeks to provide a worst-case fault injection profile where the only limitation is a fault not occurring during a sensitive window and being latched. This latch window masking is described in Chapter II.

The fault injection locations are given in Table 3 in terms of the VHDL model. For instance, *Inject (31 downto 0)* is the fault injection mask shift register. The *Location in Circuit* description is broken down *Module: Signal Name (Bit)*. For instance, *ALU: inA(0)* is the LSB of data words labeled 'A' being fed into the ALU. The full VHDL code is included in Appendix A.

Table 3. Fault injection locations

Inject (31 downto 0)	Location in Circuit	Inject (31 downto 0)	Location in Circuit
0	ALU: inA(0)	16	BCP: bcp_c(5)
1	ALU: inA(31)	17	ALU: result(17)
2	ALU: inB(14)	18	ADDR: carry_gen(18)
3	ALU: inB(6)	19	ADDR: h_sum(19)
4	BCP: inA(28)	20	BCP: opcode(2)
5	BCP: inA(0)	21	PLA: t1
6	BCP: inB(21)	22	PLA: t2
7	BCP: inB(8)	23	PLA: t3
8	ALU: opcode(2)	24	PLA: t4
9	ADDR: carry_in	25	PLA: t5
10	BCP: carry(14)	26	PLA: d(0)
11	ALU: carryout	27	ALU: result(27)
12	ALU: result(12)	28	BCP: mux_sig(28)
13	BCP: carryin	29	ALU: result(29)
14	ALU: alu_c(0)	30	ALU: result(30)
15	BCP: bcp_c(2)	31	ALU: inB(0)

The terminology will treat an error as a flipped bit in the Device-Under-Test (DUT). A detected error is one that is caught by the Berger check circuit. A fault is an error that has escaped detection and affected the system output. To simulate fault injection in our RTL model, XOR gates were used with the targeted node and fault injection signature as inputs using the ModelSim gate level simulation tool [19].

The duration of each fault injected was varied using the IEEE.MATH_REAL Library UNIFORM function to create a pseudorandom pulse width that varied between 0 and 125% of the clock period. The UNIFORM function was also used to randomly vary the position of the fault pulse with respect to the clock edge. A second ALU was run simultaneously to determine the functionally correct execution. At the end of each trial, the results of each instruction were recorded and compared to the original results data.

This data was used to quantify the inherent vulnerability of the ALU by determining a percentage of faults injected to errors recorded. The same procedure was performed on the BCP ALU designs. A similar fault injection method can be found in [20].

CHAPTER VII.

VII. RESULTS

1.) COMPARISON METHODOLOGY

The effectiveness of a Repeat-As-Needed BCP ALU fault-tolerant processor was compared against DMR and TMR implementations based on the impact of each design on data arrival time, which sets the maximum clock frequency, and synthesized area. The effectiveness of the two BCP implementations was then compared based on the percentage of detected errors and undetected errors. The data arrival time and area required were determined using the Cadence RTL compiler, which synthesized the VHDL source code to standard cells from the OSU/NCSU FreePDK 45nm logic cell library [21]. The detailed functional simulations were done using the Modelsim gate level simulation tool on the design synthesized for a Cyclone II EP2C35F672C6 FPGA Board.

2.) ZEROES COUNTER – SPEED AND AREA

Three implementations of the zeroes counter were constructed in order to determine the most efficient way to implement these inputs to the BCP. The first implementation partitions the input data word into 4-bit slices, and the outputs of these slices are fed to an adder tree. The second was a behavioral description written in VHDL as a 32-stage 1-bit adder tree. The third implementation is an adder tree. It consists of 5 stages, where there are sixteen 2-bit adders, eight 3-bit adders, four 4-bit adders, two 5-bit adders, and one 6-bit adder.

Table 4. Zeroes counter synthesis results

		Speed Optimized	Area Optimized
Literature [14]	**cell area (μm^2)**	1009	853
	data arrival time (ps)	679	985
Behavioral	**cell area (μm^2)**	825	604
	data arrival time (ps)	724	898
Adder Tree	**cell area (μm^2)**	863	618
	data arrival time (ps)	858	952

The results for the zeroes counter confirmed what was in the literature when the synthesis was optimized for speed. The data arrival time of 679 ps was 6% and 26% faster than the behavioral or adder tree implementations, respectively. However, when optimized for area, the behavioral circuit had a smaller footprint and was 5% faster than that which was described in literature. This is most likely due to the more generic VHDL code, which allowed increased optimization during synthesis. Since the 4-bit slices to adder tree method was indeed faster, and only 5% slower when optimized for area, it was used as the zeroes counter when comparing Repeat-As-Needed and Repeat-Always configurations.

3.) FAULT-TOLERANT ALU - SPEED AND AREA

The synthesis results for the unhardened ALU, and Dual Modular Redundancy (DMR), Triple Modular Redundancy (TMR), Repeat-As-Needed, and Repeat-Always schemes are shown in Tables 5 and 6. The unhardened ALU featured input and output registers, but no detection or correction measures. The size of just the ALU itself was found to be 1760 μm^2. The DMR implementation used a simple comparator to determine if an error occurred, TMR used three ALUs and a voting logic block to provide error

detection and correction. Each implementation was synthesized twice, once optimized for speed and again for area.

The results when optimized for area are presented in Table 5. It is immediately noticeable that adding redundancy results in a near doubling of the area. The Repeat-As-Needed implementation does have the smallest increase; however, the increases in data-arrival time were quite significant at 22% over DMR. The area-delay product is used as the metric for comparing the different area and speed optimized implementations. Fig. 8 presents the Area-Delay Product of each implementation, normalized to the unhardened ALU. When taking area and speed into account, the two Berger implementations are outperformed by DMR and TMR. The Repeat-Always is significantly worse.

Table 5. Optimized For Area

	cell area (μm^2)	**data arrival time (ps)**
Unhardened ALU	2592	3673
DMR	4718	4213
TMR	6404	3893
Repeat As Needed	4705	5153
Repeat Always	5985	5634

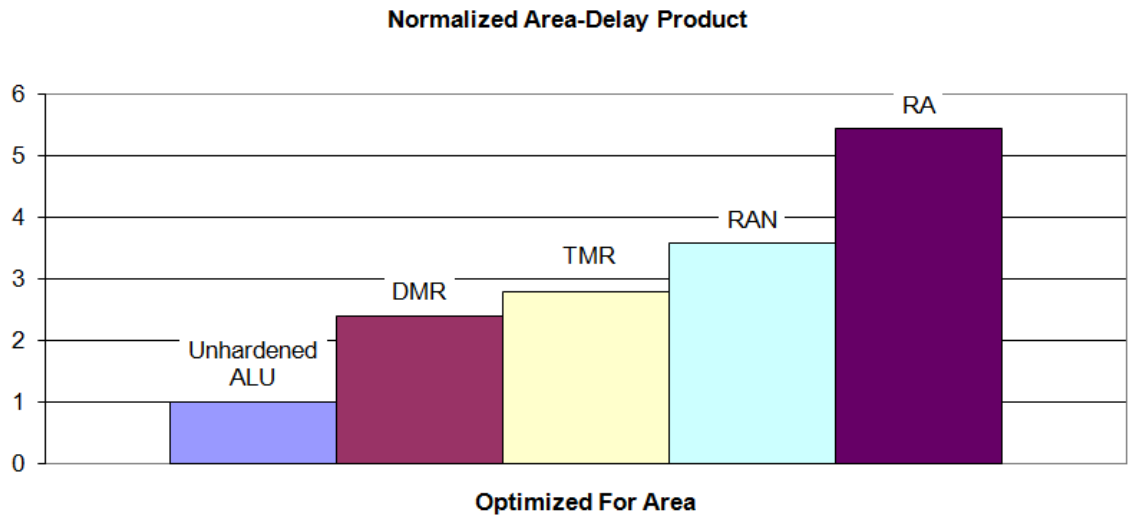

Figure 8. Area-Delay Product When Optimized For Area

The results when optimized for area are presented in Table 6. The area trends follow the previous synthesis. Adding redundancy again results in a near doubling of the area. It is noticeable about synthesizing for speed, the drastic increase in speed for the Repeat-Always and Repeat-As-Needed circuits compared to the area-optimized versions. Fig. 9 presents the Area-Delay Product of each implementation, normalized to the unhardened ALU. When again taking area and speed into account, the two Berger implementations are significantly outperformed by DMR and TMR. The Repeat-Always is again significantly worse.

Table 6. Optimized For Speed

	cell area (μm^2)	**data arrival time (ps)**
Unhardened ALU	3276	748
DMR	6271	906
TMR	8750	804
Repeat As Needed	6325	1636
Repeat Always	7756	2018

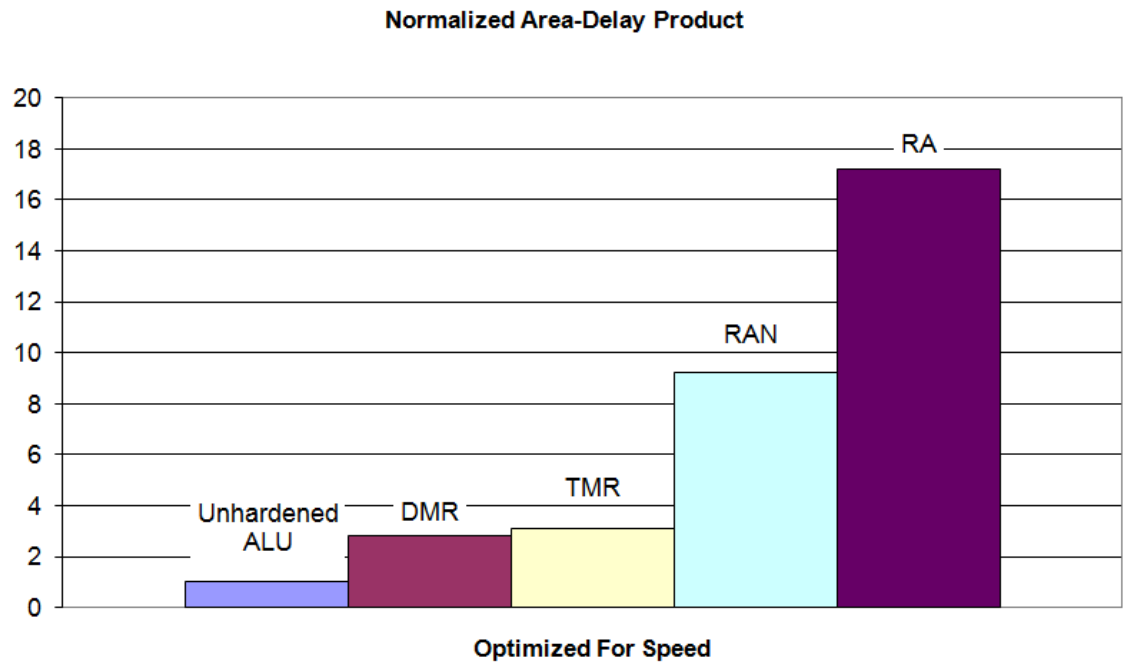

Figure 9. Area-Delay Product When Optimized For Speed

4.) FAULT-TOLERANT ALU – ACCURACY OF COMPUTATION

The total number of instructions run for the fault injection simulation was 10,404. The percentage of injected faults to errors for the unhardened ALU was 19 %. These faults occurred when a fault was injected during the sensitive window, the set-up-and-hold time of a latch. Faults that occurred in this window also caused the majority of instruction to be repeated in the Repeat-As-Needed implementation. This is shown in Fig. 10. There were still 4.3 % of faults that were not detected. A fault injected into the Repeat-Always circuit resulted in an error 63.7 % of the time. The execution time for the Repeat-As-Needed case increased by 63.4 % because of the number of additional cycles required to address the faults. The execution time for the unhardened ALU and Repeat-Always circuit were unaffected.

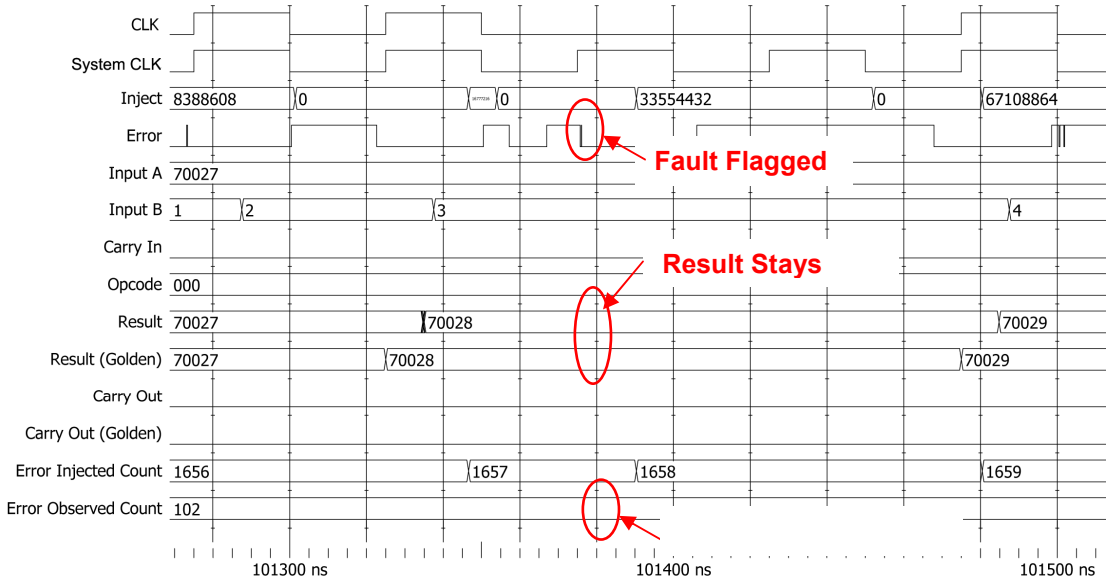

Figure 10. Repeat-As-Needed Error Detected and Corrected

Analysis of the fault injection results revealed a couple vulnerabilities of the BCP ALU and DMR ALUs. The fundamental theorem of Berger code is that data are sent through asynchronous channels. This is so that an error in one channel does not manifest as an error in the other channel. This allows for the data to be compared and an error identified, but the different delay times of the two channels create a vulnerability window where an error can be latched before it is detected. The vulnerability window is described as follows. When an error is injected close to a rising edge, there is a time during which a fault is present on the output latch, but the single event has not propagated through the zeros counter. This fault is also present in a DMR implementation, although the window is smaller, since the comparator for the DMR is faster than the zeros counter. The normal operation of the clock and the fault condition are shown in Fig. 11, and the fault condition is shown in Fig. 12.

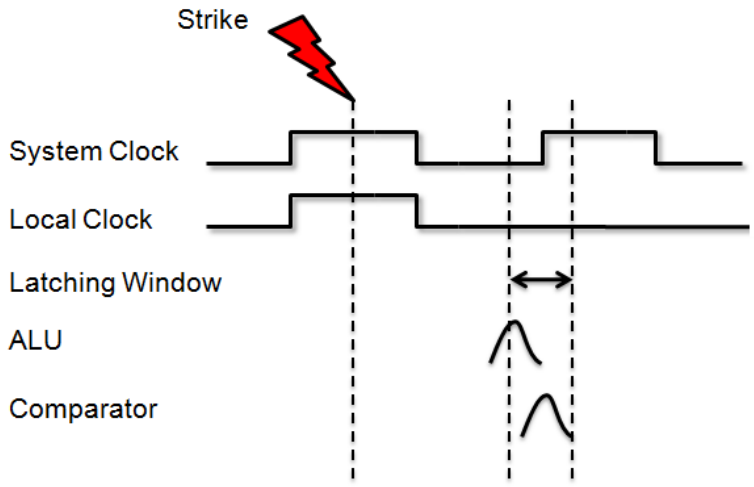

Figure 11. Example of Detected Pulse

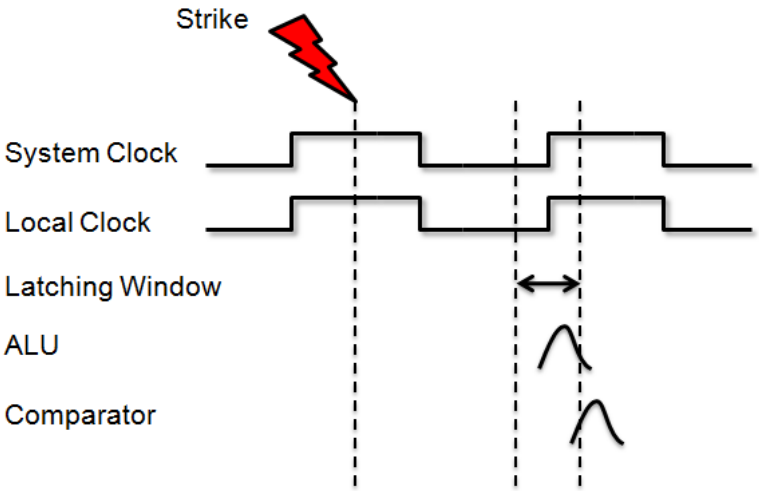

Figure 12. Example of Undetected Pulse

Repeating the fault injection simulation with a DMR ALU resulted in an error rate of 2.6 %. This type of error is shown in Fig. 13. The corresponding error with the Repeat-As-Needed Implementation is shown in Fig. 14.

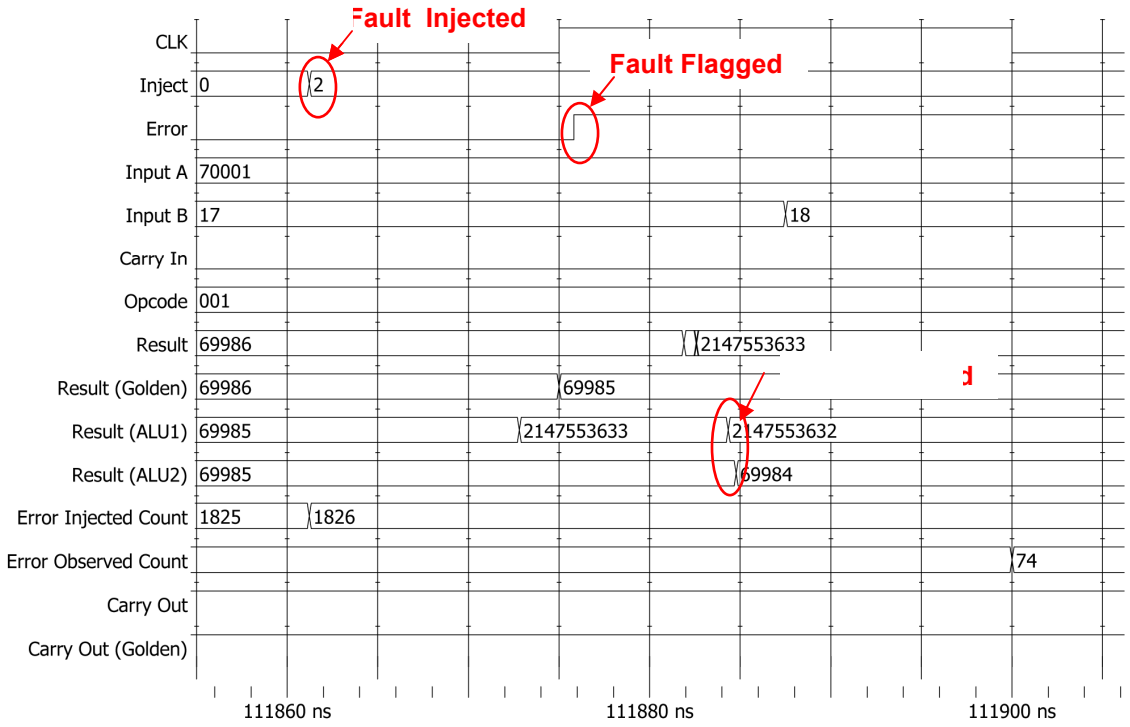

Figure 13. DMR Fault Not Detected

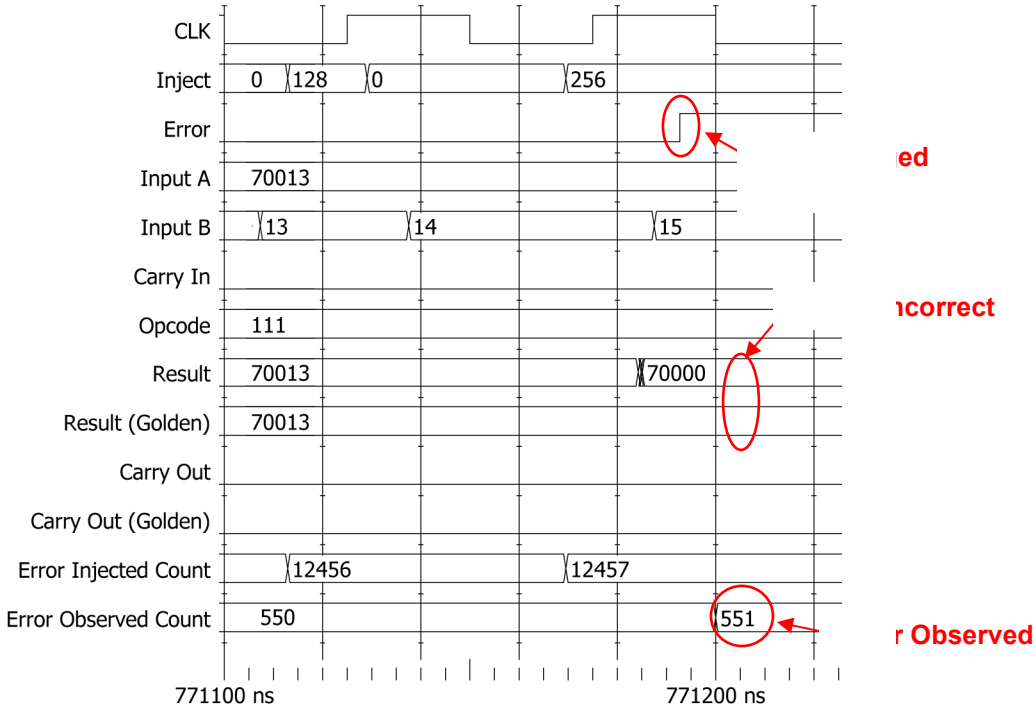

Figure 14. Repeat-As-Needed Fault Not Detected

Other results from the fault injection simulation show that the Repeat-Always implementation is vulnerable to SETs longer than half a clock cycle, since errors can be present on both halves of the clock cycle. Given this sensitivity, this implementation is not advised for environments where the maximum SET pulse width is greater than the clock period. This would allow the SET to strike without causing an error on both the positive and negative edge of the clock.

Both implementations provide the capability of soft error detection and correction. The vulnerability of the Repeat-As-Needed implementation is limited only by the inherent flaws in BCP circuits. The repeating of instructions eliminated the latching of soft errors, but at the cost of a performance penalty. In addition to this, there is an increased complexity in handling the instruction repetition. The repeating of the clock period would require external synchronization with the other components of the circuit. There would also have to be a watchdog timer to turn off the BCP circuit in the event of a static fault, otherwise the system would enter into an infinite stuck state.

The Repeat-Always implementation provides its single event protection at a reduced external complexity, but at a high performance penalty. While it does provide approximately a ten-fold decrease in single event sensitivity, the increase in logic elements and the associated area and power increases are likely prohibitive and a Repeat-As-Needed method is the better implementation. Neither of them; however, compare to DMR or TMR in terms of area, by extension power, and speed.

The delay increases for the Repeat-As-Needed is because the BCP requires the carry inputs from the ALU in order to complete arithmetic operations. There is also a

delay before it can compute the check symbol from the ALU result. This is the delay of the zeros counter that computes the check symbol from the ALU. The clock frequency in the Repeat-Always case is decreased because the result for the first half of the clock cycle is latched at the falling edge. This requires the clock frequencies to be decreased since the result must be computed in half the clock cycle.

CHAPTER VII

VII. CONCLUSIONS

The theoretical ALU with Berger check prediction from [15] can detect all unidirectional errors in both logic and arithmetic operations. The result as tested here was that it could detect 95.7% of errors. The Berger check system was used as a proof-of-concept to present two architectural methods which provide single event upset detection and correction by analyzing and correcting the data as it passes between the latches and through the combinational logic. The first approach only intervened when an error was detected and repeated the instruction. The second approach always repeated the instruction on the falling clock edge and corrected the error without additional penalty to the circuit performance. The Repeat-As-Needed scheme corrected all injected faults, but at a performance penalty of (2 + N) cycles per error. The Repeat-Always method corrected 97.2% of the faults, but suffered a reduction in performance due to a slowing of the clock. When compared to TMR, the Repeat-as-Needed method required less logical elements, but the Repeat-Always case required more elements due its latching of the result on both clock edges. Both implementations show an effective means to detect and recover from radiation-induced soft errors; however the area cost and speed penalties of these implementations are too severe for practical use.

CHAPTER VIII.

VIII. REFERENCES

[1] M. Santarini, "Cosmic radiation comes to ASIC and SOC design," *EDN*, 2005.

[2] S. W. Fu, *et al.*, "Alpha-particle-induced charge collection measurements and the effectiveness of a novel p-well protection barrier on VLSI memories," *IEEE Trans. Electron Devices*, vol. 32, pp. 49 – 54, 1985.

[3] M. P. Baze *et al.*, "SEU hardening techniques for retargetable sub-micron digital libraries," *2002 Single Event Effects Symp.*, Manhattan Beach, CA, 2002.

[4] D. R. Alexander *et al.*, "Design issues for radiation tolerant microcircuits in space," *IEEE NSREC Short Course*, 1996.

[5] G. Anelli *et al.*, "Radiation tolerant VLSI circuits in standard deep submicron CMOS technologies for the LHC experiments: practical design aspects," *IEEE Trans. on Nuclear Science*, vol. 46, pp. 1690 - 1696, 1999.

[6] O. A. Amusan *et al.*, "Charge collection and charge sharing in a 130 nm CMOS technology," *IEEE Trans. on Nuclear Science*, vol. 53, pp. 3253 - 3258, 2006.

[7] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Trans. on Device and Materials Reliability*, vol. 5, pp. 305 - 316, 2005.

[8] P. Shivakumar *et al.*, "Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic," *Proc. Int. Conf. Dependable Sys. and Networks*, pp 389 – 398, 2002.

[9] J.C. Lo *et al.*, "Concurrent Error Detection In Arithmetic and Logical Operations Using Berger Codes," *Proc. 9th Symp. Comput. Arithmetic*, pp. 233 - 240, 1989.

[10] W. Robinson *et al.*, "Soft Error Considerations for Multicore Microprocessor Design," *IEEE Int. Conf. on Integrated Circuit Design and Technology*, pp. 1 - 4, 2007

[11] S. S. Mukherjee *et al.*, "The Soft Error Problem: An Architectural Perspective," *Proc. 11th Int. Symp. High-Performance Comput. Architecture*, pp. 243 – 247, 2005.

[12] C.T. Weaver *et al.*, "Reducing the soft-error rate of a high-performance microprocessor," *IEEE Micro*, vol. 24, no 6, pp. 30 – 37, 2004

[13] S. Mukherjee, *Architecture Design For Soft Errors*. Burlington, MA. Morgan Kaufman, 2008.

[14] D.A. Pierce Jr. and P.K. Lala, "Modular Implementation of Efficient Self-Checking Checkers for the Berger Code," *J. of Electronic Testing: Theory and Applicat.*, pp. 279 - 294, 1996.

[15] J.C. Lo *et al.*, "An SFS Berger check prediction ALU and its application to self-checking processor designs," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 11, no. 4, pp.525 – 540, 1992.

[16] S.S. Gorshe and B. Bose, “A self-checking ALU design with efficient codes,” *Proc. 14th VLSI Test Symp.*, pp.157 – 161, 1996.
[17] B.L. Narasimham *et al.*, "Characterization of digital single event transient pulse-widths in 130-nm and 90-nm CMOS technologies," *IEEE Trans. on Nuclear Science*, vol. 54, pp. 2506 - 2511, 2007.
[18] B. Narasimham *et al.*, "The effect of negative feedback on single event transient propagation in digital circuits," *IEEE Trans. on Nuclear Science*, vol. 53, pp. 3285 - 3290, 2006.
[19] http://www.altera.com.
[20] F. Lima *et al.*, “On the use of VHDL simulation and emulation to derive error rates,” *6th European Conf. Radiation and Its Effects on Components and Syst.,* pp. 253 – 260, 2001.
[21] J.E. Stine *et al.*, “FreePDK: An Open-Source Variation-Aware Design Kit,” *IEEE Int. Conf. Microelectronic Sys. Educ.*, pp. 173 – 174, 2007.

APPENDIX A

REPEAT-AS-NEEDED AND REPEAT-ALWAYS VHDL BEHAVIORAL DESCRIPTION

This appendix presents the VHDL behavioral description of the fault tolerant ALU as used in the Repeat-As-Needed and Repeat-Always circuits. The Repeat-As-Needed implementation's top level design entity is given first, followed by its ALU and Berger Calculator subcircuits. The Repeat-Always top level design is then shown. Its subcircuits are identical to the Repeat-As-Needed scheme and are not included.

imp_one (Repeat-As-Needed Top Level Design Entity)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY imp_one IS
PORT(
              clk:          IN STD_LOGIC;
              carry_in:     IN STD_LOGIC;
              inject:       IN STD_LOGIC_VECTOR(31 DOWNTO 0);
              in_A:         IN STD_LOGIC_VECTOR(31 DOWNTO 0);
              in_B:         IN STD_LOGIC_VECTOR(31 DOWNTO 0);
              opcode:       IN STD_LOGIC_VECTOR(2 DOWNTO 0);
              result:       OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
              carry_out:    OUT STD_LOGIC;
              alu_c_out:    OUT STD_LOGIC_VECTOR(5 DOWNTO 0);
              bcp_c_out:    OUT STD_LOGIC_VECTOR(5 DOWNTO 0);
              error_out:    OUT STD_LOGIC
);

END imp_one;

ARCHITECTURE rtl OF imp_one IS

SIGNAL carry_out_stage2:   STD_LOGIC;
SIGNAL carry_out_alu:      STD_LOGIC;
SIGNAL result_out_alu:     STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL carry:              STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL bcp_c:              STD_LOGIC_VECTOR(5 DOWNTO 0);
SIGNAL alu_c:              STD_LOGIC_VECTOR(5 DOWNTO 0);
SIGNAL in_A_stage2:        STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL in_B_stage2:        STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL opcode_stage2:      STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL result_stage2:      STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL carry_in_stage2:    STD_LOGIC;
SIGNAL mux_sel:            STD_LOGIC;
SIGNAL alu_bcp_sel:        STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL error:              STD_LOGIC;
```

```
SIGNAL eff_clk:            STD_LOGIC;
SIGNAL in_latch:           STD_LOGIC;
SIGNAL out_latch:          STD_LOGIC;
SIGNAL preop:              STD_LOGIC_VECTOR(2 DOWNTO 0);

COMPONENT sc_alu IS
PORT(
              carry_in:    IN STD_LOGIC;
              in_A:        IN STD_LOGIC_VECTOR(31 DOWNTO 0);
              in_B:        IN STD_LOGIC_VECTOR(31 DOWNTO 0);
              opcode:      IN STD_LOGIC_VECTOR(2 DOWNTO 0);
              inject:      IN STD_LOGIC_VECTOR(31 DOWNTO 0);
              result:      OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
              carry_out:   OUT STD_LOGIC;
              eff_clk_out: OUT STD_LOGIC;
              alu_c_out:   OUT STD_LOGIC_VECTOR(5 DOWNTO 0);
              bcp_c_out:   OUT STD_LOGIC_VECTOR(5 DOWNTO 0);
              error_out:   OUT STD_LOGIC
);
END COMPONENT sc_alu;


BEGIN

       SCALU1: sc_alu
       PORT MAP(
       --into SC_ALU
              in_A => in_A_stage2,
              in_B => in_B_stage2,
              opcode => opcode_stage2,
              carry_in => carry_in_stage2,
              inject => inject,
       --out of SC_ALU
              result => result_out_alu,
              carry_out => carry_out_alu,
              error_out => error,
              alu_c_out => alu_c_out,
              bcp_c_out => bcp_c_out
              );

       error_out <= error;


       PROCESS(clk, in_A, in_B, opcode)
       BEGIN
              IF(RISING_EDGE(clk)) THEN
                     in_A_stage2 <= in_A;
                     in_B_stage2 <= in_B;
                     carry_in_stage2 <= carry_in;
                     opcode_stage2 <= opcode;
                     result <= result_out_alu;
                     carry_out <= carry_out_alu;
              END IF;
       END PROCESS;

END rtl;
```

sc_alu

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY sc_alu IS
PORT(
              clk:         IN STD_LOGIC;
              carry_in:    IN STD_LOGIC;
              inject:      IN STD_LOGIC_VECTOR(31 DOWNTO 0);
              in_A:        IN STD_LOGIC_VECTOR(31 DOWNTO 0);
              in_B:        IN STD_LOGIC_VECTOR(31 DOWNTO 0);
              opcode:      IN STD_LOGIC_VECTOR(2 DOWNTO 0);
```

```
              result:       OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
              carry_out:    OUT STD_LOGIC;
              alu_c_out:    OUT STD_LOGIC_VECTOR(5 DOWNTO 0);
              bcp_c_out:    OUT STD_LOGIC_VECTOR(5 DOWNTO 0);
              eff_clk_out:  OUT STD_LOGIC;
              error_out:    OUT STD_LOGIC
);

END sc_alu;

ARCHITECTURE rtl OF sc_alu IS

--SIGNAL carry_in:         STD_LOGIC;
SIGNAL carry_out_stage2:   STD_LOGIC;
SIGNAL carry_out_alu:      STD_LOGIC;
SIGNAL result_out_alu:     STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL carry:              STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL bcp_c:              STD_LOGIC_VECTOR(5 DOWNTO 0);
SIGNAL alu_c:              STD_LOGIC_VECTOR(5 DOWNTO 0);

SIGNAL in_A_alu:           STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL in_A_bcp:           STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL in_B_alu:           STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL in_B_bcp:           STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL result_stage2:      STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL mux_sel:            STD_LOGIC;
SIGNAL alu_bcp_sel:        STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL opcode_stage2:      STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL error:              STD_LOGIC;
SIGNAL eff_clk:            STD_LOGIC;
SIGNAL in_latch:           STD_LOGIC;
SIGNAL out_latch:          STD_LOGIC;
SIGNAL test: STD_LOGIC_VECTOR(2 DOWNTO 0);

COMPONENT alu IS
PORT(
              in_A:         IN STD_LOGIC_VECTOR(31 DOWNTO 0);
              in_B:         IN STD_LOGIC_VECTOR(31 DOWNTO 0);
              opcode:       IN STD_LOGIC_VECTOR(2 DOWNTO 0);
              carry_in:     IN STD_LOGIC;
              inject:       IN STD_LOGIC_VECTOR(31 DOWNTO 0);
              result:       OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
              carry_out:    OUT STD_LOGIC;
              carry:        OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
);
END COMPONENT alu;

COMPONENT bcp IS
PORT(
              in_A:         IN STD_LOGIC_VECTOR(31 DOWNTO 0);
              in_B:         IN STD_LOGIC_VECTOR(31 DOWNTO 0);
              in_C:         IN STD_LOGIC_VECTOR(31 DOWNTO 0);
              inject:       IN STD_LOGIC_VECTOR(31 DOWNTO 0);
              carry_in:     IN STD_LOGIC;
              carry_out:    IN STD_LOGIC;
              opcode:       IN STD_LOGIC_VECTOR(2 DOWNTO 0);
              zA_out:       OUT STD_LOGIC_VECTOR (5 DOWNTO 0);
              zB_out:       OUT STD_LOGIC_VECTOR (5 DOWNTO 0);
              zC_out:       OUT STD_LOGIC_VECTOR (5 DOWNTO 0);
              result_c:     OUT STD_LOGIC_VECTOR(5 DOWNTO 0)
);
END COMPONENT bcp;

component zero32
      port(in_A: in std_logic_vector(31 DOWNTO 0);
           z_A: out std_logic_vector(5 DOWNTO 0));
end component;

BEGIN

       ALU1: alu
       PORT MAP(
```

```
	--into ALU
		in_A => (in_A_alu(31) XOR inject(1)) & in_A_alu(30 DOWNTO 1) &
(in_A_alu(0) XOR inject(0)),
		in_B => in_B_alu(31 DOWNTO 15) & (in_B_alu(14) XOR inject(2)) &
in_B_alu(13 DOWNTO 7) & (in_B_alu(6) XOR inject(3)) & in_B_alu(5 DOWNTO 0),
		opcode => opcode_stage2(2) & (opcode_stage2(1) XOR inject(8)) &
opcode_stage2(0),
		carry_in => carry_in,
		inject => inject,
	--out of ALU
		result => result_out_alu,
		carry_out => carry_out_alu,
		carry => carry
	);

	BCP1: bcp
	PORT MAP(
	--into BCP
		in_A => in_A_bcp(31 DOWNTO 29) & (in_A_bcp(28) XOR inject(4)) &
in_A_bcp(27 DOWNTO 1) & (in_A_bcp(0) XOR inject(5)),
		in_B => in_B_bcp(31 DOWNTO 22) & (in_B_bcp(21) XOR inject(6)) &
in_B_bcp(20 DOWNTO 9) & (in_B_bcp(8) XOR inject(7)) & in_B_bcp(7 DOWNTO 0),
		in_C => carry(31 DOWNTO 15) & (carry(14) XOR inject(10)) &
carry(13 DOWNTO 0),
		inject => inject,
		carry_out => carry_out_alu XOR inject(11),
		carry_in => carry_in XOR inject(13),
		opcode => (opcode_stage2(2) XOR inject(20)) & opcode_stage2(1
DOWNTO 0),
	--out of BCP
		result_c => bcp_c
	);

	stage1: zero32 port map(result_out_alu, alu_c);

PROCESS(in_A, in_B, opcode, alu_c, bcp_c)
begin
			in_A_alu <= in_A;
			in_B_alu <= in_B;

			in_A_bcp <= in_A;
			in_B_bcp <= in_B;

			opcode_stage2 <= opcode;
			result <= result_out_alu;
			carry_out <= carry_out_alu;
			alu_c_out <= alu_c;
			bcp_c_out <= bcp_c;
				IF ((alu_c(5 DOWNTO 1)& (alu_c(0) XOR inject(14))) =
((bcp_c(5) XOR inject(16)) & bcp_c(4 DOWNTO 3)& (bcp_c(2) XOR inject(15))&
bcp_c(1 DOWNTO 0)))  THEN
					error <= '0';
				ELSE
					error <= '1';
				END IF;
			error_out <= error;

	END PROCESS;

END rtl;
```

alu

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_unsigned.all;

ENTITY alu IS
PORT(
		in_A:		IN STD_LOGIC_VECTOR(31 DOWNTO 0);
		in_B:		IN STD_LOGIC_VECTOR(31 DOWNTO 0);
```

```
              carry_in:    IN STD_LOGIC;
              opcode:      IN STD_LOGIC_VECTOR(2 DOWNTO 0);
              inject:      IN STD_LOGIC_VECTOR(31 DOWNTO 0);
              result:      OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
              carry_out:   OUT STD_LOGIC;
              carry:       OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
);
END alu;


ARCHITECTURE structure OF alu IS
SIGNAL sum: STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL in_B_addr: STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL all_ones: STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL carry_in_addr: STD_LOGIC;
SIGNAL carry_addr: STD_LOGIC_VECTOR(31 DOWNTO 0):= (others => '1') ;
SIGNAL carry_out_addr: STD_LOGIC;

COMPONENT c_l_addr IS
PORT(
         x_in       :  IN   STD_LOGIC_VECTOR(31 DOWNTO 0);
         y_in       :  IN   STD_LOGIC_VECTOR(31 DOWNTO 0);
         carry_in   :  IN   STD_LOGIC;
         inject     :  IN   STD_LOGIC_VECTOR(31 DOWNTO 0);
         sum        :  OUT  STD_LOGIC_VECTOR(31 DOWNTO 0);
         carry      :  OUT  STD_LOGIC_VECTOR(31 DOWNTO 0);
         carry_out  :  OUT  STD_LOGIC
);
END COMPONENT c_l_addr;
BEGIN

       ADDR1: c_l_addr
       PORT MAP(
       --into ADDR1
              x_in => in_A,
              y_in => in_B_addr,
              carry_in => carry_in_addr XOR inject(9),
              inject => inject,
        --out of ADDR1
              sum => sum,
              carry_out => carry_out_addr,
              carry => carry_addr
              );


PROCESS(opcode, in_A, in_B, carry_in, sum, carry_addr, carry_out_addr)
       BEGIN
                     in_B_addr <= in_B;
                     carry_in_addr <= carry_in;

              IF opcode = "000" THEN -- S = A + B + carry_in
                     result <= sum;
                     carry <= carry_addr;
                     carry_out <= carry_out_addr;
              ELSIF opcode = "001" THEN -- S = A - B - carry_in
                     in_B_addr <= NOT(in_B);
                     carry_in_addr <= NOT(carry_in);
                     result <= sum;
                     carry <= carry_addr;
                     carry_out <= carry_out_addr;
              ELSIF opcode = "010" THEN -- Rotate Left (xn-1,....x0,xn)
                     result <= in_A(30 DOWNTO 0) & in_A(31);
                     carry_out <= '0';
                     carry <= (others => '0');
              ELSIF opcode = "011" THEN -- Rotate Right (x0,xn,....x1)
                     result <= in_A(0)&in_A(31 DOWNTO 1);
                     carry_out <= '0';
                     carry <= (others => '0');
              ELSIF opcode = "100" THEN  -- S = A AND B
                     result <= in_A and in_B;
                     carry_out <= '0';
                     carry <= (others => '0');
```

```
            ELSIF opcode = "101" THEN -- S = A XOR B
                  result <=in_A xor in_B;
                  carry_out <= '0';
                  carry <= (others => '0');
            ELSIF opcode = "110" THEN  -- S = A OR B
                  result <=in_A or in_B;
                  carry_out <= '0';
                  carry <= (others => '0');
            ELSIF opcode = "111" THEN  -- S = A
                  result <=in_A;
                  carry_out <= '0';
                  carry <= (others => '0');
            ELSE
                  result <= (others => '0');
                  carry_out <= '0';
                  carry <= (others => '0');
            END IF;
      END PROCESS;
END structure;
```

c_l_addr

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY c_l_addr IS
    PORT
        (
         x_in      :  IN   STD_LOGIC_VECTOR(31 DOWNTO 0);
         y_in      :  IN   STD_LOGIC_VECTOR(31 DOWNTO 0);
         carry_in  :  IN   STD_LOGIC;
         sum       :  OUT  STD_LOGIC_VECTOR(31 DOWNTO 0);
         carry     :  OUT  STD_LOGIC_VECTOR(31 DOWNTO 0);
         carry_out :  OUT  STD_LOGIC
        );
END c_l_addr;

ARCHITECTURE behavioral OF c_l_addr IS

SIGNAL    h_sum              :    STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL    carry_generate     :    STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL    carry_propagate    :    STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL    carry_in_internal  :    STD_LOGIC_VECTOR(31 DOWNTO 1);

BEGIN
    h_sum <= x_in XOR y_in;
    carry_generate <= x_in AND y_in;
    carry_propagate <= x_in OR y_in;
    PROCESS (carry_generate,carry_propagate,carry_in_internal)
    BEGIN
    carry_in_internal(1) <= carry_generate(0) OR (carry_propagate(0) AND
carry_in);
        inst: FOR i IN 1 TO 30 LOOP
              carry_in_internal(i+1) <= carry_generate(i) OR
(carry_propagate(i) AND carry_in_internal(i));
              END LOOP;
    carry_out <= carry_generate(31) OR (carry_propagate(31) AND
carry_in_internal(31));
    END PROCESS;

    sum(0) <= h_sum(0) XOR carry_in;
    sum(31 DOWNTO 1) <= h_sum(31 DOWNTO 1) XOR carry_in_internal(31 DOWNTO 1);
    carry <= (carry_generate(31) OR (carry_propagate(31) AND
carry_in_internal(31))) & carry_in_internal(31 DOWNTO 1);
END behavioral;

bcp.vhd
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_unsigned.all;
```

```
ENTITY bcp IS
PORT(
            in_A:         IN STD_LOGIC_VECTOR(31 DOWNTO 0);
            in_B:         IN STD_LOGIC_VECTOR(31 DOWNTO 0);
            in_C:         IN STD_LOGIC_VECTOR(31 DOWNTO 0);
            carry_in:     IN STD_LOGIC;
            carry_out:    IN STD_LOGIC;
            inject:       IN STD_LOGIC_VECTOR(31 DOWNTO 0);
            opcode:       IN STD_LOGIC_VECTOR(2 DOWNTO 0);
            result_c:     OUT STD_LOGIC_VECTOR(5 DOWNTO 0);
            zA_out:       OUT STD_LOGIC_VECTOR (5 DOWNTO 0);
            zB_out:       OUT STD_LOGIC_VECTOR (5 DOWNTO 0);
            zC_out:       OUT STD_LOGIC_VECTOR (5 DOWNTO 0)
);
END bcp;

ARCHITECTURE structure OF bcp IS
SIGNAL t:           STD_LOGIC_VECTOR(5 DOWNTO 1);
SIGNAL d:           STD_LOGIC_VECTOR(1 DOWNTO 0);
SIGNAL and_sig:     STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL or_sig:      STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL mux_sig:     STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL mux_sel:     STD_LOGIC_VECTOR(1 DOWNTO 0);
SIGNAL z_A:         STD_LOGIC_VECTOR(5 DOWNTO 0);
SIGNAL z_B:         STD_LOGIC_VECTOR(5 DOWNTO 0);
SIGNAL z_C:         STD_LOGIC_VECTOR(5 DOWNTO 0);
SIGNAL mult_C:      STD_LOGIC_VECTOR(5 DOWNTO 0);
SIGNAL nand_C:      STD_LOGIC_VECTOR(5 DOWNTO 0);
SIGNAL xor_B:       STD_LOGIC_VECTOR(5 DOWNTO 0);
SIGNAL and_B:       STD_LOGIC_VECTOR(5 DOWNTO 0);
SIGNAL mcsa_result: STD_LOGIC_VECTOR(5 DOWNTO 0);

COMPONENT pla IS
PORT(
            carry_in:   IN STD_LOGIC;
            carry_out:  IN STD_LOGIC;
            opcode:      IN STD_LOGIC_VECTOR(2 DOWNTO 0);
            inject:      IN STD_LOGIC_VECTOR(31 DOWNTO 0);
            t_out:                    OUT STD_LOGIC_VECTOR(5 DOWNTO 1);
            d_out:                    OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
);
END COMPONENT pla;

COMPONENT mcsa IS
PORT(
            x_c:         IN STD_LOGIC_VECTOR(5 DOWNTO 0);
            y_c:         IN STD_LOGIC_VECTOR(5 DOWNTO 0);
            c_c:         IN STD_LOGIC_VECTOR(5 DOWNTO 0);
            d:           IN STD_LOGIC_VECTOR(1 DOWNTO 0);
            result:      OUT STD_LOGIC_VECTOR(5 DOWNTO 0)
);
END COMPONENT mcsa;

component zero32
      port(in_A: in std_logic_vector(31 DOWNTO 0);
           z_A: out std_logic_vector(5 DOWNTO 0));
end component;

BEGIN
       PLA1: pla PORT MAP (
             carry_in     => carry_in,
             carry_out    => carry_out,
             opcode       => opcode,
             inject       => inject,
             t_out        => t,
             d_out        => d
       );

       MCSA1: mcsa PORT MAP (
             x_c          => z_A,
             y_c          => xor_B(5 DOWNTO 1) & (xor_B(0) XOR inject(31)),
             c_c          => (nand_C(5) XOR inject(29)) & nand_C(4 DOWNTO 0),
```

```
		d		=> d,
		result => mcsa_result
	);

	--path for C input
	PROCESS(opcode, in_A, in_B, t, mult_C, nand_C)
	BEGIN
		and_sig	<= in_A AND in_B;
		or_sig	<= in_A OR in_B;
		mux_sel	<= opcode(2) & t(1);
		nand_C	<= NOT ((t(3)&t(3)&t(3)&t(3)&t(3)&t(3)) AND mult_C);
		zC_out	<= nand_C;
	END PROCESS;

	--MUX to select operand
	WITH mux_sel SELECT
		mux_sig <=	in_C(31 DOWNTO 20) & (in_C(19) XOR inject(19)) &
in_C(18 DOWNTO 0)	WHEN "00",
				or_sig(31 DOWNTO 19) & (or_sig(18) XOR
inject(18)) & or_sig(17 DOWNTO 0)	WHEN "01",
				and_sig(31 DOWNTO 18) & (and_sig(17) XOR
inject(17)) & and_sig(16 DOWNTO 0)	WHEN "10",
				or_sig WHEN "11";

	--zeros counter for input A, B, C
	stageA: zero32 port map(in_A, z_A);
	stageB: zero32 port map(in_B, z_B);
	stageC:	zero32 port map(mux_sig, z_C);

	--multiply by 2
	WITH t(2) SELECT
		mult_C <=	z_C				WHEN '0',
				z_C(4 DOWNTO 0)&"0"		WHEN '1';

	--path for B input
	PROCESS(and_B, z_B, t, xor_B, inject)
	BEGIN
		--and_B <= z_B AND (t(3)&t(3)&t(3)&t(3)&t(3)&t(3));
		and_B <= (z_B(5 DOWNTO 3) & (z_B(2) XOR inject(30)) & z_B(1 DOWNTO
0)) AND (t(3)&t(3)&t(3)&t(3)&t(3)&t(3));
		xor_B <= and_B XOR (t(4)&t(4)&t(4)&t(4)&t(4)&t(4));
		zB_out <= xor_B;
	END PROCESS;

	--output
	WITH t(5) SELECT
		result_c <= mcsa_result			WHEN '0',
				mcsa_result + "100000"	WHEN '1'; -- PLUS N

END structure;
```

pla

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_unsigned.all;

ENTITY pla IS
PORT(
		carry_in:	IN STD_LOGIC;
		carry_out:	IN STD_LOGIC;
		opcode:		IN STD_LOGIC_VECTOR(2 DOWNTO 0);
		inject:		IN STD_LOGIC_VECTOR(31 DOWNTO 0);
		t_out:		OUT STD_LOGIC_VECTOR(5 DOWNTO 1);
		d_out:		OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
);
END pla;

ARCHITECTURE structure OF pla IS
SIGNAL c_in, c_out, d: STD_LOGIC_VECTOR (1 DOWNTO 0);
SIGNAL t : STD_LOGIC_VECTOR (5 DOWNTO 1);
```

```
BEGIN

	PROCESS(opcode, carry_in, carry_out, c_in, c_out)
	BEGIN
		t <= "00000";
		d <= "00";
		c_in <= "0" & carry_in;
		c_out <= "0" & carry_out;

		CASE(opcode) IS
			WHEN "000" => -- X+Y+cin
				t <= "00100";
				d <= c_out - c_in + 1;
			WHEN "001" => -- X-Y-1-cin
				t <= "11100";
				d <= c_out - ("0" & NOT(carry_in)) + 2;
			WHEN "010" => -- Rotate Left (xn-1,....x0,xn)
				t <= "00000";
				d <= "01";
			WHEN "011" => -- Rotate Right (x0,xn,....x1)
				t <= "00000";
				d <= "01";
			WHEN "100" => --AND
				t <= "00101";
				d <= "01";
			WHEN "101" => -- XOR
				t <= "10110";
				d <= "01";
			WHEN "110" => --OR
				t <= "00100";
				d <= "01";
			WHEN "111" => --Identity
				t <= "00000";
				d <= "01";
			WHEN OTHERS =>
				t <= "00000";
				d <= "00";
		END CASE;
		t_out <= (t(5) XOR inject(25)) & (t(4) XOR inject(24)) & (t(3) XOR
inject(23)) & (t(2) XOR inject(22)) & (t(1) XOR inject(21));
		--t_out <= "00000";
		d_out <= d(1) & (d(0) XOR inject(27));
	END PROCESS;
END structure;
```

mcsa

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_unsigned.all;

ENTITY mcsa IS
PORT(
		x_c:		IN STD_LOGIC_VECTOR(5 DOWNTO 0);
		y_c:		IN STD_LOGIC_VECTOR(5 DOWNTO 0);
		c_c:		IN STD_LOGIC_VECTOR(5 DOWNTO 0);
		d:	IN STD_LOGIC_VECTOR(1 DOWNTO 0);
		result: OUT STD_LOGIC_VECTOR(5 DOWNTO 0)
);
END mcsa;

ARCHITECTURE structure OF mcsa IS

SIGNAL partial_sum: STD_LOGIC_VECTOR (5 DOWNTO 0);
SIGNAL shift_carry: STD_LOGIC_VECTOR (5 DOWNTO 0);
SIGNAL ps_sc_sum:   STD_LOGIC_VECTOR (6 DOWNTO 0);
SIGNAL ps:		STD_LOGIC_VECTOR (6 DOWNTO 0);
SIGNAL sc:		STD_LOGIC_VECTOR (6 DOWNTO 0);

BEGIN
```

```
	PROCESS(x_c, y_c, c_c, d, partial_sum, shift_carry, ps, sc, ps_sc_sum)
	BEGIN

		partial_sum <= x_c XOR y_c XOR c_c;
		shift_carry <=	(x_c AND y_c) OR (x_c AND c_c) OR (y_c AND
c_c);
		ps		<= "0" & partial_sum;
		sc		<= shift_carry & "0";
		ps_sc_sum	<= ps + sc + ("0000" & d);
		result	<= ps_sc_sum(5 DOWNTO 0);

	END PROCESS;
END structure;
```

zero32

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_unsigned.all;

ENTITY zero32 IS
PORT(  in_A:		IN STD_LOGIC_VECTOR(31 DOWNTO 0);
		--in_C:		IN STD_LOGIC_VECTOR(31 DOWNTO 0);
		z_A:		OUT STD_LOGIC_VECTOR(5 DOWNTO 0)
		--z_C:		OUT STD_LOGIC_VECTOR(5 DOWNTO 0)
	);
END zero32;

ARCHITECTURE structure OF zero32 IS

component zerocount
      port(in1: in std_logic_vector(3 DOWNTO 0);
           out1: out std_logic_vector(2 DOWNTO 0));
end component;

signal z_Aa, z_Ab, z_Ac, z_Ad, z_Ae, z_Af, z_Ag, z_Ah: std_logic_vector(2
DOWNTO 0);
signal z_Ai, z_Aj, z_Ak, z_Al: std_logic_vector(3 DOWNTO 0);
signal z_Am, z_An: std_logic_vector(4 DOWNTO 0);
--signal Cbar: std_logic_vector(31 downto 0);
--signal ia, ib, ic, id, ie, iff, ig, ih, ii, ij, ik, il, im, inn, io, ip:
std_logic_vector(1 DOWNTO 0);
--signal iq, ir, iss, it, iu, iv, iw, ix: std_logic_vector(2 DOWNTO 0);
--signal iy, iz, iaa, ibb: std_logic_vector(3 DOWNTO 0);
--signal icc, idd: std_logic_vector(4 DOWNTO 0);

BEGIN
--individual 4 bit 0's counters
	stage1: zerocount port map(in_A(31 DOWNTO 28), z_Aa);
	stage2: zerocount port map(in_A(27 DOWNTO 24), z_Ab);
	stage3: zerocount port map(in_A(23 DOWNTO 20), z_Ac);
	stage4: zerocount port map(in_A(19 DOWNTO 16), z_Ad);
	stage5: zerocount port map(in_A(15 DOWNTO 12), z_Ae);
	stage6: zerocount port map(in_A(11 DOWNTO 8), z_Af);
	stage7: zerocount port map(in_A(7 DOWNTO 4), z_Ag);
	stage8: zerocount port map(in_A(3 DOWNTO 0), z_Ah);

--add results of 4 bit 0's counters stage1
	z_Ai <= ("0"&z_Aa) + ("0"&z_Ab);
	z_Aj <= ("0"&z_Ac) + ("0"&z_Ad);
	z_Ak <= ("0"&z_Ae) + ("0"&z_Af);
	z_Al <= ("0"&z_Ag) + ("0"&z_Ah);
--add results of 4 bit 0's counters stage2
	z_Am <= ("0"&z_Ai) + ("0"&z_Aj);
	z_An <= ("0"&z_Ak) + ("0"&z_Al);
--add results of 4 bit 0's counters stage3
	Z_A <= ("0"&z_Am) + ("0"&z_An);

----32 stage 1 bit adders
```

```
--     z_B <=        NOT("00000" & in_B(0)) + NOT("00000" & in_B(1)) +
NOT("00000" & in_B(2)) + NOT("00000" & in_B(3)) +
--                   NOT("00000" & in_B(4)) + NOT("00000" & in_B(5)) +
NOT("00000" & in_B(6)) + NOT("00000" & in_B(7)) +
--                   NOT("00000" & in_B(8)) + NOT("00000" & in_B(9)) +
NOT("00000" & in_B(10)) + NOT("00000" & in_B(11)) +
--                   NOT("00000" & in_B(12)) + NOT("00000" & in_B(13)) +
NOT("00000" & in_B(14)) + NOT("00000" & in_B(15)) +
--                   NOT("00000" & in_B(16)) + NOT("00000" & in_B(17)) +
NOT("00000" & in_B(18)) + NOT("00000" & in_B(19)) +
--                   NOT("00000" & in_B(20)) + NOT("00000" & in_B(21)) +
NOT("00000" & in_B(22)) + NOT("00000" & in_B(23)) +
--                   NOT("00000" & in_B(24)) + NOT("00000" & in_B(25)) +
NOT("00000" & in_B(26)) + NOT("00000" & in_B(27)) +
--                   NOT("00000" & in_B(28)) + NOT("00000" & in_B(29)) +
NOT("00000" & in_B(30)) + NOT("00000" & in_B(31));
--
----cascaded in_C stage 1
--     Cbar <= NOT(in_C);
--     ia <= ("0" & Cbar(0))+("0" & Cbar(1));
--     ib <= ("0" & Cbar(2))+("0" & Cbar(3));
--     ic <= ("0" & Cbar(4))+("0" & Cbar(5));
--     id <= ("0" & Cbar(6))+("0" & Cbar(7));
--     ie <= ("0" & Cbar(8))+("0" & Cbar(9));
--     iff <= ("0" & Cbar(10))+("0" & Cbar(11));
--     ig <= ("0" & Cbar(12))+("0" & Cbar(13));
--     ih <= ("0" & Cbar(14))+("0" & Cbar(15));
--     ii <= ("0" & Cbar(16))+("0" & Cbar(17));
--     ij <= ("0" & Cbar(18))+("0" & Cbar(19));
--     ik <= ("0" & Cbar(20))+("0" & Cbar(21));
--     il <= ("0" & Cbar(22))+("0" & Cbar(23));
--     im <= ("0" & Cbar(24))+("0" & Cbar(25));
--     inn <= ("0" & Cbar(26))+("0" & Cbar(27));
--     io <= ("0" & Cbar(28))+("0" & Cbar(29));
--     ip <= ("0" & Cbar(30))+("0" & Cbar(31));
--
----cascaded in_C stage 2
--     iq <= ("0" & ia) + ("0" & ib);
--     ir <= ("0" & ic) + ("0" & id);
--     iss <= ("0" & ie) + ("0" & iff);
--     it <= ("0" & ig) + ("0" & ih);
--     iu <= ("0" & ii) + ("0" & ij);
--     iv <= ("0" & ik) + ("0" & il);
--     iw <= ("0" & im) + ("0" & inn);
--     ix <= ("0" & io) + ("0" & ip);
--
----cascaded in_C stage 3
--     iy <= ("0" & iq) + ("0" & ir);
--     iz <= ("0" & iss) + ("0" & it);
--     iaa <= ("0" & iu) + ("0" & iv);
--     ibb <= ("0" & iw) + ("0" & ix);
--
----cascaded in_C stage 4
--     icc <= ("0" & iy) + ("0" & iz);
--     idd <= ("0" & iaa) + ("0" & ibb);
--
----cascaded in_C stage 5
--     z_C <= ("0" & icc) + ("0" & idd);

END structure;
```

zerocount

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY zerocount IS
PORT(
             in1:          IN STD_LOGIC_VECTOR(3 DOWNTO 0);
```

```
            out1:         OUT STD_LOGIC_VECTOR(2 DOWNTO 0)
      );

END zerocount;

ARCHITECTURE rtl OF zerocount IS

SIGNAL ia, ib, ic, id, ie, iff, ig, ih, ij, ik, il, im, inn, ip, iq, ir:
STD_LOGIC;
SIGNAL inbar:       STD_LOGIC_VECTOR(3 DOWNTO 0);

BEGIN
      inbar <= NOT(in1);
      out1(2) <= inbar(3) AND inbar(2) AND inbar(1) AND inbar(0);

      ia <= inbar(0) OR inbar(1) OR inbar(2);
      ib <= inbar(0) OR inbar(2) OR inbar(3);
      ic <= inbar(0) OR inbar(1) OR inbar(3);
      id <= inbar(1) OR inbar(2) OR inbar(3);
      ie <= NOT(inbar(0) AND inbar(1) AND inbar(2) AND inbar(3));

      iff <= NOT(inbar(0)) AND inbar(1) AND NOT(inbar(2)) AND NOT(inbar(3));
      ig <= NOT(inbar(0)) AND NOT(inbar(1)) AND NOT(inbar(2)) AND inbar(3);
      ih <= NOT(inbar(0)) AND NOT(inbar(1)) AND inbar(2) AND NOT(inbar(3));
      ij <= inbar(0) AND NOT(inbar(1)) AND NOT(inbar(2)) AND NOT(inbar(3));
      ik <= inbar(0) AND inbar(1) AND NOT(inbar(2)) AND inbar(3);
      il <= NOT(inbar(0)) AND inbar(1) AND inbar(2) AND inbar(3);
      im <= inbar(0) AND NOT(inbar(1)) AND inbar(2) AND inbar(3);
      inn <= inbar(0) AND inbar(1) AND inbar(2) AND NOT(inbar(3));

      ip <= ia AND ib AND ic;
      iq <= iff OR ig OR ih OR ij;
      ir <= ik OR il OR im OR inn;

      out1(1) <= ip AND id AND ie;
      out1(0) <= iq OR ir;

END rtl;
```

sc_alu (Repeat-As-Needed Top Level Design Entity)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY sc_alu IS
PORT(
            clk:          IN STD_LOGIC;
            carry_in:     IN STD_LOGIC;
            in_A:         IN STD_LOGIC_VECTOR(31 DOWNTO 0);
            in_B:         IN STD_LOGIC_VECTOR(31 DOWNTO 0);
            opcode:       IN STD_LOGIC_VECTOR(2 DOWNTO 0);
            inject:             IN STD_LOGIC_VECTOR(31 DOWNTO 0);
            result:       OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
            carry_out:    OUT STD_LOGIC;
            alu_c_out:    OUT STD_LOGIC_VECTOR (5 DOWNTO 0);
            bcp_c_out:    OUT STD_LOGIC_VECTOR (5 DOWNTO 0);
            error_out:   OUT STD_LOGIC
);

END sc_alu;

ARCHITECTURE rtl OF sc_alu IS

SIGNAL carry:                   STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL bcp_c:               STD_LOGIC_VECTOR(5 DOWNTO 0);
SIGNAL alu_c:               STD_LOGIC_VECTOR(5 DOWNTO 0);
SIGNAL carry_out_stage2:    STD_LOGIC;
SIGNAL in_A_stage2p:        STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL in_B_stage2p:        STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL opcode_stage2p:      STD_LOGIC_VECTOR(2 DOWNTO 0);
```

```
SIGNAL carry_in_stage2p: STD_LOGIC;
SIGNAL carry_out_stage3p: STD_LOGIC;
SIGNAL result_stage2p:     STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL in_A_stage2n:       STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL in_B_stage2n:       STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL carry_in_stage2n: STD_LOGIC;
SIGNAL carry_out_stage3n: STD_LOGIC;
SIGNAL opcode_stage2n:     STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL result_stage2n:     STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL result_stage3p:     STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL result_stage3n:     STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL in_A_stage2:        STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL in_B_stage2:        STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL opcode_stage2:      STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL result_stage2:      STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL carry_in_stage2:    STD_LOGIC;
SIGNAL in_A_stage2_in:     STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL in_B_stage2_in:     STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL mux_sel:            STD_LOGIC;
SIGNAL mux_sel_latch:      STD_LOGIC;

COMPONENT alu IS
PORT(
             in_A:         IN STD_LOGIC_VECTOR(31 DOWNTO 0);
             in_B:         IN STD_LOGIC_VECTOR(31 DOWNTO 0);
             opcode:       IN STD_LOGIC_VECTOR(2 DOWNTO 0);
             carry_in:     IN STD_LOGIC;
             inject:              IN STD_LOGIC_VECTOR(31 DOWNTO 0);
             result:       OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
             carry_out:    OUT STD_LOGIC;
             carry:               OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
);
END COMPONENT alu;

COMPONENT bcp IS
PORT(
             in_A:         IN STD_LOGIC_VECTOR(31 DOWNTO 0);
             in_B:         IN STD_LOGIC_VECTOR(31 DOWNTO 0);
             in_C:         IN STD_LOGIC_VECTOR(31 DOWNTO 0);
             inject:              IN STD_LOGIC_VECTOR(31 DOWNTO 0);
             carry_in:    IN STD_LOGIC;
             carry_out:   IN STD_LOGIC;
             opcode:       IN STD_LOGIC_VECTOR(2 DOWNTO 0);
             zA_out:              OUT STD_LOGIC_VECTOR (5 DOWNTO 0);
             zB_out:              OUT STD_LOGIC_VECTOR (5 DOWNTO 0);
             zC_out:              OUT STD_LOGIC_VECTOR (5 DOWNTO 0);
             result_c:     OUT STD_LOGIC_VECTOR(5 DOWNTO 0)
);
END COMPONENT bcp;


BEGIN

       ALU1: alu
       PORT MAP(
       --into ALU
             in_A => (in_A_stage2(31) XOR inject(1)) & in_A_stage2(30 DOWNTO 1)
& (in_A_stage2(0) XOR inject(0)),
             in_B => in_B_stage2(31 DOWNTO 15) & (in_B_stage2(14) XOR
inject(2)) & in_B_stage2(13 DOWNTO 7) & (in_B_stage2(6) XOR inject(3)) &
in_B_stage2(5 DOWNTO 1) & (in_B_stage2(0) XOR inject(31)),
             opcode => opcode_stage2(2) & (opcode_stage2(1) XOR inject(8)) &
opcode_stage2(0),
             carry_in => carry_in_stage2,
             inject => inject,
       --out of ALU
             result => result_stage2,
             carry_out => carry_out_stage2,
             carry => carry
       );

       BCP1: bcp
```

```
	PORT MAP(
	--into BCP
		in_A => in_B_stage2(31 DOWNTO 29) & (in_B_stage2(28) XOR
inject(4)) & in_B_stage2(27 DOWNTO 1) & (in_B_stage2(0) XOR inject(5)),
		in_B => in_B_stage2(31 DOWNTO 22) & (in_B_stage2(21) XOR
inject(6)) & in_B_stage2(20 DOWNTO 9) & (in_B_stage2(8) XOR inject(7)) &
in_B_stage2(7 DOWNTO 0),
		in_C => carry(31 DOWNTO 15) & (carry(14) XOR inject(10)) &
carry(13 DOWNTO 0),
		inject => inject,
		carry_out => carry_out_stage2 XOR inject(11),
		carry_in => carry_in_stage2 XOR inject(13),
		opcode => (opcode_stage2(2) XOR inject(20)) & opcode_stage2(1
DOWNTO 0),
	--out of BCP
		result_c => bcp_c
	);

	--positive edge of clock
	PROCESS(clk, in_A, in_B, opcode)
	BEGIN
		IF(RISING_EDGE(clk)) THEN
			in_A_stage2p <= in_A;
			in_B_stage2p <= in_B;
			carry_in_stage2p <= carry_in;
			opcode_stage2p <= opcode;
			if (mux_sel_latch = '1') then
				result <= result_stage3p;
				carry_out <= carry_out_stage3p;
			else
				result <= result_stage3n;
				carry_out <= carry_out_stage3n;
			end if;

		END IF;
	END PROCESS;

	--negative edge of clock
	PROCESS(clk, in_A, in_B, opcode)
	BEGIN
		IF(FALLING_EDGE(clk)) THEN
			in_A_stage2n <= in_A;
			in_B_stage2n <= in_B;
			carry_in_stage2n <= carry_in;
			opcode_stage2n <= opcode;
			mux_sel_latch <= mux_sel;

		END IF;
	END PROCESS;

	alu_c_out <=alu_c;
	bcp_c_out <=bcp_c;

	mux_sel <= '1' WHEN ((alu_c(5 DOWNTO 1)& (alu_c(0) XOR inject(14))) =
((bcp_c(5) XOR inject(16)) & bcp_c(4 DOWNTO 3)& (bcp_c(2) XOR inject(15))&
bcp_c(1 DOWNTO 0))) AND (clk = '1') ELSE '0';
	error_out <= mux_sel;

	result_stage3p <= result_stage2 WHEN clk='1';
	carry_out_stage3p <= carry_out_stage2 WHEN clk='1';

	result_stage3n <= result_stage2 WHEN clk='0';
	carry_out_stage3n <= carry_out_stage2 WHEN clk='0';

	WITH clk SELECT
		in_A_stage2 <= in_A_stage2p WHEN '1',
					in_A_stage2n WHEN OTHERS;

	WITH clk SELECT
		in_B_stage2 <= in_B_stage2p WHEN '1',
					in_B_stage2n WHEN OTHERS;
```

```
	WITH clk SELECT
		opcode_stage2 <= opcode_stage2p WHEN '1',
					opcode_stage2n WHEN OTHERS;

	WITH clk SELECT
		carry_in_stage2 <= carry_in_stage2p WHEN '1',
					carry_in_stage2n WHEN OTHERS;

	--zeros counter

	alu_c <= ("00000"&NOT(result_stage2(0))) +
("00000"&NOT(result_stage2(1))) + ("00000"&NOT(result_stage2(2)))+
("00000"&NOT(result_stage2(3)))+
			("00000"&NOT(result_stage2(4))) +
("00000"&NOT(result_stage2(5)))+ ("00000"&NOT(result_stage2(6)))+
("00000"&NOT(result_stage2(7)))+
			("00000"&NOT(result_stage2(8))) +
("00000"&NOT(result_stage2(9)))+ ("00000"&NOT(result_stage2(10)))+
("00000"&NOT(result_stage2(11)))+
			("00000"&NOT(result_stage2(12) XOR inject(12))) +
("00000"&NOT(result_stage2(13)))+ ("00000"&NOT(result_stage2(14)))+
("00000"&NOT(result_stage2(15)))+
			("00000"&NOT(result_stage2(16))) +
("00000"&NOT(result_stage2(17) XOR inject(17)))+
("00000"&NOT(result_stage2(18)))+ ("00000"&NOT(result_stage2(19)))+
			("00000"&NOT(result_stage2(20))) +
("00000"&NOT(result_stage2(21)))+ ("00000"&NOT(result_stage2(22)))+
("00000"&NOT(result_stage2(23)))+
			("00000"&NOT(result_stage2(24))) +
("00000"&NOT(result_stage2(25)))+ ("00000"&NOT(result_stage2(26)))+
("00000"&NOT(result_stage2(27) XOR inject(27)))+
			("00000"&NOT(result_stage2(28))) +
("00000"&NOT(result_stage2(29) XOR inject(29)))+ ("00000"&NOT(result_stage2(30)
XOR inject(30)))+ ("00000"&NOT(result_stage2(31)));
END rtl;
```