Abstract

A duplicate system has two identical pieces of hardware which perform the same computation in parallel and then compare the results; if they disagree, an error message is generated. The duplicate system cannot tolerate faults but can only detect them. There is no method of determining which unit is faulty. However, a duplicate system can mask faults if the voter knows which unit is faulty.

We implemented an AN code protected duplicate ALU system. We injected arbitrary faults and measured the fault detection capability. We found a 90% improvement in our duplex system over the simplex system.

1. Introduction

One way to provide uninterrupted reliable computation is to use a TMR (three modular redundancy) configuration in which a majority voter decides the answer. It is possible to use less hardware to provide similar reliability if the faulty unit is known. We propose to study a system which will be able to detect the error and provide correct data in case of a faulty unit.

2. Background

In any system with modular redundancy (duplicate, TMR, nMR), the reliability of the system can be increased if it is immediately evident that the data coming from one unit has errors. For example, a normal TMR system needs to have at least two units providing the same answer to provide the correct result. If the data coming from each unit could be interpreted to determine if it were in error, then only one unit would have to give the correct result for the voter to decide the correct result. One way to indicate the "data is in error" is to use codewords to encode the data.

Error correcting codes are used extensively to protect data in DRAM, disks, and networks. The most common are the SEC-DED codes, which can correct any single bit error and detect two bit errors. There are also some other types like SEC-SBD, SEC-DED-SBD, etc. However, the limitation of these general types of codes is that to do some arithmetic operation, the data must be separated from the codeword. After the arithmetic operation, the data must be encoded into a codeword again. Thus the code does not protect the data as the arithmetic operation is executed. An error in the adder, for example, would not produce an invalid codeword because the incorrectly-added result was encoded into a valid codeword after addition.

A class of codes called Arithmetic Codes exist which protect data through certain arithmetic operations. Thus the entire data-path can be protected from errors. These errors will be immediately apparent by inspecting the resulting codeword.

3. Arithmetic Codes

Arithmetic Codes are very useful when it is desired to check arithmetic operations such as addition, subtraction, multiplication, and division. The data presented to the arithmetic operation is encoded before the operations are performed. After completing the arithmetic operations, the resulting codewords are checked to make sure that they are valid codewords. If the resulting codewords are not valid, an error condition is signalled.

An Arithmetic Code must be an invariant to a set of arithmetic operations. An arithmetic code, f(), has the property that $f(b*c) = f(b)*f(c)$, where b and c are operands, * is some arithmetic operation, and f(b) and f(c) are the arithmetic codewords for the operands b and c respectively. To completely define an Arithmetic Code, the method of encoding and the arithmetic operations for which the code is invariant must be specified. The most common examples of the arithmetic codes are the AN codes, residue codes, and inverse residue codes.

The simplest Arithmetic Code is the AN code which is formed by multiplying each data word by some constant, A. It should be mentioned here that the AN codes are invariant to addition and subtraction but not to multiplication or division. If N and M are two operands to be encoded, the resulting codewords will be A*N and A*M, respectively. If the two codewords are added, the sum is (A*N+A*M), which is the codeword of the correct sum. The operations performed under an AN code can be checked by determining if the results are evenly divisible by the constant A. If not, an error condition is signalled.

The magnitude of the constant, A, determines both the number of extra bits required to represent the codewords and the error detection capabilities provided. The selection of the constant, A, is critical to the effectiveness and the efficiency of the resulting code. First, for binary codes, the constant must not be a power of two. To see the reason for this limitation, suppose that we encode the binary number (a_{n-1} a_{n-2} $a_{n-3}a_{n-4}$ … … … a_2 a_1 a_0) by multiplying by constant $A = 2^a$. Multiplication by 2^a is equivalent to left arithmetic shift of the original binary word, so the resulting codeword will be (a_{n-1} a_{n-2} $a_{n-3}a_{n-4}$ … … … a_2 a_1 a_0 0 0 … … 0) where a 0s have been appended to the original binary number. The decimal representation of the codeword is given by

$$a_{n-1}2^{a+n-1} \ldots \ldots + a_2 2^{a+2} + a_1 2^{a+1} + a_0 2^a + 0*2^{a-1} + \ldots \ldots \ldots + 0*2^1 + 0*2^0$$

which is evenly divisible by 2^a. It is also easy to see, however, that changing just one coefficient will still yield a result that is evenly divisible by 2^a. For example, if the coefficient of the 2^a term changes from 0 to 1, the result will remain evenly divisible by 2^a. Thus AN code that has $A = 2^a$ is not capable of detecting single bit errors.

An example of an AN code is the 3N code where all words are encoded by multiplying by a factor of 3. If the original data words are n-bits in length, the codewords for the 3N code will require n+2 bits. The encoding of the operands in the 3N code can be performed by a simple addition if we can recognize that we can multiply any number by 3 by adding the original number to a value that is twice that number. In other words, we form 3N by adding N and 2N. In general, the value of A should be relatively a prime number.

A 3N code can be used to detect single bit errors because no words with hamming distance one are divisible by three. A valid codeword with a single-bit error can be represented by $n*3 \pm 2^n$, where n is the bit in error. This is not evenly divisible by three because 2^n mod 3 is not zero.

4. Objective

Our objective is to build a AN protected ALU system, to inject faults into the system, and to measure the fault detection capabilities.

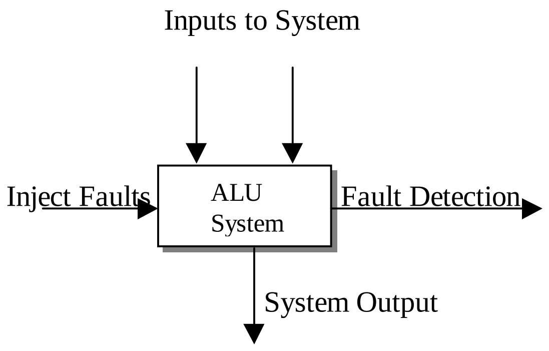


Figure 1: System Model

5. ALU System Design and Implementation

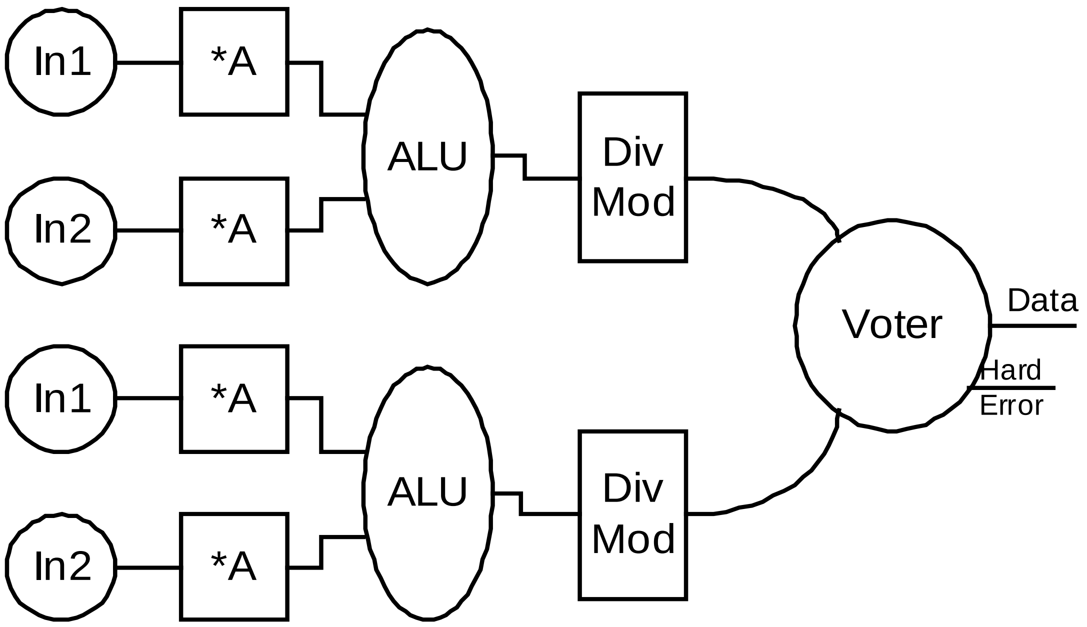


Figure 2: Overall System Implementation

We implemented a duplicate ALU system with a voter. The prime responsibility of the voter was to detect the error and to select the correct output. The voter output was compared with the "Golden Adder" (which generates the true sum of the two inputs) output to see whether the voter output was correct or not (for statistics). Our entire design was fully pipelined.

5.1 Sub-Modules

5.1.1 Input and Codeword Generation

The inputs are expecting two eight bit words from the external world which are stored in the first stage pipeline and fed into the codeword generation block. As we have previously discussed, we are implementing an Arithmetic Code and we selected the value of A=3. This multiply was implemented using a simple eight-bit adder, i.e. N + (N + N). The final adder output after the codeword generation is ten bits. The coded input is stored in the second stage pipeline registers. Figure 3 describes the codeword generator block.

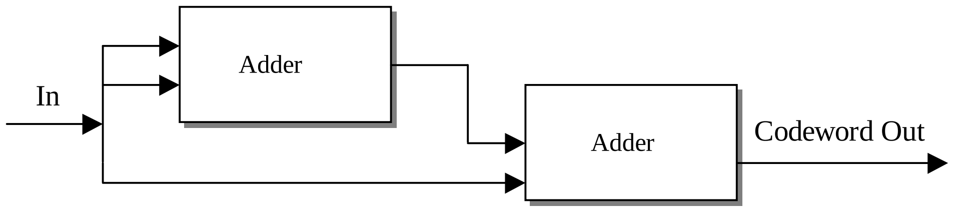


Figure 3: Codeword Generation

5.1.2 Arithmetic Operation

We restricted our arithmetic operation to addition only for simplicity. The inputs to the adder come from the coded input out of the codeword generator block and the adder output is captured in the third stage pipeline register.

5.1.3 Quotient and Remainder Generation

After the coded ALU output is generated, it needs to be divided by the constant A=3 to get the quotient and the remainder. For the division, we followed a very simple technique (Figure 4). After the remainder and the quotient are generated, the remainder is checked to see whether it is non-zero. In case of a non-zero remainder an error flag is generated. Depending on the two error signals from two ALUs, the voter selects the correct quotient. After the final quotient has been selected, it is compared with the golden adder output to get the statistics (Figure 5).

5.1.4. Flags and Counters

The remainder is checked for non-zero value and the quotient is checked with the golden adder output for the correct result. These give us two different flags: Data_Corrupt (compared to the true value) and Error_Detectetd (for non-zero remainder). From these, we were able to derive four different cases of flag combinations and four different counters were used to keep track of the case statistics.

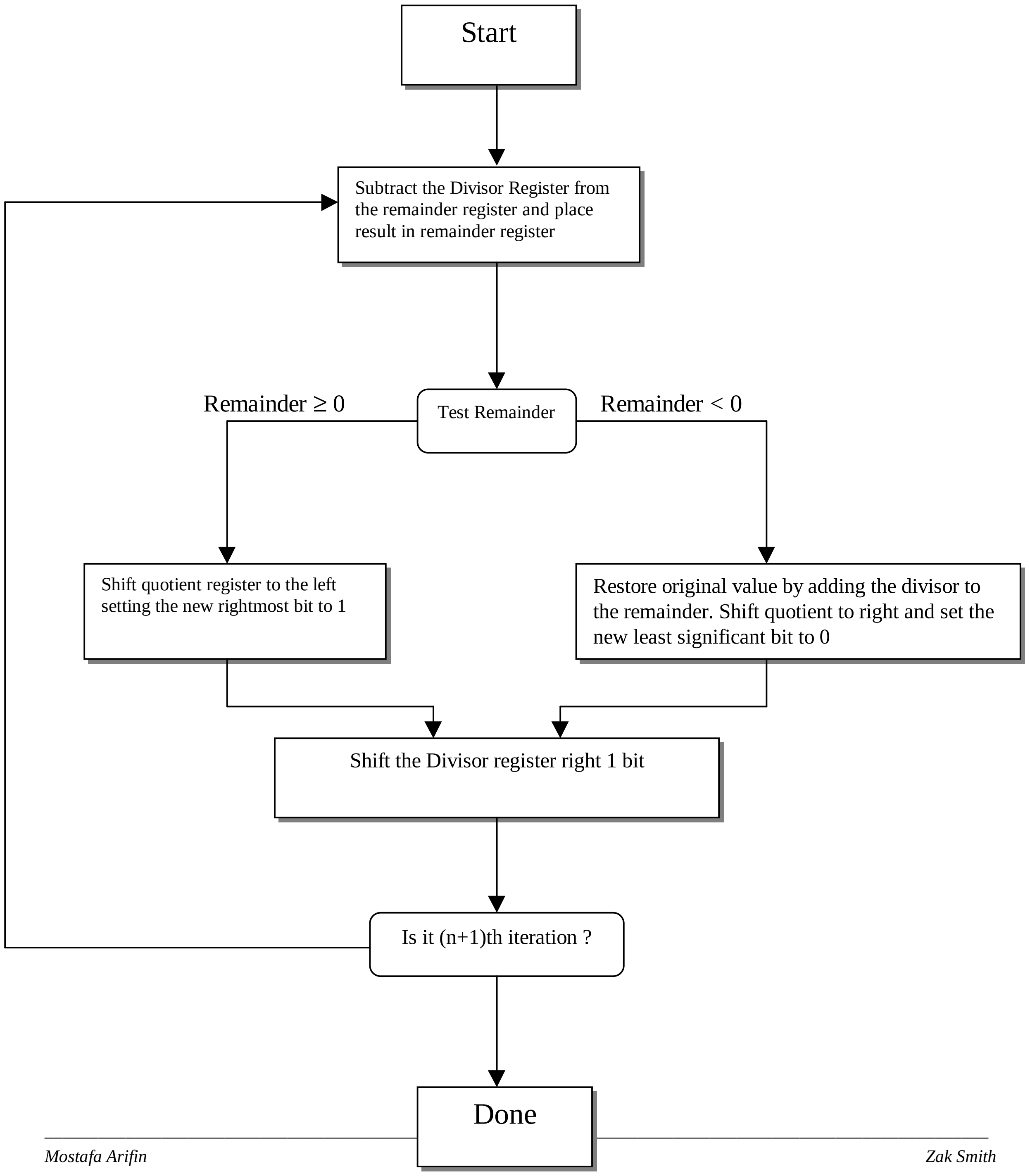
Case Name	Shortened Name	Case Description
Data_Ok, No Error Detected	DO, NE	Fault had no effect on output
Data_Ok, Error Detected	DO, E	Error Detected, Data not affected
Data_Not_Ok, Error Detected	DNO, E	Fault affected output and Detected
Data_Not_Ok, No Error Detected	DNO, NE	Fault affected output but not detected

Table 1: Case Description

6. Implementation Technique

For the overall design of our schematic, we used Mentor pld_da (design architect), and for functional verification, we used Mentor pld_quicksim. Our original target of this project was to implement this in hardware with the help of Xilinx XC40006E FPGAs, but working FPGAs were not available. As a result, we had to focus on simulations only.

Figure 4: First Iteration of Division Algorithm (not an optimized algorithm)


Start
Subtract the Divisor Register from the remainder register and place result in remainder register
Remainder ≥ 0
Test Remainder
Remainder < 0
Shift quotient register to the left setting the new rightmost bit to 1
Restore original value by adding the divisor to the remainder. Shift quotient to right and set the new least significant bit to 0
Shift the Divisor register right 1 bit
Is it (n+1)th iteration ?
Done

7. Simulation Methodology

We injected arbitrary faults in the circuit and then simulated the schematic with the input data set. Since we are dealing with two eight bit input data sets, there are 2^{16} possible input combinations. But more importantly, simulations of 2^{16} inputs would take about 45 minutes. As a result, we limited our input combinations such that the both the inputs are divisible by seven which gave us 361 different combinations. At the end of each run, we recorded the values of the case counters. For the next result, we picked a different fault at a different place and repeated the same procedure.

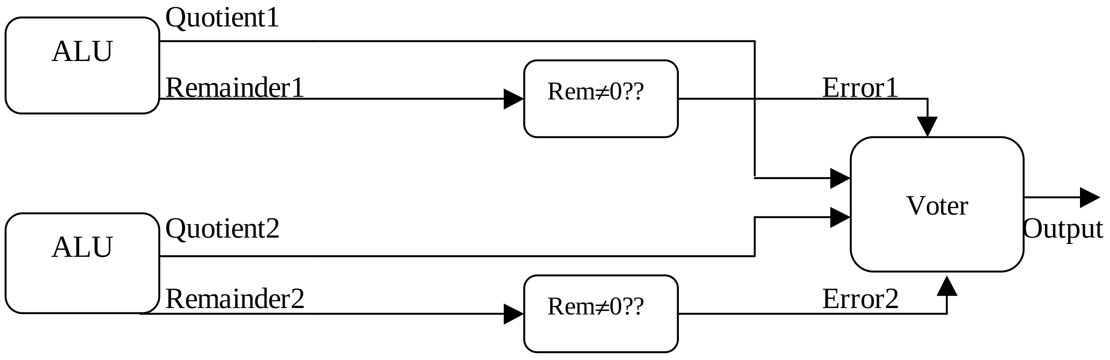


Figure 5: Error Detection and Correct Quotient Selection

8. Fault Models

We injected four different types of fault models: SSA (single stuck-at), LGS (logic gate substitution), Bridging (BD), and Bizarre (BZ). The following table describes the definitions of each fault model with respect to our schematic and the number of faults injected.

(Note: not just injected in core – voter/divider/encoder not assumed to be fault free)

Fault Models	Definition	Number of faults injected
SSA (single stuck-at)	One of the wires in the bus is always either connected to VCC or to GND	15
LGS (logic gate substitution)	One of the gates is changed to other gate. Example: an AND gate becoming a NAND	7
BD(bridging)	Two or three wires of a bus are connected to each other	5
BZ(bizarre)	Random bits (say bit 2, 3, 5, 7) are interchanged	2

Table 2: Fault Models and Definitions

9. Results

9.1.1 SSA Fault Model Statistics

Iteration Number	DO, NE	DO, E	DNO, E	DNO, NE
1	77	0	284	0
2	181	0	180	0
3	361	0	0	0
4	0	361	0	0
5	2	0	0	359
6	176	0	185	0
7	0	361	0	0
8	180	0	181	0
9	100	0	0	261
10	82	0	0	279
11	82	0	0	279
12	361	0	0	0
13	190	0	0	171
14	361	0	0	0
15	0	361	0	0
Total	2153	1083	830	1349

Table 3: SSA fault model statistics

9.1.2 Graphical Representation

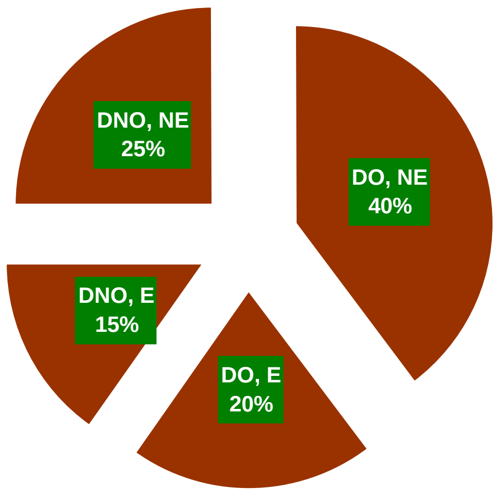


Figure 6: SSA Fault Model Statistics

9.1.3 SSA Fault Model Analysis

1. Most of the injected faults caused no errors (⇒ 2153/(15*361) = 39.8%)
2. All undetected faults (⇒ 1349/(15*361) = 24.9%) caused by either
   i. Faulty Inputs
   ii. Faulty divider or faulty voter
3. Data errors detected (⇒ 830/(15*361) = 15.3%)
4. Erroneous error detection (⇒ 1083/(15*361) = 20%)
5. Safety = (# Fault not excited + # Error detected) / (#Total Input set)
   = (2153 + 1083 + 830)/(15*361) = 75%

9.2.1 LGS Fault Model Statistics

Iteration Number	DO, NE	DO, E	DNO, E	DNO, NE
1	0	361	0	0
2	0	0	0	361
3	361	0	0	0
4	262	0	99	0
5	104	76	181	0
6	81	100	180	0
7	213	0	148	0
Total	1021	537	608	361

Table 4: LGS fault model statistics

9.2.2 Graphical Representation

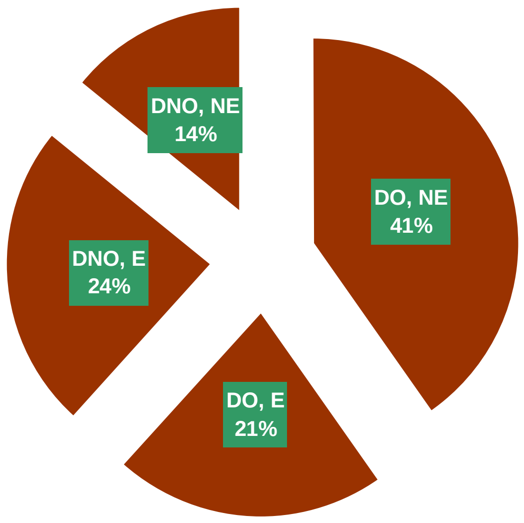


Figure 7: LGS Fault Model Statistics

9.2.3 LGS Fault Model Analysis

1. Most of the injected faults caused no errors (⇒ 1021/(7*361) = 40.4%)
2. All undetected faults (⇒ 361/(7*361) = 14.29%) caused by
   i. Faulty Inputs
   ii. Faulty divider or voter
3. Data Errors detected (⇒ 608/(7*361) = 24.1%)
4. Erroneous error detection (⇒ 537/(7*361) = 21.25%)
5. Safety = (# Fault not excited + # Error detected) / (#Total Input set)
   = (1021 + 537 + 608)/(7*361) = 85.71%

9.3.1 BD Fault Model Statistics

Iteration Number	DO, NE	DO, E	DNO, E	DNO, NE
1	82	0	0	279
2	172	0	189	0
3	188	0	173	0
4	180	91	90	0
5	180	90	91	0
Total	802	181	543	279

Table 5: BD Fault Model Statistics

9.3.2 Graphical Representation

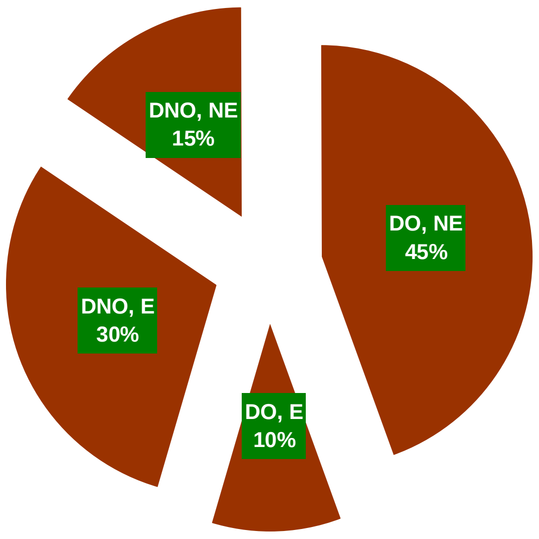


Figure 8: BD Fault Model Statistics

9.3.3 BD Fault Model Analysis

1. Most of the injected faults caused no errors (⇒ 802/(5*361) = 44.4%)
2. All undetected faults (⇒ 279/(5*361) = 15.46%) caused by
    i. Faulty Inputs
    ii. Faulty divider or voter
3. Data Errors detected (⇒ 543/(7*361) = 30.08%)
4. Erroneous error detection (⇒ 181/(5*361) = 10.03%)
5. Safety = (# Fault not excited + # Error detected) / (#Total Input set)
   = (802 + 181 + 543)/(5*361) = 84.54%

9.4.1 BZ Fault Model Statistics

Iteration Number	DO, NE	DO, E	DNO, E	DNO, NE
1	38	0	13	310
2	38	0	13	310
Total	76	0	26	620

Table 6: BZ Fault Model Statistics

9.4.2 Graphical Representation

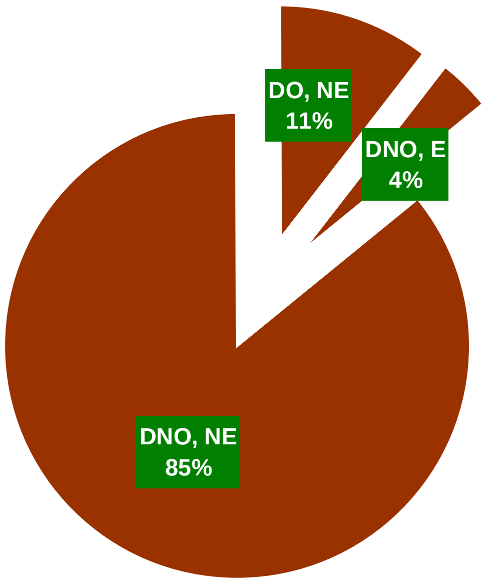


Figure 9: BZ Fault Model Statistics

9.4.3 BZ Fault Model Analysis

1. Most of the injected faults caused undetected errors (⇒ 620/(2*361) = 85.87%) because bits scrambled were internally consistent, but not OK compared to the outside world
2. Data OK, No Error Detected (⇒ 76/(2*361) = 10.53%) caused by
3. Data Errors detected (⇒ 26/(2*361) = 10.53%)
4. No erroneous error detection (⇒ 0/(2*361) = 0.0%)
5. Safety = (# Fault not excited + # Error detected) / (#Total Input set)
   = (76 + 26)/(2*361) = 14.13%

9.5.1 Overall Fault Model Statistics

Fault Model Name	DO, NE	DO, E	DNO, E	DNO, NE
SSA	2153	1083	830	1349
LGS	1021	537	608	361
BD	802	181	543	279
BZ	76	0	26	620
Total	4052	1801	2007	620

Table 6: Overall Fault Model Statistics

9.5.2 Graphical Representation

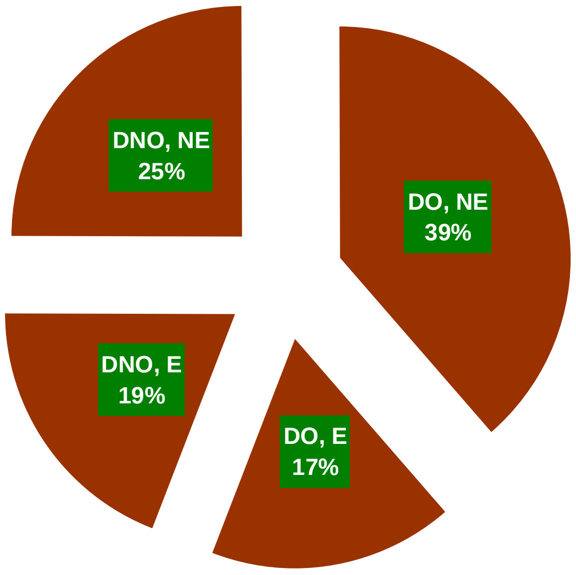


Figure 10: Total Fault Model Statistics

9.5.3 Overall Fault Model Analysis

1. Most of the injected faults caused no errors (⇒ 4052/(29*361) = 38.7%)
2. All undetected faults (⇒ 2609/(29*361) = 24.92%) caused by
    - i. Faulty Inputs
    - ii. Faulty divider or voter
2. Data Errors detected (⇒ 2007/(29*361) = 19.17%)
3. Erroneous error detection (⇒ 1801/(29*361) = 17.2%)
4. Safety = (# Fault not excited + # Error detected) / (#Total Input set)
   = (4052+1801+2007)/(29*361) = 75.08%

(note: the above statistics were weighted with the number of injected faults)

10. Results Summary

The voter can decide correctly if and only if the faulty unit signals error, or the data was ok. From the overall fault model statistics, we found out that the probability of a duplex system working with one fault is 75.1%, where as the probability of a simplex system working with one fault is 38.7%. So from these we can conclude that we get about 90% improvement over the simplex system.

11. Future Work

For future work, we could build TMR system and evaluate performance with similar fault injection procedure. Secondly, we could compare the performance of plain TMR system with our AN code protected duplex system. Last of all, we could be able to compare hardware costs of the two systems.

Reference:

[raofuj:89] T. R. N. Rao, E. Fujiwara. “Error-Control Coding from Computer Systems”, Englewood Cliffs, NJ. Prentice Hall, 1996

[wake:78] J. Wakerly, “Error Detecting Codes, Self-Checking Circuits, and Applications”, New York, NY. Elsevier North-Holland, 1978

[petwel:72] W. W. Peterson, E. J. Weldon, JR. “Error-Correcting Codes” 2nd Edition, Cambridge, MA. MIT Press 1972

[prad:96] D. K. Pradhan, editor, “Fault-Tolerant Computer System Design” 1st Edition, Prentice Hall, 1996

[henmann: 68] Henry. B. Mann, editor, “Error-Correcting Codes”, proceedings of a symposium, New York, Wiley 1968

[gossgraf:93] Michael Gossel, Steffen Graf, “Error-Detection Circuits” New York, NY. McGraw-Hill 1993

[patthenn:98] D. Patterson, J. Hennessy, “Computer Organization and Desgin: The Hardware/Software Interface” Morgan Jaufamann Publishers Inc., 2nd Edition, 1998;

Appendix:

Mentor Schematics in order:

1. Overall Duplicate ALU Schematic with Voter
2. Single ALU Unit
3. Codeword Generator
4. Division Unit
5. A=3 Block