Robustness of Numerical Representations

ABSTRACT

Memory devices are used as storage for integers and rational numbers in various binary representations such as one’s complement, two’s complement, and floating-point. In theory, these devices should be able to store numbers accurately, but manufacturing variations and environmental noise can inadvertently cause bitwise errors in the form of flipped or stuck bits. As transistors and other semiconductor materials continue to shrink into the nano scale, variations exert a larger and larger impact on chip performance, causing errors more often. The goal of this work is to analyze the error-tolerance of different numerical representations. We first model the effect that different bitwise faults induce within each distinct numerical representation. Then, we compare the robustness of these representations for each fault type. Using Monte Carlo simulations, we show that an error in the floating-point representation is significantly more harmful than corresponding errors in the one's complement or two's complement systems. Furthermore, we propose a novel optimization for the representation of floating-point numbers and demonstrate its unique benefits. With an error tolerant numerical storage system, a greater percentage of products that do not meet reliability requirements can be released into the market, thus increasing production efficiency, profit margins, and performance stability.

EXECUTIVE SUMMARY

The general trend in technology thus far has been to synthesize electrical components on a smaller scale. As devices such as memory chips continue to shrink and approach the nano-scale, manufacturing variations and other slight differences are more likely to cause inconsistent chip performance. Flash memory functions by storing numbers in various binary representations, including one's complement, two's complement, and floating-point. Although in theory, memory devices should be able to store numbers accurately, the aforementioned hardware variations cause bitwise errors in the form of flipped or stuck bits. A bitwise error occurs when a bit, or binary digit, in the sequence of a binary number, "flips" (changes from a 1 to a 0 or vice versa) or gets "stuck" (when a bit remains a 1 or 0). The severity of these errors depends not only on the binary representation employed but also on the location of the flipped or stuck bit. For this reason, a multi-staged approach is necessary to model and analyze the varying accuracy levels. First, utilizing the programming language, Python, we wrote and ran a code that simulated the effect that different bitwise faults induce within each numerical representation. We then compared the error results obtained from each of these representations for each fault type. From the results of the tests, we found that one's and two's complement were the most error tolerant. Thus, we created a new version of floating point that incorporated two's complement and that would therefore be more error tolerant. After running this new form of floating point, we discovered that it was more error tolerant than the original floating point form. This novel technique has broad implications. In today's world, countless defective chips are discarded because they do not meet the manufacturers specifications. With an error-tolerant numerical storage system, a greater percentage of products can be released into the market, thus greatly increasing production efficiency and profit margins.

Research Report

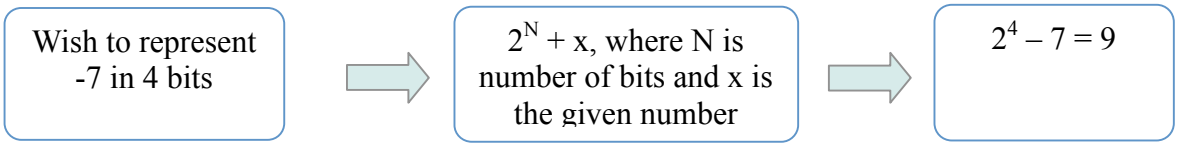
I. INTRODUCTION

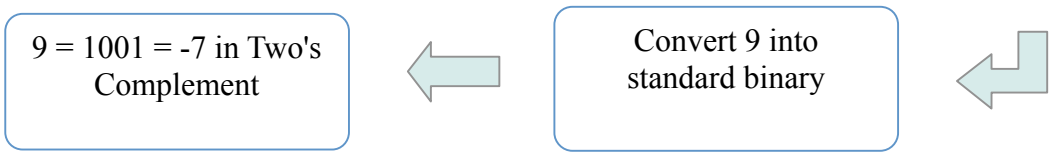
1.1 – Background

Prior to a discussion of our research, we must first introduce the concept of numerical representations. As is widely known, computers store alphanumerical values using a system called binary. Each binary digit is called a 'bit' and can only have a value of 0 or 1. In the same way as the decimal system is a base-10 system, the binary representation is a base-2 system. Thus, in the decimal system, $110 = (1 \times 10^2) + (1 \times 10^1) + (0 \times 10^0)$, but in binary, $110 = (1 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) = 6$.

The general form of binary is widely termed 'unsigned binary' because there is no ingrained method to portray negative numbers. To that effect, when computing was a relatively new concept, many different methods (called numerical representations) were suggested. Our research is centered on three of the most widely used signed number representations (one's complement, two's complement, and floating-point):

1. One's Complement
   - positive numbers are represented in the standard binary form
   - negative numbers are formed by inverting all the bits to form the 'complement' of a positive
2. Two's Complement
   - positive numbers are represented in the standard binary form
   - negative numbers are obtained by inverting all the bits of a positive number and adding one to the subsequent binary number; however, a generally accepted shortcut also exists:





3. Floating-Point
   - 1st part - first bit is a sign bit: 0 for positive numbers and 1 for negative numbers
   - 2nd part - following eight bits are unsigned exponent bits
     - in order to portray small values (on the order of 2^{-100}), a bias of -127 is added
     - using 8 unsigned bits, 256 values can be represented
     - with the bias, values from -127 to 128 can be portrayed in the exponent
   - 3rd part – next 23 bits are the significand (number without exponent or sign added to it)
   - complete format = (sign) x (signficand) x $2^{exponent}$

1.2 – Previous Work and Our Research

Moore's law dictates that the number of transistors that can be placed on a chip of a constant size doubles every one to two years [1]. The inevitable result of this trend is increased performance within a smaller size. We can see this pattern in mass market applications such as laptops or mobile phones: the general trend in technology thus far has been to synthesize electrical components on a smaller scale. As devices such as memory chips continue to decrease in size (current measurement is 22 nanometers) [2], process variations (i.e. fluctuations in dopant molecules) are likely to cause significant inconsistencies in chip performance, meaning that many of these chips will not meet the manufacturing specifications they were designed for [3]. An inevitable result of this phenomenon is the decrease of raw manufacturing yield. Error-tolerant, or stochastic, computing is a novel way to deal with the problem. Under this computing paradigm, errors that do not compromise the accuracy of the task significantly are allowed to happen [4]. Significant research has been done to explore the viability of stochastic computing [5][6], but, to our best knowledge, none explore the error-tolerance of numerical representations.

The work in [5] demonstrates the fault density (the number of bit errors per unit area) in flash memory that will produce acceptable audio quality by inducing errors in the encoded audio bit stream, and [6] proposes a scheme to maximize information density (the number of information bits per unit area) in cases when the probability of a bit-cell failure in SRAM is given.

There has been some work, however, on the subject of optimizing bit streams to obtain the maximum accuracy, or information storage [7]. The focus of the tests, though, was not on the comparison of numerical representations or on the optimization of a specific numerical representation to increase accuracy. The authors in [7] designed a system to explore the trade-off between quantization noise (rounding or truncation error), fixed by ascribing more data bits to store a number, and bit cell failure (flipped or stuck bits), fixed by reducing the number of data bits and adding redundancy bits so that accuracy is preserved.

Our research contributes unique and relevant information since the numerical representations that we study are ubiquitous in memory devices. In these devices, the process variations mentioned previously manifest themselves as errors in number storage. This means that the value of a bit at a certain location will change if an error occurs. Our project works to quantize that change, measuring how much effect an error can have and classifying which numerical representation best handles error. If we were to enter an era of stochastic computing, efforts would have to be made not only to identify the amount of errors that can result in acceptable performance but also to increase the level of tolerance (allowing more errors while still maintaining acceptable performance). The primary concern in the creation of the original numerical representations was storing the greatest amount of information possible in the least number of bits. Now, the situation has changed. Variability in number storage is quickly becoming a major issue, shifting the perspective from storage capacity to error-tolerance. Our

work specifically addresses this shift in perspective by first characterizing errors in existing numerical representations and subsequently exploring methods to increase error-tolerance.

II. MATERIALS AND METHODS

2.1 – Background

Prior to initializing tests on the various binary systems, we researched the most commonly used numerical representations in memory devices: one's complement, two's complement, and floating-point. In doing so, we studied how numbers are translated from decimal form to the binary form for each representation, a process used by computers to store the numbers in memory chips. Subsequently, utilizing the scripting language, Python, we wrote programs to convert decimal numbers into each of the four representation forms.

2.2 - Methodology

The goal of the Python programs was to induce errors in numbers of each binary representation. We operated under two main error models and a total of three error types:

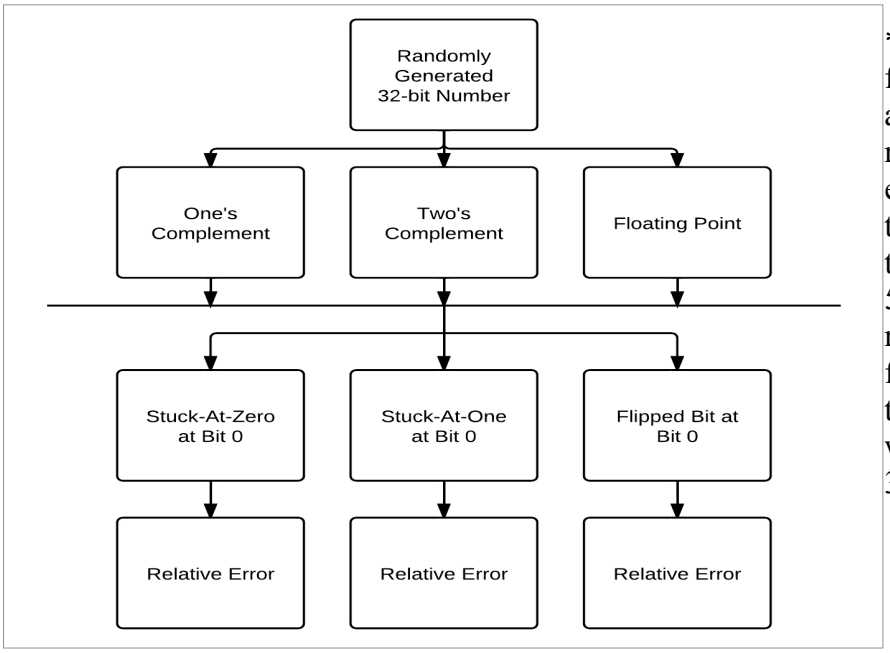
- Stuck At Fault Model – emulates the unchanging value of a bit when a wire is disconnected in a transistor
  - stuck at zero (the bit remains "stuck" at 0; if the bit is 1, it switches to a 0)
  - stuck at one (the bit remains "stuck" at 1; if the bit is 0, it switches to a 1)
- Soft Error Model – simulates a bit flip when a charged particle interacts with a transistor, altering the voltage and thus the value of the bit register
  - flipped bit (where the bit "flips" from a 0 to 1 or a 1 to 0)
  - the flipped bit error only occurs when the collected charge of a charged particle, Q_{coll}, is greater than the critical charge, Q_{crit}
  - critical charge is designated by the manufacturer and is different for every type of chip

32-bit binary values were used for all of the different representations to ensure a standardized error distribution. The method for measuring errors was as follows: For each of the

32-bit numbers, we calculated the original decimal value, separately induced errors at each of the 32 possible bit locations, calculated the resulting decimal value (number with error), and computed the resulting relative (or percent) error. These relative errors were then compared both across representations and across error types.

2.3 – Monte Carlo Simulation

The range of 32-bit numbers that can be represented reaches into the area of four billion possible numbers (2^{32}). This range of binary numbers is much too large for us to test every possibility in a realistic time period. Thus, we implemented a Monte Carlo simulation in our code. A Monte Carlo simulation utilizes repeated random sampling from a domain to approximate the actual outcome of a function on a set of numbers [8]. To ensure the accuracy of our Monte Carlo samples, we tested five million numbers for each of the 32 possible error locations and repeated the simulation five separate times at each error location as well. A flow chart to illustrate the method (for errors at bit 0) follows:



*The actual method differs from the flow chart. For accuracy, we used 5 million random numbers for each error location and averaged the errors that resulted. We then repeated the simulation 5 times and averaged the 5 relative errors to acquire the final error percentage. A total of 25 million numbers was tested for each of the 32 possible error locations.

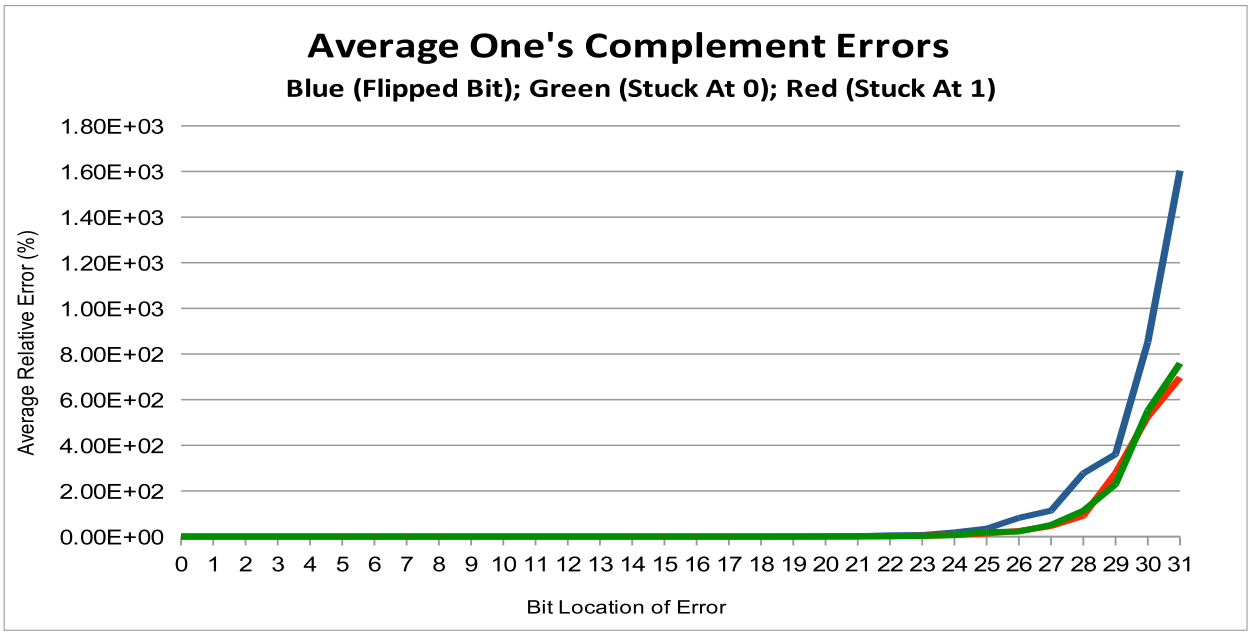
After running each of the trials, the average of each of the sets of five relative errors was calculated, producing the average relative errors for each of the bit locations in each of the

representations and error types. This data was put into charts according organized by numerical representation and bitwise error type and then graphed. An important fact to note is the arrangement of binary numbers. Specific bits are numbered in a reverse manner. In the binary value 110, for example, 0 is bit 0, 1 is bit 1, and the second 1 is bit 3.

2.4 – Division of Work

As partners, we split the work of writing/running the error inducing simulations for the various numerical representations. For example, while one partner wrote the code to convert decimal numbers into one's complement, the other wrote the code to convert decimal numbers into two's complement. In addition, when bugs showed up in the program, both of us worked to fix it so that a mutual understanding was reached at the end instead of a lopsided one. The graphical analysis was divided by representation also. One partner analyzed the error distribution for one's complement by graphing the data, and the other analyzed two's complement by graphing as well. Floating-point was analyzed in much the same way. We sought to divide work equally throughout the project to ensure that both of us could analyze and present the final result of our project.

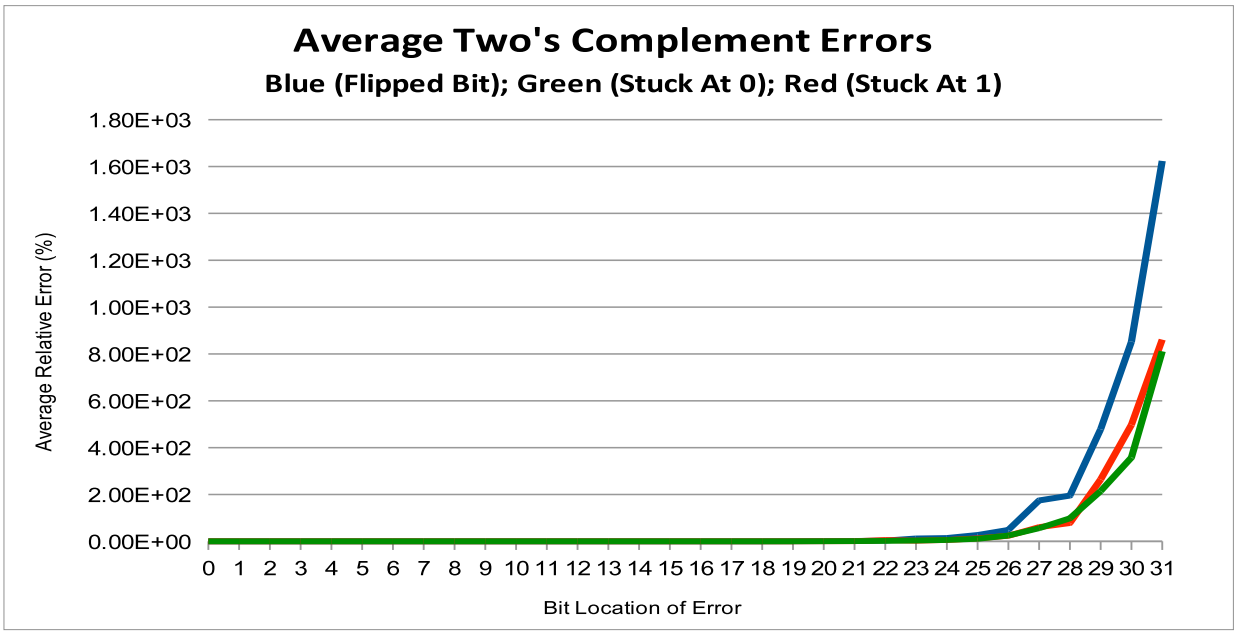
Many people in the lab that we worked in also supported our project. One of the graduate students under our professor taught us important concepts such as the Monte Carlo method and Hamming Code [9]. He also helped debug our code when we started seeing unreasonable error values. Another graduate student first explained to us the implications of our research, and an undergraduate student guided us through fundamental binary functions such as addition and subtraction to gain a better understanding of the base-2 system.



3.1 – One's Complement

One's complement errors seem to follow a general increasing trend. Errors in the less significant bit locations (from bit 0 to bit 23) have an average relative error of close to 0%, but the error rates begin to rise quickly at the more significant bits, with the flipped bit error rising more rapidly than either of the stuck-at errors. The error distributions of stuck-at-zero and stuck-at-one are much the same throughout the 32 possible locations of error. The flipped bit errors maintain a general value of approximately two times those of the stuck-at errors throughout the graph. The flipped bit errors reach a maximum of 1.604×10^3 % at bit number 31, while the stuck-at-zero and stuck-at-one errors reach maximums of 7.593×10^2 % and 6.975×10^2 %, respectively, also at bit number 31. In general, average relative error follows an increasing trend as the bit-location of the error increases to the more significant bits. An error in the more significant bits of a one's complement number would clearly have more of an effect than would an error in the less significant bits; thus the trend is reasonable.

Additionally, it makes sense that the flipped bit error distribution is twice the stuck-at-zero and stuck-at-one distributions since flipped bit errors affect a number twice as often as do stuck-at errors. Consider a hypothetical situation in which there is a flipped bit error at a certain bit location in one bit stream and a stuck-at-zero error in the same location of another bit stream. The flipped bit fault will always result in error, no matter the value of the binary number at the location of error, whereas the stuck-at-zero error will only affect the number if the bit stream has a value of 1 at the location of the stuck-at-zero error. If the value is 0 at the location of a stuck-at-zero error, then the relative error would be 0%. Over a wide range of numbers, the value of a bit stream at a certain location will be 0 half the time and 1 the other half, which means that a stuck-at error would have only half the effect that a flipped bit error would have.



3.2 – Two's Complement

Errors for two's complement are very similar to those of one's complement, following an increasing trend as the bit location of error increases from 0 to 31. Again, the flipped bit error distribution remains roughly twice the amount of the stuck-at errors. This finding is discussed in Section 4.1, and the explanation remains valid for two's complement as well.

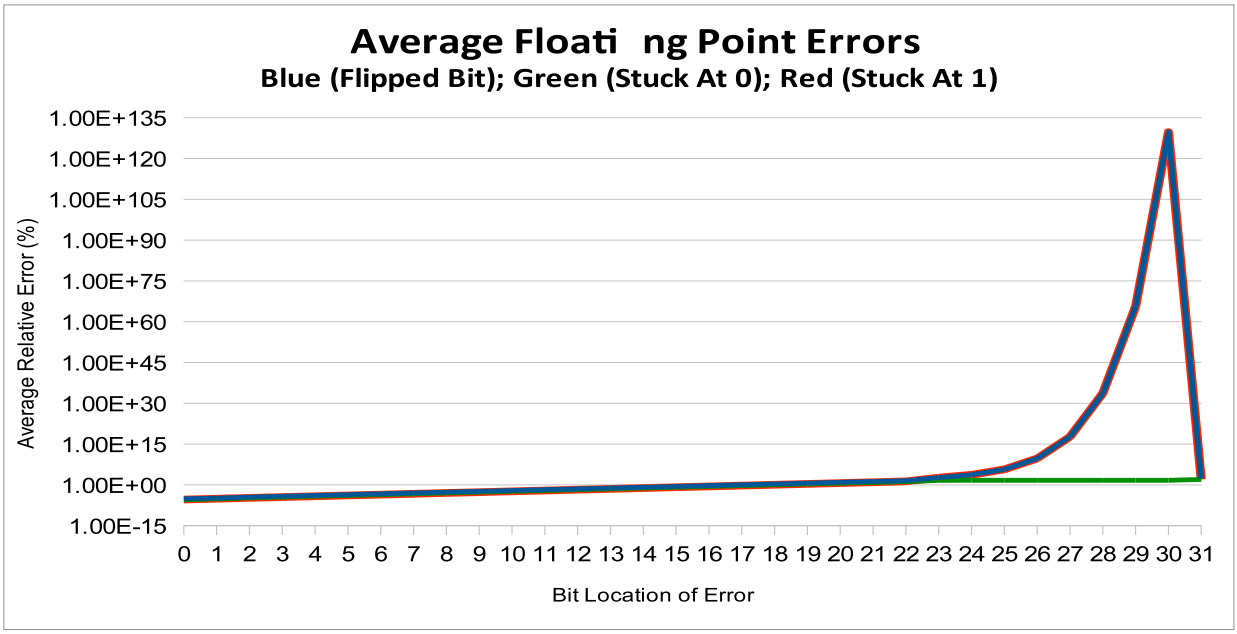
Further discussion is necessary, however, on the similarities between one's complement and two's complement. This similarity results from both representations being able to portray a range of numbers that is nearly identical. A 32-bit one's complement number can be any value from -2147483647 to 2147483647. A 32-bit two's complement representation has a range of -2147483648 to 2147483647. Moreover, both one's complement and two's complement are linear representations, which means that changing the value of a bit will always have the same effect, regardless of the values of other bits in the binary number. Here, we present the problem in a simplified sense. This example illustrates the similarities between one's complement and two's complement using 4-bit numbers:

Binary	One's	Two's	Binary	One's	Two's
1000	-7	-8	0000	0	0
1001	-6	-7	0001	1	1
1010	-5	-6	0010	2	2
1011	-4	-5	0011	3	3
1100	-3	-4	0100	4	4
1101	-2	-3	0101	5	5
1110	-1	-2	0110	6	6
1111	-0	-1	0111	7	7

The average percent change was calculated for flipped bit error at all 4 possible bit locations of error. For an error at bit 0, the absolute change is always equal to one. 1000 in one's complement is equal to -7, and with a bit flip error at location 0, the binary number changes to 1001, or -6. The absolute error here is 1, as mentioned previously, and the relative change is equal to 1/7 since relative error is the quotient of absolute change and original value. We then took the summation of the sequence of relative errors at each bit to find average relative error: $\frac{\sum (relative\ errors\ at\ bitlocation)}{number\ of\ possible\ values} \times 100$, with the number of possible values being 14 for one's complement and 15 for two's complement to account for the fact that 0 cannot be a value in

the denominator of a relative error calculation. Since the ranges of both representations are equal, the errors of one's complement and two's complement in 4 bits are also similar:

	One's Complement Avg. Rel. Error	Two's Complement Avg. Rel. Error
Bit 0:	37.04%	35.40%
Bit 1:	74.08%	70.81%
Bit 2:	148.16%	141.62%
Bit 3:	296.33%	283.24%



3.3 – Floating-point

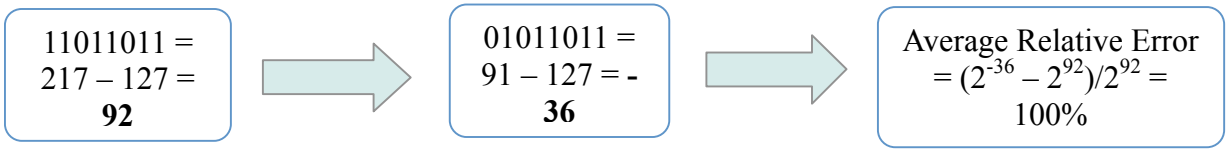
*Note: This graph is on a different scale than are the others. The y-axis is on a logarithmic scale, meaning that the errors in the floating-point representation are orders of magnitude higher than errors in one's or two's complement.

The drastic difference in error between floating-point and the other representations becomes logical when we consider the range of possible numbers in a 32-bit floating point representation. Section 2.1 of this paper discusses the components of a 32-bit floating-point number, of which one part is the exponent section. An error in these 8 bits would result in significant changes from the original number. As is shown in the graph, errors stay relatively low until the bit location of error reaches bit 23, the first bit of the exponent. Errors rise exponentially

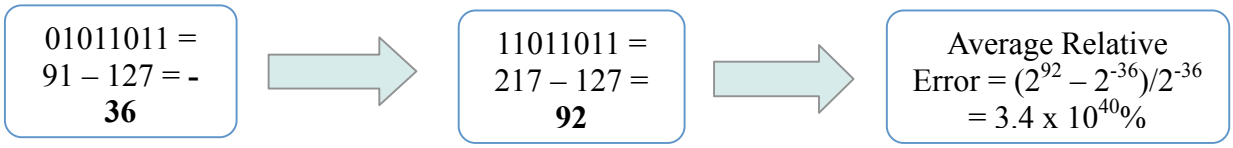
from there until bit 30, the last bit of the exponent. The subsequent drop in average relative error is a result of the bit location of error reaching the bit 31, the sign bit. A flipped bit error in the sign bit always causes an error of 200% since the number would undergo a change of twice its own magnitude.

Then we move on to the relationship between stuck-at-one and flipped bit in the floating-point representation. The minimum possible flipped bit error in the exponent bits of a floating-point number is 100%. This occurs when a 1 is flipped to a 0. When a 0 flips to a 1, the error becomes astronomical. The scenario below illustrates errors in floating point. The first scenario is when the most significant bit of an exponent in floating-point gets flipped from a 1 to a 0, and the second scenario represents the reverse situation, a flip from 0 to 1 in the same position.

Scenario 1:



Scenario 2:

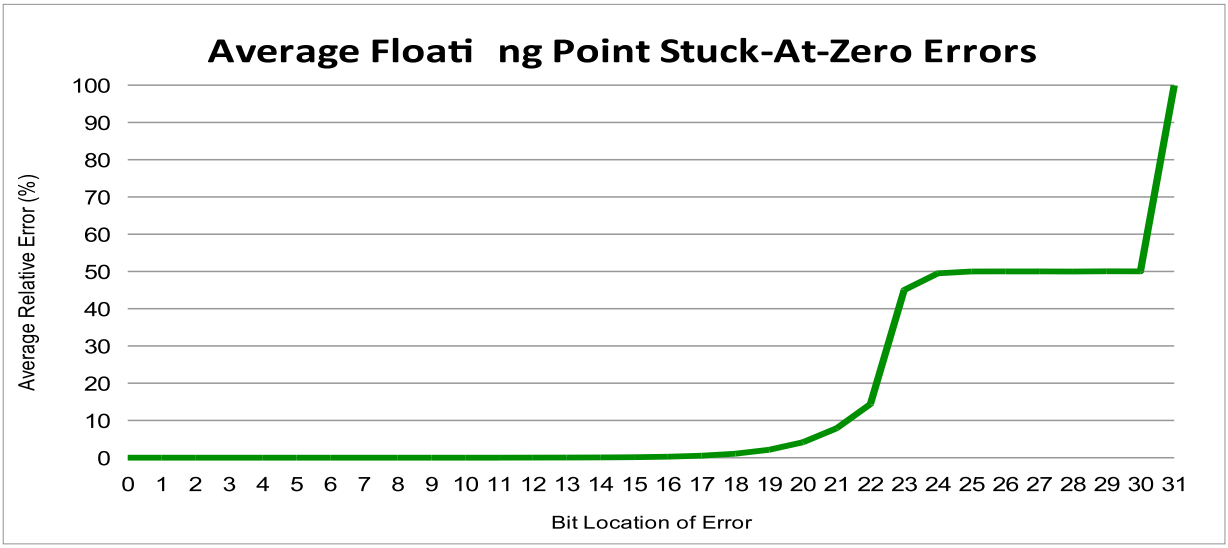


*Note these are the exponents in a floating point number in form (sign) x (signficand) x $2^{exponent}$

In the case of a stuck-at-one error, Scenario 2 would remain the same (bit value changes from 0 to 1), but Scenario 1 would result in 0% error instead of the 100% error of flipped bit. The change from 100% error to 0% error makes no significant difference when calculating overall average relative error because the other error percentages are orders of magnitude greater. Thus, the flipped bit and stuck-at-one error distributions appear extremely similar in the graphical analysis.

3.4 – Floating Point Stuck-At-Zero Errors

The relationship between the stuck-at errors for floating-point is also interesting. Unlike the one's complement and two's complement representations, the stuck-at-one and stuck-at-zero error distributions differ by a significant amount in floating-point. The reason for this drastic difference again lies in the two hypothetical scenarios outlined in section 3.3. Stuck-at-zero errors are represented by scenario 1. The maximum possible relative error for a stuck-at-zero fault is 100%, as shown by scenario 1, and the minimum possible error is 0%, when the bit value is already 0 at the location of error. This makes the average relative error close to 50% for the exponent bits of a stuck-at-zero floating point error. Bit 31 is the sign bit, meaning that error is either 200% or 0%, averaging to 100%:



3.5 – Proposing an Improvement

We discovered that two's complement and one's complement were far better representations in terms of error tolerance than floating point. This observation was confirmed with further analysis: floating point representation contains a whole section of binary digits that represent exponents. An error in this section could result in numbers that were orders of magnitude higher or lower than the original number. Because of its anomalous behavior, we

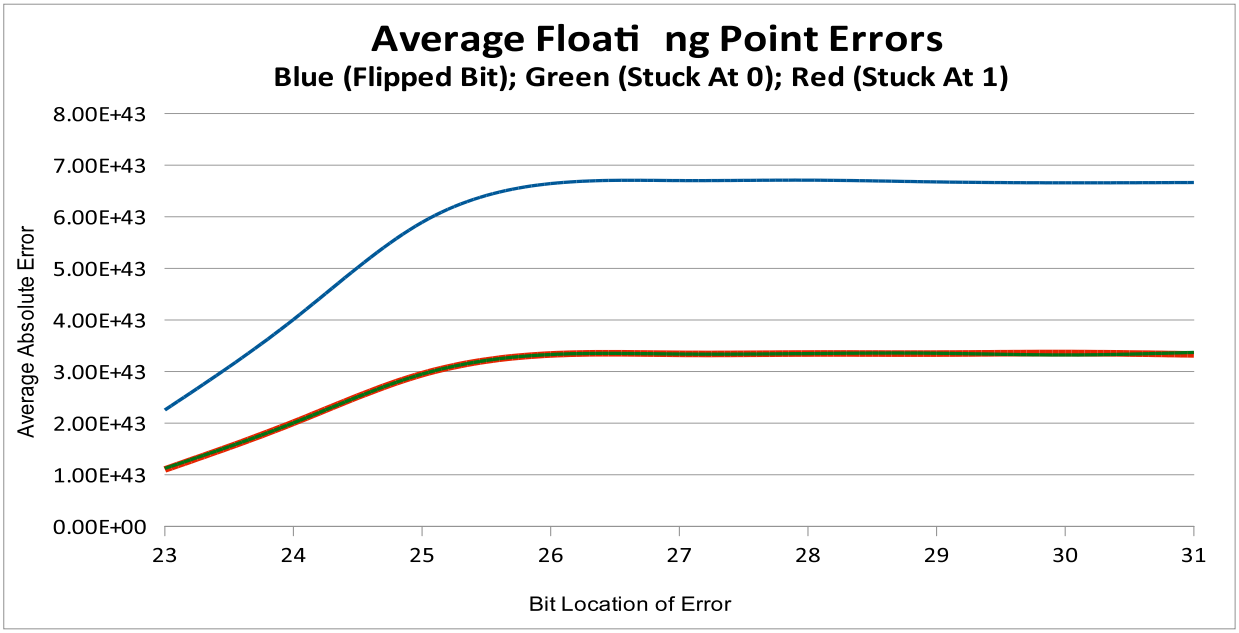
decided to modify the exponent section of floating point representation, changing it from the original biased form (where the decimal value of the binary number is biased by -127 in order to include negative exponents or, small numbers) into two's complement form. We hoped to reduce the amount of error of our new floating point representation by using a numerical representation that produces less error where error tolerance is critical, in the exponent section. We ran the tests for this altered form of floating point in the same fashion as the other numerical representations but used only one set of five million random numbers and tested induced errors in only the 8 exponent bits since those were the most significantly impacted by high error rates.

3.6 – Error Type and Optimized 'Two's Complement' Floating-point

In some situations more than others, relative error can be seen as a less than accurate way to portray error rates. The advantage of using relative error is that it is standardized irrespective of error magnitudes. A 50% error, for example, could be from 1 to 1.5 or from 1000 to 1500. Relative error was used in our analysis because it provided the unique advantage of being able to compare across representations even though the range of numbers in each representation differed. When comparing within floating-point itself, however (our optimization to floating-point can portray a range of numbers almost exactly the same as the original floating-point), absolute error would be a better gauge of error-tolerance. Therefore, our simulation was changed to represent average absolute error in the analysis of our optimization to floating-point. We first measured absolute errors in the original floating point representation for a comparison benchmark.

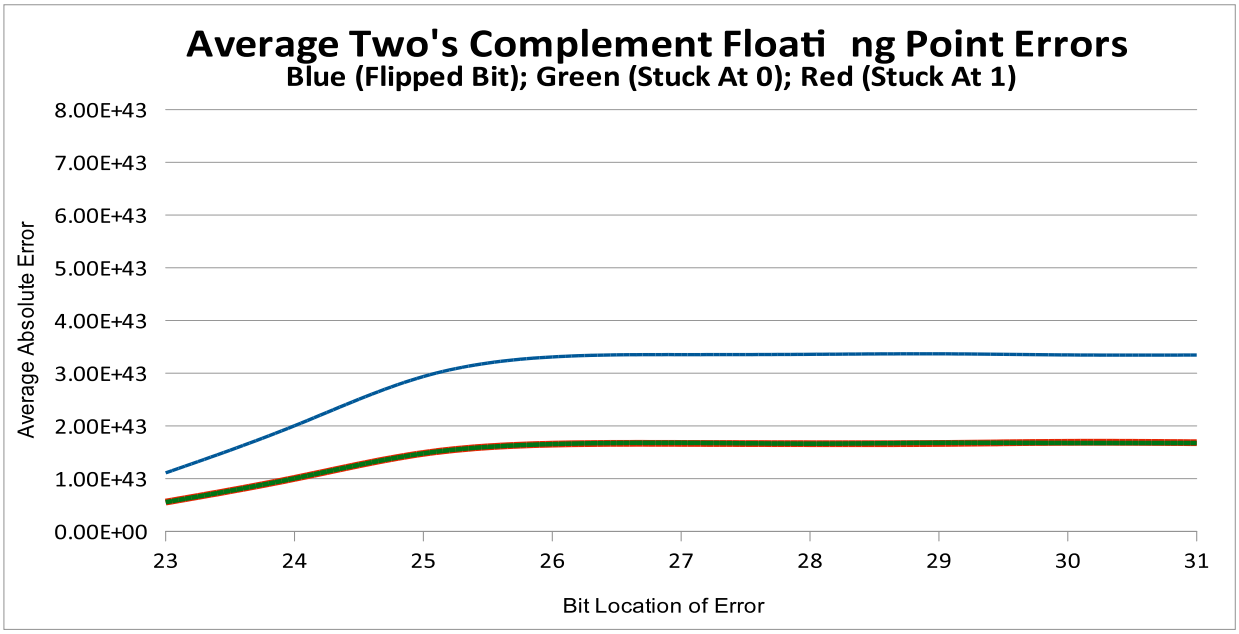
As is clearly shown, both flipped bit errors and stuck-at errors have a general increasing trend. The absolute errors for flipped bit are greater in magnitude than those of the stuck-at faults. Absolute error is unlike relative error here since stuck-at-one and stuck-at-zero have the

same error distributions. In a measure of relative error, each of the absolute errors would have to be divided by different numbers in the denominator, depending on whether a stuck-at-zero error or a stuck-at-one error occurred. This division is what made the stuck-at errors seem extremely different. In terms of absolute error, stuck-at-zero and stuck-at-one are equivalent. Because the exponent value of the floating point number itself, grows exponentially, (significand x $\mathbf{2}^{\text{value of the exponent bits}}$), the absolute errors will begin to decrease much more slowly as the bit-location increases. The maximum absolute error that can occur in the exponent bits is $2^{\text{original exponent value}}$, thus, the absolute error will level off as it approaches this value. Here are the results:



We now compare absolute errors of the original floating-point representation to absolute errors of the optimized floating point with two's complement in place of the original biased representation. Clearly, the two's complement floating point optimization produces a lower absolute error distribution. Errors in the original floating point are close to two times the two's complement floating point errors. This is true mostly because of a change in the range of the representation (the exponent represented in 8 bits of two's complement is comparable, but not exactly the same as the original floating-point itself). The range of an 8-bit two's complement

number (as in the exponent of the altered form of floating-point) has a range of -128 to 127, whereas the biased exponent in the original floating-point has a range of -127 to 128. Thus, because $2^{128}/2^{127} = 2$, the absolute error of the original floating point is twice as large as the error in the new version. Here are the new results:



IV. DISCUSSION

4.1 – IEEE floating-point Standard

The IEEE 754-2008 binary32 floating point standard (also known as single precision floating point), a widely accepted format that is consistently used in hardware and software technology today [10], was considered in our analyses. Existing work on errors in floating-point address a form of error completely different from the one we analyze here. The author in [10] discusses rounding error, a problem in the floating-point representation that is not related to errors that result from variability in computing.

4.2 – Uniqueness and Comparisons

Our research brought signed number representations to light in terms of robustness rather than the common characteristics that have often been used to classify them previously, such as

range, compactness, simplicity and arithmetic efficiency. Robustness will be a more significant classification metric in the near future as variability in computing platforms becomes more critical.

Other works have also dealt with errors in flash memory, [11] for example, dealt with the data integrity of flash memory in case of a power failure. Such analyses are clearly different from our work, however, because their focus is on characterization of error in a specific situation, whereas this work presents a general snapshot of the effects of error.

As previously mentioned, the authors in [5] addressed the fault density (number of faulty bits per unit area) in flash memory that would lead to an acceptable voice output when speech signals were encoded into a bit stream. The main focus of [5] was identifying the maximum possible fault density that would result in recognizable vocal patterns, whereas the main application of our work is to compare the effects of induced errors in various numerical representations. Further, the 2nd part of our work discusses methods to improve the error-tolerance of existing systems, which would decrease the effects of errors and allow a greater fault density with the same performance, a situation that [5] does not consider.

4.3 – Observed Trends/Implications

The major trend observed throughout our results is that a representation that can portray a larger range of numbers will usually have a lower error-tolerance. This result presents a unique trade-off, one between range of possible numbers to be represented (implicitly represents the same concerns as storage does) and error-tolerance of the representation. A possible method of analyzing the trade-off would be to create an average relative error to range' ratio and test various ranges of numbers by increasing or decreasing the number of bits used to display the number. The road to the future in signed number representations consists of finding a new balance

between storage capacity and error-tolerance.

The implications of a more error-tolerant numerical representation are clear. As of today, the downscaling of semiconductor components causes variability in chip performance. Manufacturers are forced to discard memory chips that do not meet their specifications, decreasing their overall raw yield. A more error-tolerant system will allow manufacturers to keep those previously discarded chips, allowing a greater yield as well as higher profit margins.

4.4 – Environmental Noise and Soft Errors

Error-tolerant, or stochastic, processing has benefits other than being able to deal with variability in number storage as well. First, error-tolerant software would increase the lifetime of a hardware. Old hardware becomes riddled with errors, some of which could be addressed through using software that allows insignificant errors to occur. Secondly, stochastic computing would have the further benefit of addressing the problem of soft errors. Soft errors are errors that are caused by interaction between charged particles in the environment (cosmic radiation) and a transistor [12]. The rate of soft error in designs at smaller scales is higher than that of designs in bigger scales [13], meaning that the rate is only going to increase in the future. However, many systems can tolerate a certain level of soft errors [14]. Stochastic computing will help solve the problem by letting insignificant errors happen and reducing the effects of more significant errors.

V. CONCLUSIONS and FUTURE WORK

5.1 – Summary and Validity of Results

A correlation between range of possible numbers in the representation and error rate was found. One's complement and two's complement, both with extremely similar ranges of possible values, also had extremely similar error distributions. Stuck-at errors were found to have approximately one half the effect that flipped bit errors had for both one's complement and two's

complement. Floating-point representation has a range of possible numbers much higher than that of one's or two's complement. Floating-point stuck-at-zero was found to be less harmful than either flipped bit or stuck-at-one since the a stuck-at-zero error always reduces the number. The highest possible error percentage of floating-point stuck-at-zero is 100%, which occurs when the most significant bit is a 1 and gets stuck-at-zero, and the lowest possible error was 0%. Therefore the average relative error was near to 50%.

The exponent bits of the floating-point representation were found to have the highest relative error rates. Thus, we embedded a two's complement notation in place of the biased exponent of the original floating-point representation and found that the average absolute errors when considering this altered representation were lower than the original.

As mentioned previously in Section 3.6, relative error may not be the best option to compare errors in specific situations. However, relative error is generally the figure to report as a result of its usefulness as a standard measure of performance across all disciplines. It also translates directly when a comparison is necessary between values of unequal scope or magnitude, and is thus a good metric.

5.2 – Future Work and Further Questions

Future work lies in finding the optimal ratio between possible range of numbers and error-tolerance for floating-point in particular. Further ways to modify the representations also exist. Error-correcting codes, such as Hamming Code, could be applied to reduce the amount of error and/or redundancy bits could be added to make sure the more significant bit values do not fail. The question we should ask ourselves going forward is: how can we optimize the ratio of storage to error-tolerance so that we obtain the greatest possible error tolerance and range of numbers simultaneously.

REFERENCES

[1] Gordon E. Moore, “Cramming more components onto integrated circuits,” *Electronics,* vol. 38, number 8, April 1965

[2] “Intel 22nm 3-D Tri Gate Transistor Technology,” May 2011, http://newsroom.intel.com/docs/ DOC-2032

[3] *International Technology Roadmap for Semiconductors,* 2006 update, http://www.itrs.net/Links/ 2006Update/2006UpdateFinal.htm

[4] David Lammers. “The Era of Error-Tolerant Computing,” *IEEE Spectrum*, Nov 2010

[5] H. Zhu and M. A. Breuer, “An Illustrated Methodology for Analysis of Error Tolerance,” *IEEE Design and Test of Computers,* vol. 25, issue 2, pp. 168 – 177

[6] X. Li, “Maximum-Information Storage System: Concept, Implementation, and Application,” *IEEE/ACM International Conference on Computer Aided Design*, Nov 2010

[7] X. Li, “Rethinking Memory Redundancy: Optimal Bit Cell Repair for Maximum Information Storage,” *IEEE/ACM Design Automation Conference*, pp. 316 – 321, 2011

[8] H.L. Anderson. “Metropolis, Monte Carlo and the MANIAC”. *Los Alamos Science.* pp. 14 – 96, 1986

[9] Frey, Brendan J., and David J.C. Mackay. "A Revolution: Belief Propagation in Graphs With Cycles." *Advances in Neural Information Processing Systems 10*. By Michael I. Jordan, Sara A. Solla, and Michael J. Kearns. Cambridge, MA: MIT, 1998. 480-81. Print.

[10] David Goldberg. “What Every Computer Scientist Should Know About Floating-Point Arithmetic”. *Computing Surveys.* Association for Computing Machinery 1991.

[11] Hung-Wei Tseng, Laura M. Grupp, and Steven Swanson. “Understanding the Impact of Power Loss on Flash Memory”. *DAC*. 2011.

[12] Subhashish Mitra. “Robust System Design.” Presentation: Robust Systems Group, Dept. of EE & Dept. of CS, Stanford University

[13] S. Mitra, N. Seifert, M. Zhang, Q. Shi, K.S. Kim. “Robust System Design with built-in Soft-Error Resilience”. 2005

[14] Ritesh Mastipuram and Edwin C. Wee. “Soft Errors' Impact on System Reliability.” *Electronics Design, Strategy, News.* 2004.