Using NOR Flash Memory in Microcontrollers for Generating True Random Numbers

by

PRAWAR POUDEL

A THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Engineering
in
The Department of Electrical & Computer Engineering
to
The School of Graduate Studies
of
The University of Alabama in Huntsville

HUNTSVILLE, ALABAMA

2018

In presenting this thesis in partial fulfillment of the requirements for a master's degree from The University of Alabama in Huntsville, I agree that the Library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by my advisor or, in his/her absence, by the Chair of the Department or the Dean of the School of Graduate Studies. It is also understood that due recognition shall be given to me and to The University of Alabama in Huntsville in any scholarly use which may be made of any material in this thesis.

_____ _____

(student signature) (date)

THESIS APPROVAL FORM

Submitted by Prawar Poudel in partial fulfillment of the requirements for the degree of Master of Science in Engineering in Computer Engineering and accepted on behalf of the Faculty of the School of Graduate Studies by the thesis committee.

We, the undersigned members of the Graduate Faculty of The University of Alabama in Huntsville, certify that we have advised and/or supervised the candidate on the work described in this thesis. We further certify that we have reviewed the thesis manuscript and approve it in partial fulfillment of the requirements for the degree of Master of Science in Engineering in Computer Engineering.

_____ Committee Chair
(Date)

_____

_____

_____

_____ Department Chair

_____ College Dean

_____ Graduate Dean

ABSTRACT

The School of Graduate Studies
The University of Alabama in Huntsville

Degree Master of Science in Engineering College/Dept. Engineering/Electrical & Computer Engineering

Name of Candidate Prawar Poudel
Title Using NOR Flash Memory in Microcontrollers for Generating True Random Numbers

This thesis introduces a new technique for generating true random numbers that exploits read noise of perturbed cells in NOR Flash memories. The existing techniques for generating true random numbers rely on additional hardware resources that serve as a source of entropy or exploit memory components such as SRAM or NAND Flash memories. The proposed technique utilizes NOR Flash memories that are readily available in modern microcontrollers. We characterize behavior of the NOR Flash memories, introduce an algorithm for inducing a perturbed state of Flash memory cells, and introduce an algorithm for extracting randomness from these cells and generating true random numbers. The proposed technique is demonstrated and experimentally evaluated on a TI MSP430 family of microcontrollers. The experimental evaluation shows that the proposed technique enables high-throughput and low-cost extraction of random sequences that pass tests from the NIST statistical suite. The proposed technique requires no hardware modifications, is entirely implemented in software, and can be tailored to work in low-end and low-cost embedded systems.

Abstract Approval: Committee Chair _____

Department Chair _____

Graduate Dean _____

This thesis is dedicated to
all who have supported me.

ACKNOWLEDGMENTS

My greatest and sincere thanks goes to Dr. Aleksandar Milenkovic, who served as my advisor. Besides that, he has always been the person that I could reach out anytime I needed help. He has been a guardian figure for last six months for me, a motivator and counsellor at times.

I would also like to thank Dr. Jeffrey Kulick. In addition to serving in my committee, Dr. Kulick served as my instructor in several courses. Working as a TA in his Operating Systems course gave me an opportunity to learn new things in the area of computer security.

Special thanks goes to Dr. Biswajit Ray, who has constantly provided moral and technical support. His initial work in NAND Flash memory true random number generators served as a starting point for this research.

I am grateful to my lab mates Mr. Ranjan Hebbar Raviraj and Mrs. Mounika Ponugoti. They have been true friends to me whom I could reach out even for smallest of the problems that I faced. Thank you for helping me selflessly, even during the busiest of your times.

I would like to express my gratitude to Dr. Gorur, the ECE Chairman, for his continual support. Special thanks goes to Ms. Jacqueline Siniard who has helped me and all other students with the paper works and administrative matters.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Background and Motivation

Random numbers are corner stone in many computer applications, including secure communication and authentication, simulations, machine learning algorithms, games, and electronic gambling, to name just a few. These applications pose a different set of requirements for random numbers, some requiring numbers to be non-deterministic and others requiring deterministic random numbers. In secure communications, a communication session between two parties is identified by a session key. This key needs to be unique and unpredictable. Every message being transferred between communicating parties is encrypted. Keys used for encryption and decryption operations also need to be as unpredictable as possible. But in cases like simulation, to achieve the repeatability of the result, random numbers are desired to follow a certain pattern.

Generation of random numbers may utilize non-deterministic physical phenomenon or some deterministic algorithms. True random numbers are a result of non-deterministic physical phenomenon. Sampling and conditioning physical source for producing output forms a low throughput system. Pseudo-random numbers are a

result of deterministic operations on small true random numbers, often referred to as *seed*. This makes generation of pseudo random number a high throughput system. However, this also poses a risk of many cryptanalytic attacks [1]. One recent attack on casinos exploiting the deterministic nature of pseudo random numbers and attackers ability to guess the seed caused significant loss to the gambling industry [2]. True random numbers are more secure against such attacks. However, they pose challenges in terms of complexity and throughput since they have to be extracted from physical random processes. This highlights a need for a simple system able to produce a true random number.

Flash memories are available in almost every type of computing devices, from low-end embedded systems, mobile devices and smartphones, to high-end computers. They are an important part of modern microcontrollers that are brains of modern Internet-of-Things. This makes Flash memories an ideal platform for generating true random numbers.

1.2 What is this thesis about?

This thesis introduces a new random number generator that is easy to use, scalable, fast, and robust. It exploits read noise of perturbed NOR Flash memory cells that are typically found in microcontrollers. Flash memory cells exhibit threshold voltage fluctuations caused by thermal noise and random telegraph noise effects. Recent proposals have demonstrated how these inherent Flash memory characteristics can be exploited for generating true random numbers. However, these proposals focus on NAND Flash memories that have higher density and are more susceptible to noise and they cannot be directly applied to NOR Flash memories. NOR Flash

memories are used in modern microcontrollers to store code and read only data. They have lower density and feature larger memory cells that are less susceptible to noise and have high endurance.

We characterize behavior of NOR Flash memories found in a commercial microcontroller family and introduce an algorithm for inducing a perturbed state in Flash memory cells. Perturbed Flash memory cells can be read as either logic 1 or logic 0 depending on read noise, which is a combination of both thermal and random telegraph noise effects. We introduce an algorithm for generating true random numbers from strongly perturbed Flash memory cells.

The proposed algorithms have been demonstrated on the TI's MSP430 family of microcontrollers. The generated sequences are tested using the NIST statistical test suite [3]. The evaluation shows that the proposed algorithm passes the NIST tests.

1.3 Contributions

The major contributions made through this thesis are as follows:

1. We introduce an algorithm for inducing perturbed states in NOR Flash memory cells.
2. We characterize split-gate NOR Flash memory found in TI's MSP430 family of microcontrollers.
3. We introduce an algorithm for generating true random numbers that utilizes read noise of perturbed NOR Flash memory cells.

4. We demonstrate the proposed techniques and evaluate their effectiveness and quality of the generated random sequences using the NIST statistical tests.

1.4 Outline

This thesis is organized as follows. CHAPTER 2 gives background. It introduces random number generators and gives an overview of related work from the open literature. CHAPTER 3 discusses Flash memory organization in microcontrollers. CHAPTER 4 introduces the proposed algorithm for perturbing Flash memory cells, the algorithm used for identifying the nature of perturbed Flash memory cells, and finally the algorithm for generating true random numbers. CHAPTER 5 discusses the experimental environment, and presents the two aspects of implementation, analysis and production. CHAPTER 6 presents the results of the experimental evaluation. CHAPTER 7 presents the concluding remarks and future work.

CHAPTER 2

RANDOM NUMBER GENERATORS

Random numbers are routinely used in many areas of computing, including simulations, gaming, electronic gambling, and secure communications and cryptography. Software and hardware artifacts used to generate random numbers are called Random Number Generators (RNGs). The requirements for random number generators differ depending on application type. Some applications, e.g., simulations and gaming, require random numbers to be deterministic or predictable. Some other applications, e.g., cryptography applications, require random numbers to be nondeterministic or unpredictable. Thus, we recognize two major types of random number generators: (a) Pseudo Random Number Generators (PRNGs) and (b) True Random Number Generators (TRNGs).

Section 2.1 discusses types of random number generators. Section 2.2 discusses relevant related work where we present academic as well as commercial implementations of random number generators. In Section 2.3, we give a brief overview of the existing software-based random number generators, and finally in Section 2.4 we present the case for the proposed TRNG.

2.1 Types of Random Number Generators

2.1.1 Pseudo Random Number Generator

Pseudo Random Number Generators (PRNG) are also called Deterministic Random Bit Generators (DRBG). DRBGs are the software artifacts that use a random seed to produce a bit sequence that acts as a random number. This means that they have a predefined algorithm that takes in the seed as an input, performs some operations, and gives a random number as the output.

The output of such a DRBG is random as long as the seed cannot be guessed correctly. The whole operation of a DRBG thus depends on the seed value and complexity of the deterministic algorithm employed internally.

National Institute of Standards and Technology (NIST) defines recommendations in SP 800-9A [4] how to generate random number using DRBGs. The NIST also specifies conditions an entropy source should meet to be used for DRBGs, several algorithms that can be used for DRBGs, implementation issues, and assurance conditions.

Figure 2.1 shows the functional model of a DRBG described in the NIST's SP 800-9A [4]. The Instantiate function takes in the entropy input from an entropy source, combines it with other inputs if available, e.g. nonce value and personalized string, and produces the seed. The Reseed function creates the seed for other rounds by combining the entropy input, other available inputs and the current internal state of DRBG. Here, the internal state refers to all the information that is stored about the DRBG. Reseeding, in particular, is an important factor since using the same seed over time might reveal sufficient information that might compromise the

security of DRBG. When requested, the Generate function generates the pseudorandom values based on the current internal state, and also saves the new internal state. The Generate function can be one of many functions. It can be a hash based function (e.g., Hash_DRGB and HMAC_DRBG), block cipher based algorithm, dual elliptic curve deterministic DRBG, etc.

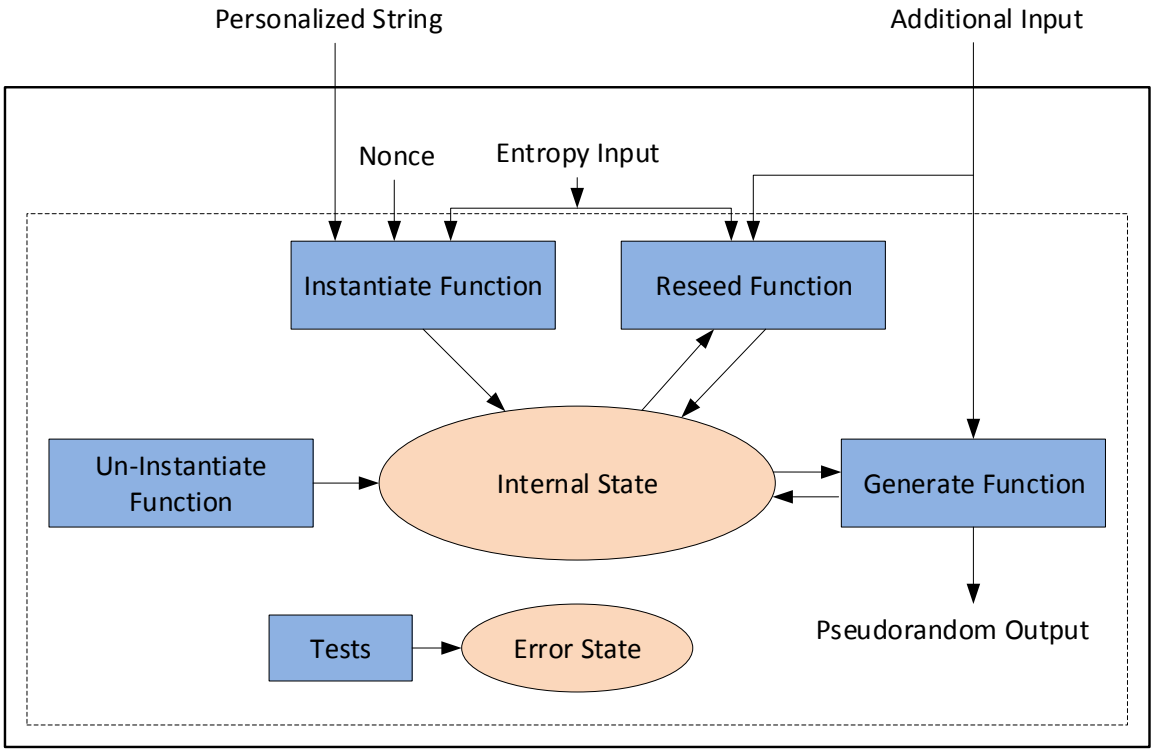

Figure 2.1 Functional Model of DRBG [4]

2.1.2 True Random Number Generators

True Random Number Generators, or TRNGs for short, are dependent upon a physical entropy source. These systems utilize the randomness in the entropy source to generate the output that is inherently random, after certain preconditioning steps. These preconditioning steps are dependent upon the situation and the entropy

source being used. However, TRNGs are considered to be not very efficient and are often difficult to implement. Since entropy is the major component, NIST recommendations presented in the NIST SP 800-90B [5] can be followed to verify if the entropy source in question can be used for random number generation.

Figure 2.2 shows a sample entropy source model as defined in NIST SP 800-90B [5] where a noise source is used to provide entropy. Entropy can be derived from multiple noise sources if the entropy derived from a single source is not enough to meet the randomness requirement. The output of such source/s has to be digitized since the output is highly likely to be an analog signal. Reading the user movement or thermal noise can be taken as an example of analog entropy source. After digitization, the signal can be processed to filter out more robust signal that is not anymore biased to any specific property of the entropy source. The last step of conditioning is mentioned as an optional step for further reduction of bias. This is a deterministic function that might depend on the prior steps and highly based on the signal that is under consideration. Health tests are also performed to check if the entropy level is maintained in the source. This is to ensure that any failure in the source of entropy does not affect the quality of output produced.

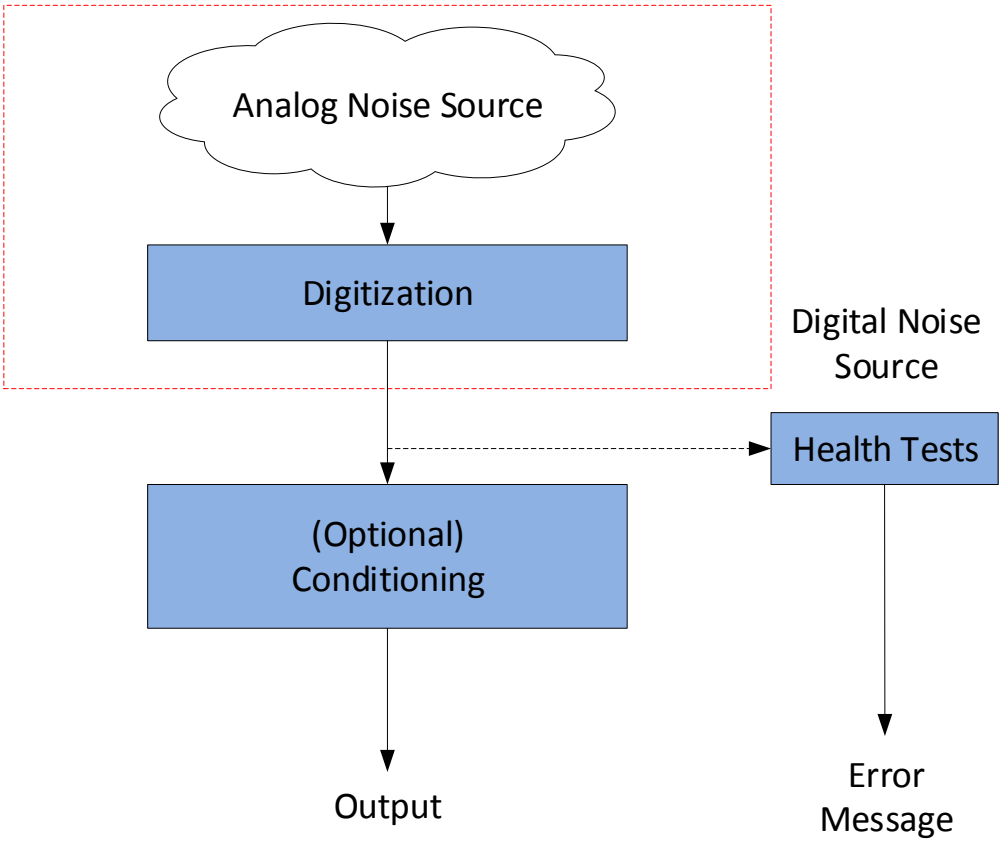

Figure 2.2. Entropy Source Model for TRNG [5]

2.2 Related Work

This section gives a brief overview of academic research in the area of TRNGs as well as commercial state-of-the-art implementations for generating random numbers. Section 2.2.1 discusses hardware TRNGs that makes use of electronic components for generating TRNGs. Although memory components like SRAM, Flash memory, DRAM are electronic components, we assign a separate section to describe them in Section 2.2.2. The primary reason for doing this is to discuss works utilizing memory components separately so that readers can have a clear view on works making use of memory components and get an idea about works relevant to our work.

Commercial random number generators are presented in Section 2.2.3. It discusses mainly Intel's and AMD's random number generators. Some other random number generators that are implemented using non-electronic means as major source of entropy are discussed in Section 2.2.4.

2.2.1 Electronic Components Based TRNGs

Oscillators are used as one of the major entropy source for random number generators. Maiti et al. use clock jitters from ring-oscillators to produce true random numbers [6]. Here, output of many ring-oscillators are fed into XOR gates and the output is fed into a D flip-flop. Post-processing gives a TRNG, while the pair-wise comparison of the ring-oscillators output is used to create a Physical Un-clonable Function (PUF) that can be used to identify a device. Texas Instruments has released an application note on an oscillator-based random number generator using MSP430 family of microcontrollers [7]. Here, two independent clock sources are used and the timing differences induced by these clock sources is exploited for generation of random numbers.

Majzoobi et al. implement a metastability based TRNG in FPGA-based TRNG [8]. In this work the authors induce metastability in bi-stable circuit elements by using programmable delay lines. Hata et al. achieve metastability using two look-up tables for NAND gate in FPGA implementation of RS flip-flop [9]. Multiple such latches are input to an XOR gate to produce a single bit of a random number, thus forming TRNG. Tokunaga et al. propose a quality control based TRNG using metastability [10] by implementing a mechanism to counteract the changes in TRNG. Here, the resolution time is recorded for each of the metastable events which allows

the capability to tune the output. Wu et al. [11] introduce four different circuits that make use of metastable to bi-stable state transition for generation of true random numbers. The major idea is that when a circuit switches from metastable to bi-stable state, the resulting state will be random. Wieczorek et al. use the concept of dual-metastability [12]. Here the researchers, instead of relying on the instability of the logical states, use time taken by bi-stable circuits to resolve its state, or resolve time, as a source of entropy.

2.2.2 Memory Components Based TRNGs

Several research proposals have introduced TRNGs that utilize Flash memories as a source of entropy. Wang et al. use NAND-based Flash memory for generating true random numbers [13]. In addition, they also demonstrate how the same Flash memory can be used as device fingerprint. Here, the Flash memory cells are brought into unstable state using repeated partial programming. Next, the noise cells are characterized into those that exhibit Random Telegraphic Noise (RTN) [14] or Thermal Noise. The Flash memory cells that exhibit RTN or both RTN and Thermal Noise are used for generating true random numbers. For device fingerprinting, the authors use threshold voltage variation of Flash cells to create unique identifier for each chip. Wang et al utilize program time of Flash memory cells as a technique to hide information in Flash memory cells [15].

Ray and Milenkovic introduce an alternative NAND-based Flash memory TRNG. Their technique [16] uses repeated programming of Flash memory using checkerboard pattern to stress the Flash memory cells. This causes the threshold voltage to fluctuate from its original level introducing read noise in memory cells.

The noisy Flash memory cells that exhibit either thermal, RTN or both thermal and RTN noise behavior are used for generating random numbers, thus increasing throughput.

Another approach to generate random number exploits the power-up state of SRAM memory. Holcomb et al. demonstrate that on power-up, SRAM cells are in an undefined state – some cells are skewed towards logic 0, some cells are skewed towards logic 1, and some cells are neither [17]. The SRAM cells that are skewed can be used to create a unique device fingerprint. The state read from a single power-up is termed as latent fingerprint of the particular SRAM. From the latent fingerprint of 512-bits, a random number of up to 128 bits in length can be generated.

DRAM memories exhibits remanence effects. Remanence effect refers to the procedure by which some information remains in the memory even after the power is turned off. Using this property, researchers have proposed extraction of random numbers. Tehranipoor et al. [18] propose a method in which they write 1's to all the DRAM cells in use, power them off and after some delay time turn the power back on to find not all the bits are settled to 0. They say that the difference in storage capacitance of each bit cause them to have different properties, which makes it possible to read random values out of them.

At start up, DRAM cells act similar to SRAM cells and do not all go to state of '0' as expected, but rather go to either state of '1' or '0'. Tehranipoor et al. develop PUF exploiting variation in each of the DRAM cells' start-up values [19]. Pyo et al have proposed refresh cycles in DRAM cells can cause variations in access times which can be exploited to produce random numbers [20].

Vendor specific Flash memory has also been used to generate PUF as well as random number. Clark et al. use SST39VF1601C device from Microchip Inc. to interrupt the erase cycle of the Flash memory thereby giving partial erase operation [21]. The key idea here is that over multiple trials, same fingerprint from bit values read can be generated with some bit mismatches. Certain functions are applied on the raw-bit strings to generate TRNG.

Balatti et al. use resistive switching RAM (RRAM) for generation of random numbers [22]. Here, voltages are applied independently to two RRAM cells connected in parallel, and resulting output voltage is measured. The High Resistance State (HRS) of the cells cause large statistical variations which is the source of entropy in this experiment.

Jiang et al. develop TRNG based on memristors [23]. Delay in threshold switching in the memristors at On-switching is exploited to generate true random numbers. Here, a constant width pulse is applied to the memristor. Under this voltage, the memristor is turned On which would cause sudden rise in output voltage after some delay time. A comparator is used to compare this output with a pulse.

2.2.3 Commercial RNGs

Intel introduced its random number generator that uses a ring oscillator based analog circuit [24]. Their RNG uses Johnson's Noise or thermal noise derived from voltage across un-driven resistors [25]. Since this RNG is based on an oscillator, the thermal noise was used to drive the slow clock. The output of a faster clock is sampled on every output of the slower clock. The researchers claim that given the slower clock contains enough randomness, any other non-random signals should not

affect the quality of randomness. The major demerits of this circuits is that the signal used to drive the slower clock are very small and they have to be amplified, which would consume more power.

Intel introduced an alternative TRNG that exploits metastability in digital circuitry. Moving to digital circuitry made the throughput increase from few kilobits per second to 3 gigabits per second [24]. Intel also provides machine instructions for dealing with the random numbers *RDRAND* and *RDSEED* [26]. In this technology named Bull Mountain [24] [26], they sample the thermal noise source, which is the entropy source, condition it with AES-CBC-MAC that will generate a high quality random seeds of 256 bits. This seed is used to generate Cryptographically Secure Random Number (CSPRNG) and other high quality seed.

Ring Oscillators are also a major source of randomness in case of other computer vendors. AMD [27] uses 16 ring oscillators as the source of entropy in its random number generator system under Cryptographic Co-processor (CCP). Here, 512 raw bits are fed to AES-256 CBC-MAC to construct 128 bit of random seed. The process is repeated three times so that an initial seed of 384 bits is generated which is fed to DRBG.

2.2.4 Other Physical TRNGs

Several research efforts propose random number generators that use other physical components besides the electronic components as a major source of entropy.

Tang et al. [28] propose a random number generator that utilizes the noise from an image sensor in a smartphone camera. Guo et al. suggest using Phase Noise in laser signals to generate TRNG [29]. Similarly, Zhang et al. propose the random

number generation based on use of chaotic laser signals generated by dividing the original signal and using feedback [30]. Terashima et al. use an alternative approach for generating chaotic signals using photonic integrated circuit and employing post processing to generate random signal [31]

2.3 Software Based Tools for Random Numbers

There are many software based implementations of random number generators. Most of the time these random number generators use a source of entropy which is sampled for production of a random seed. The random seed is then fed to a deterministic algorithm. This deterministic algorithm then generates random number. Thus software based random number generators are inherently pseudo-random in nature.

rand() function is provided in C++ for the purpose of generation of random numbers under *cstdlib* library [32]. Many random number generation templates, like *linear_congruential_engine, mersenne_twister_engine* and *subtract_with_carry_engine* can be chosen in C++11 using *random* header [33]. Custom user defined seed can be defined using *seed()* function as well. r*and()* function is also available in OpenSSL library.

Random class in defined in Java for similar purpose. It uses a 48-bit seed and *linear congruential* formula [34]. Python also has implementation of *random* class, using function *getrandbits()* of which pseudo random bits can be obtained. It uses *mersenne_twister* generator internally [35].

2.4 The Case for NOR Flash Memory TRNG

TRNGs produce sequence of bits that are derived from a source that is non deterministic, which is the case in most of the systems discussed in Section 2.2. It can also be seen that most hardware based TRNG implementations described in Section 2.2 have some extra component attached to a processing element, while others seem to be stand-alone TRNG. This means if someone has to integrate a random number generation functionality in his/her product, it has to come from that added component, which increases on-chip area and hardware complexity. It might not be a viable approach in all the cases since increase in on-chip area increases the cost and area occupied. Ring oscillators based implementations in particular are known for consuming more energy.

We propose a Flash memory based random number generator. There have been implementations of TRNG using Flash memory, as already discussed. But the proposals are external components which can only be used as external TRNG while attached to another processing element. This does not always form a portable system. This is the case with previous works where off-the-shelf NAND Flash memory was used [16], [36]. Our work is based on NOR Flash memory that is available in all modern microcontrollers, typically integrated with processors on a single chip. Being on a single chip, and being addressable through programming makes our approach no-cost, zero extra space occupying solution for TRNG and totally portable. Our work is the first of its kind to use on-system Flash memory component as an entropy source.

Modification to hardware is another challenge in TRNG systems proposed until now. Our work is fully implemented in software and does not require any sys-

tem that is already deployed to go through any modifications in terms of hardware. Our work can be used as a library or a function call. Thus, it does not interfere with the already existing software in the system. Our implementation algorithm has small memory footprint, and has a high throughput for producing high quality random numbers. This makes it an easy-to-install and easy-to-deploy system. This also addresses the problem that most TRNGs proposed are only deployable to new ASICs.

Since our mechanism is implemented on-system, we can boast that our TRNG is robust in the sense that it is tamper proof. Any physical attacker has to attack the processor physically to damage the TRNG system. NAND Flash memory based techniques use the method of stressing the memory cells. Our method is based on partial programming, which is a one-time operation. So there is no chance of destroying the memory components this is particularly important in case of systems with NOR Flash memory since they have lesser endurance than NAND Flash memory.

CHAPTER 3

FLASH MEMORIES IN MICORCONTROLLERS

This chapter gives a brief background in Flash memories. Section 3.1 introduces basics of Flash memories with a special emphasis on NOR Flash memories that are used in this thesis. It presents physical structure of Flash memory cells, Flash memory organization, and basic Flash memory operations. Section 3.2 presents the programmer's view of Flash memory in microcontrollers. Section 3.3 describes perturbed states of NOR Flash memory cells and how they can be exploited for generating random numbers.

3.1 Flash Memory Basics

Flash memories are non-volatile memory components. This means that data stored in Flash memories are retained even when the power supply is turned off. They are found in almost all the electronic devices that store information and/or contain software or firmware. They are found in computing systems ranging from small embedded devices to laptops and workstation computers, or specialized standalone storage devices. In Section 3.1.1 we discuss general structure of Flash memory, while in Section 3.1.2, we discuss the structure of a single memory element of the Flash memory that is used in this research.

3.1.1 Structure of Flash Memory

Flash memories are organized as an array of memory cells. Each of the cells is a floating gate transistor. Floating gate transistors are MOSFETs that are similar to standard MOSFETs but contain an extra gate. This means that floating gate transistors have two gates: a control gate (CG) and a floating gate (FG) as shown in the Figure 3.1. The floating gate (FG) is surrounded on all sides by oxide layer which makes it completely electrically isolated and the charge is trapped in it. To program a Flash memory cell means inducing charge carriers i.e. electrons to the floating gate (FG) and to erase means to remove these charge carriers. Logically, erasing would change the state from '0' to '1' and programming would change the state from '1' to '0'.

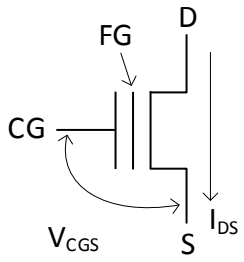

Figure 3.1 Floating Gate Flash Memory Cell

Based on the arrangement of the memory cells, the Flash memory can be of two types: a) NAND Flash memory and b) NOR Flash memory. Section 3.1.1.1 and Section 3.1.1.2 introduce each of these Flash memory, respectively.

3.1.1.1 NAND Flash Memory

Memory cells in NAND Flash are organized in series as shown in Figure 3.2(a). This series array is called NAND String [37] and can contain 32 to 64

memory cells. There are two selection transistors that connect each NAND String to Source Line and Bit Line.

Figure 3.2(b) shows NAND Flash memory that combines multiple NAND Strings from Figure 3.2(a) to form a complete storage element. A Word Line (WL) connects control gates of all memory cells in a row. Cells that are connected by a single Word Line form a memory Page, which is the smallest unit that can be programmed. Multiple Word Lines form a Block in a Flash memory, which is the smallest unit to be erased i.e. any erase operation has to span an entire Block. Each of the cells store either a single bit of information in case of Single Level Cell (SLC) technology or multiple bits of information in case of Multi-Level Cell (MLC) Flash technology.

NAND Flash memory allows for higher storage density compared to other kinds of Flash memory, e.g. NOR Flash memory. They have faster program times and faster erase times. However, since they are arranged in series random access is not available. They are good for high volume storage applications, but since they do not provide random access, they are not used in cases where execute-in-place[1] functionality is required.

[1] Execute In Place is a method of executing codes from memory where they are stored rather than copying them to RAM

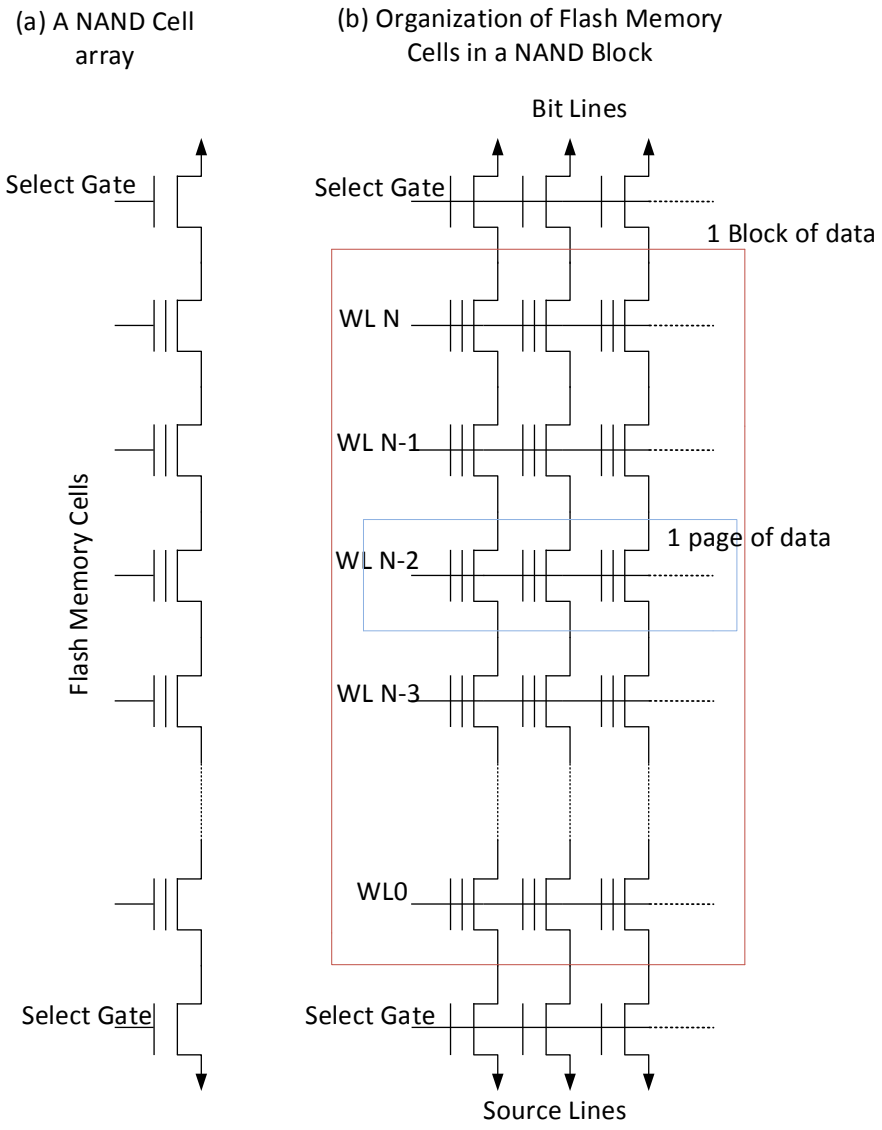

Figure 3.2 NAND Flash Memory Structure.

(a) Arrangement of Flash Memory Cells in an Array Forming NAND String (b) A Complete Storage Element Forming NAND Flash Memory

3.1.1.2 NOR Flash Memory

Memory cells in NOR Flash memory are arranged in parallel. Here, all the control gates of memory cells in a row are connected through Word Line (WL). Simi-

larly, the memory cells in a column are connected through the Bit Line (BL) which connects the drain of these cells. The source is connected to a common source terminal as shown in Figure 3.3.

Since the memory cells are connected in parallel, random access of the data is possible in NOR Flash memory. Random access of data is particularly useful in cases where execute-in-place is a necessity. Execute-in-place allows the code stored in NOR Flash memory to be executed from Flash memory without copying the code to RAM, which is often the case in embedded applications. Another advantage of a NOR based Flash memory is that byte or word programming is possible, while in case of NAND Flash memory, most of the times the minimum entity to be programmed is a page, which is much larger than a word.

Figure 3.3(a) shows a NOR Flash memory block. Multiple such blocks together form a segment. A segment is the smallest unit of memory that can be erased before programming again. Multiple such segments group together to form a bank as shown in Figure 3.3(b). Here the capacity of the illustrated Flash memory block is 128 bytes i.e. 64 16-bit words. Four such blocks form a segment and 128 such segments form a bank.

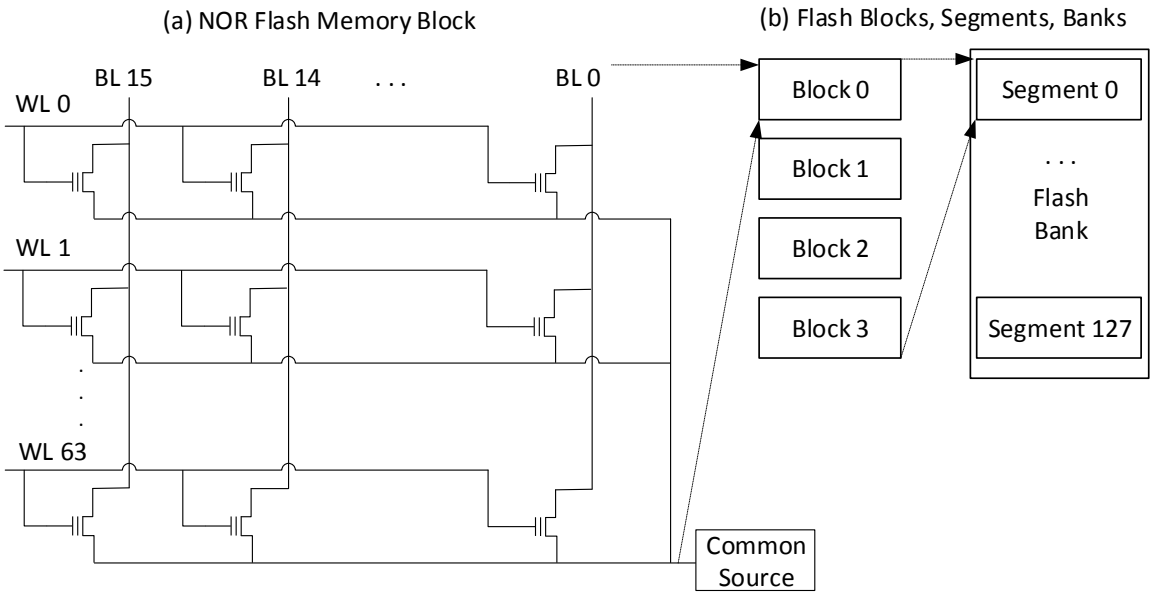

Figure 3.3 NOR Flash Memory Architecture.

(a) A Block of NOR Based Flash Memory (b) Organization of Flash Memory Blocks, Segments and a Bank

The Flash memory used in this work is a NOR Flash memory. Specifically, it is composed of Split-Gate Flash memory cells. The following section gives a brief description of Split-Gate Flash memory cell.

3.1.2 Split-Gate Flash Memory Cell

Flash memory used in our research is composed of a special memory cell, known as the split-gate Flash memory cell. Figure 3.4 shows a cross section of a split-gate Flash memory cell. The floating gate (FG) in case of split-gate Flash memory cell occupies only a portion of the area above the substrate between the source and drain. The Control Gate (CG) occupies a portion of the area between the source and drain, as well as the area that lies above the floating gate.

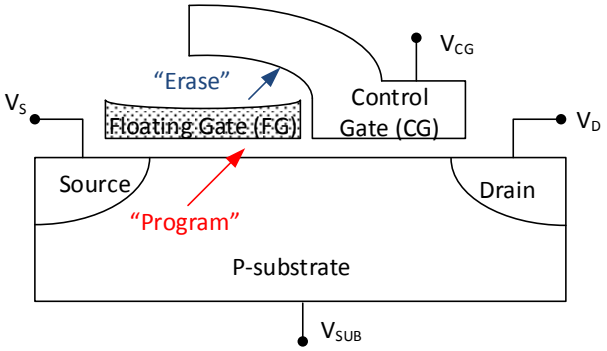

Figure 3.4 Cross Sectional View of a Split-Gate Flash Memory Cell

Flash memory cell can be in one of two states in a stable condition. Logic '1' is erased state while logic '0' is programmed state. Operationally, in order to change the state of Flash memory cell, either a *program* operation has to be performed, or an *erase* operation has to be performed. Other operation that can be performed in Flash memory is read. In this section, we will briefly discuss each of the operations for split-gate Flash memory cell.

To program the Flash memory cell means to charge the floating gate with electrons. During this operation, a large voltage (Vs > 10V) is applied to the source terminal. This application of large voltage results in electron injection on the floating gate by Source-Side Hot Carrier Injection (SSI) as illustrated by the red arrow in Figure 3.4. The electrons are thus trapped into the floating gate. This negative charge on the floating gate effectively lowers the voltage between the control gate and the source terminal which increases the threshold voltage (V_{TH}), which is now the threshold voltage for programmed state (V_{TH}=V_{THP}) as shown in Figure 3.5.

To erase the Flash memory cell, a large voltage has to be applied to the control gate. This large voltage (V_{CG} ~12V) removes the trapped electrons from the float-

ing gate through the Fowler-Nordheim (F-N) tunneling as illustrated by the blue arrow in Figure 3.4 [38], [39]. This removal of electrons reduces the threshold voltage V_{TH}, which is now the erase threshold voltage ($V_{TH}=V_{THE}$) as shown in Figure 3.5.

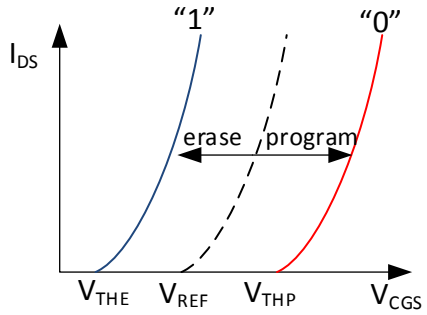

Figure 3.5 I-V Characteristic of Split Gate Flash Memory Cell

Memory Cells Reading from the Flash memory implies application of voltage to the control gate and the drain terminal. The voltage applied to the control gate is called the read voltage ($V_{READ}=V_{CG}=3V$) and that applied to the drain is called sense voltage ($V_{SENSE}=V_D=2V$). The threshold voltage is sensed to determine the state of the Flash memory cell. The erased cell, which does not have the trapped electrons in the floating gate would conduct the current, and thus give a logic '1'. On the other hand, the programmed cell will not be able to conduct the current, thereby giving a logic '0' as shown in Figure 3.6. The reference voltage (V_{REF}) is chosen between the programmed threshold voltage (V_{THP}) and erased threshold voltage (V_{THE}), so that the correct state can be identified.

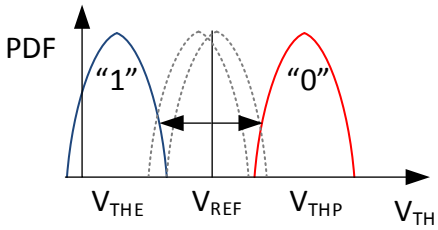

Figure 3.6 Threshold Voltage Distribution for Different States of Flash

3.2 Programmers' View of NOR Flash Memory

NOR Flash memory is an integral part of modern microcontrollers. By default, the operation to be performed in Flash memory is read in which case it acts as a ROM, but it can be programmed and erased as well. As already discussed in Section 3.1.2, it can be programmed and erased based on need. However, it should be noted that each of the bit can be programmed from '1' to '0' individually, but reprogramming from '0' to '1' requires an erase cycle.

The Flash memory module used in this research includes a controller that is responsible for controlling the erase and program (or write) operations. As shown in Figure 3.7, it consists of a voltage generator, timing generator and control registers. The voltage generator is responsible for generating voltages for erase and program operations. Timing generator controls the duration of operations. While in read mode, i.e. in the default mode the voltage generator and timing generator are both off.

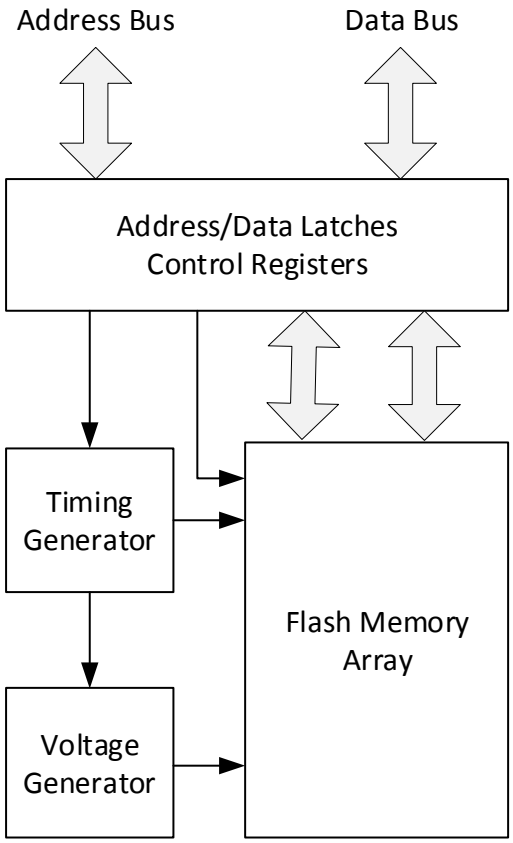

Figure 3.7 Flash Memory Module: Controller and Flash Memory

Writing to a Flash memory or the programming operation can be done from Flash memory itself or from RAM. While writing from the Flash memory, the CPU is halted until the completion of operation, but it is not the case when programming is done from RAM. The processor continues to execute the code from RAM.

Figure 3.8 shows a typical Flash memory programming operation cycle. It includes the time for the voltage generator to bring up the programming voltage, time to perform programming operation itself, and time to remove the programming voltage. During this whole period, the CPU cannot access the Flash memory and must wait until the completion of the operation (it can be detected by checking a flag

which in our case it is the BUSY signal), either the code is executed from Flash memory itself or from RAM.

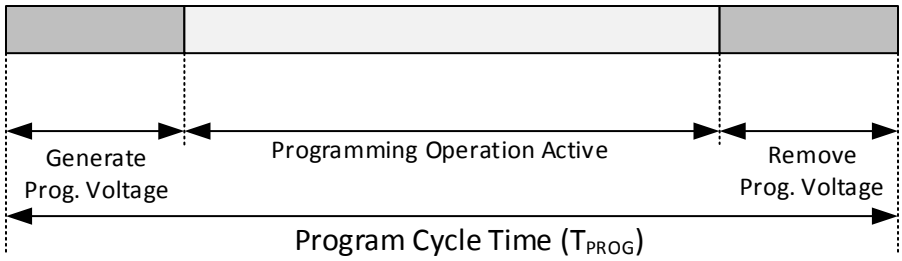

Figure 3.8 Typical Flash Memory Programming Cycle

Figure 3.9 shows a typical Flash erase cycle that includes times for the voltage generator to bring up erase voltages, time to perform erasing, and time to turn off the voltage generator. However, the erase cycle typically takes significantly more time than the typical program cycle. In our particular case, the Flash memory programming for a word/byte takes between 64 and 85 μs, whereas erasing a segment takes between 23 and 32 ms.

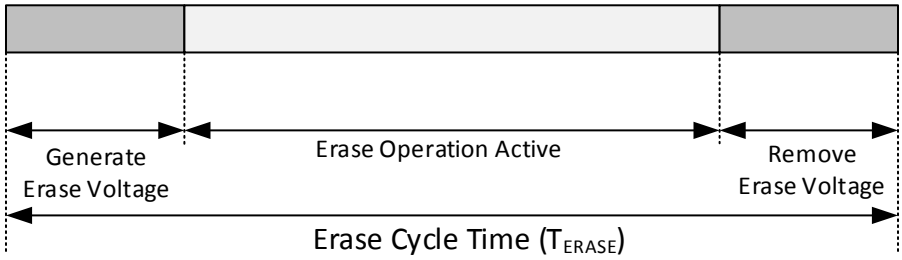

Figure 3.9 Typical Flash Erase Cycle

Figure 3.10 shows the program flow for Flash memory segment erase operation. Figure 3.11 shows a C language subroutine for the same operation. The first step is to check if the Flash memory is currently busy programming or erasing which

is done by checking the BUSY signal. Then, the Flash control registers are initialized operation (lines 5 and 7 in Figure 3.11). The erase operation is triggered by a dummy write operation (line 9 in Figure 3.11). The program waits for the operation completion by checking the BUSY signal which is high as long as the operation is in progress (line 11 in Figure 3.11).By setting the LOCK bit, the Flash memory returns to its default read mode.

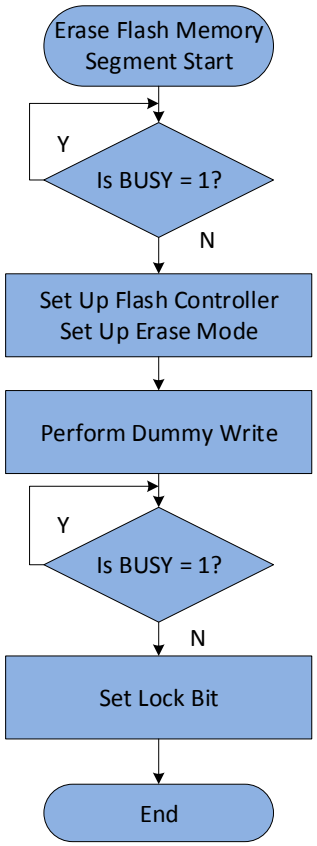

Figure 3.10 Flash Memory Segment Erase Cycle in Software

Flash Memory Erase Subroutine

```
 1.   void EraseFlashSegment(uint_16 *pFlashAdr) {
 2.   //wait while BUSY=1
 3.     while(FCTL3&BUSY);
 4.   //Clear LOCK bit to unlock the flash memory for erasing
 5.     FCTL3 = FWPW;
 6.   //Enable erase
 7.     FCTL1 = FWPW + ERASE;
 8.   // Dummy write (erase)
 9.     *pFlashAdr = 0;
10.   //Wait while BUSY=1
11.     while(FCTL3&BUSY);
12.   //Set LOCK bit to lock the flash memory for writing or erasing
13.     FCTL3 = FWPW + LOCK;
14.   }
```

Figure 3.11. Subroutine for Flash Memory Segment Erase

Program flow for writing to a Flash memory word is shown in Figure 3.12 while a C language subroutine for the same operation is shown in Figure 3.13. First the busy state of Flash is checked by checking the BUSY flag. Then, the controller is set into the programming mode by setting the corresponding control registers (lines 5 and 7 in Figure 3.13). The desired information is then written to a selected location (line 9 in Figure 3.13). The BUSY signal is inspected to detect the end of operation (line 11 in Figure 3.13) and then the default read mode is restores (lines 13 and 15 Figure 3.13).

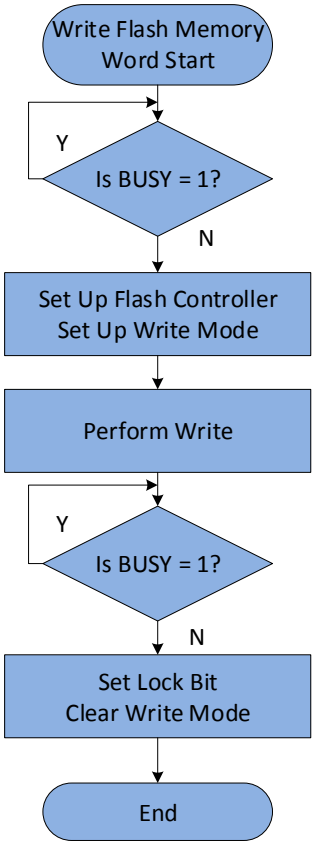

Figure 3.12 Flash Memory Word Write Cycle in Software

Flash Memory Write Subroutine

```
 1.   void WriteFlashWord(uint_16 *pFlashAdr, uint_16 wVal) {
 2.   //wait while BUSY=1
 3.     while(FCTL3&BUSY);
 4.   //Clear LOCK bit to unlock the flash memory for writing
 5.     FCTL3 = FWPW;
 6.   //Enable write
 7.     FCTL1 = FWPW+WRT;
 8.   //Write new value
 9.     *pFlashAdr = wVal;
10.   //Wait while BUSY=1
11.     while(FCTL3&BUSY);
12.   // Clear WRT bit
13.     FCTL1 = FWPW;
14.   //Set LOCK to lock the flash memory for writing or erasing
15.     FCTL3 = FWPW + LOCK;
16.   }
```

Figure 3.13 Subroutine for Flash Word Write

3.3 Perturbed States in NOR Flash Memory

In a steady state, read from the Flash memory cell gives either logic '1' or '0' depending whether it is in the erased or programmed state, respectively. During the read operation, as discussed in Section 3.1.2, the read voltage (V_{READ}) is applied on the Word Line (refer to Figure 3.3) and the sense voltage (V_{SENSE}) is applied to the Bit Lines. The current will flow from those cells in the selected word if the threshold voltage is less than read reference voltage. The threshold voltage in Flash memory cell might fluctuate because of the noise.

The noise is caused by either Random Telegraphic Noise (RTN) [14] or thermal noise, or both. We call this noise the read noise and it is always present in a Flash memory. Thermal noise is the white noise that is present because of the thermal agitation of the charge carriers. RTN is the noise caused by random trapping and emission of charge carriers in small electronic devices because of the defects present. This trapping and emission of the charge carriers cause random variations in the resistance of device. These variations are called RTN. However, the noise margin for reference voltage is generally kept at a safe level, so that the value read from a Flash cell is unaffected during any read operation.

Figure 3.6 shows the distribution of threshold voltages in cases when the logic state is either '0' (programmed) or '1' (erased), centered around V_{THP} and V_{THE}, respectively. The solid lines represent the state of bits in stable conditions, while the vertical line representing V_{REF} is the reference voltage used to determine the state of Flash memory cell.

In order to extract any behavior that is random, the approach is to agitate the Flash memory cells. The threshold voltage should be brought closer to the read ref-

erence voltage so that combination of the noises, referred to in our text as "read noise" would influence the threshold voltage, thereby giving alternate reading between '0' and '1'. The Flash memory cells in this state where the threshold voltage is brought closer to the reference voltage are referred to as perturbed cells.

Previous experiments done by Ray and Milenkovic [16] here at the UAH uses an approach where repeated programming of a checkerboard pattern is performed to disturb states of NAND Flash memory cells. The experiments done by Wang et al. [36] also employ the technique of partial programming. Here the bits in NAND Flash memory cells are programmed-partially until sufficient amount of noise is observed. The bit values are conditioned later to produce random numbers.

In this thesis we use partial programming to perturb state of Flash memory. This way we do not require repeated program-erase cycles or repeated program cycles that take more time and wear-down Flash memory faster.

CHAPTER 4

PROPOSED NOR FLASH TRNG

The technique for generating true random numbers from NOR Flash involves three major steps. First, we perturb the Flash memory cells using partial programming as described in Section 4.1. Second, we classify the Flash memory cells based on the values read as described Section 4.2. The partially programmed cells are repeatedly read and bits that fluctuation between states '1' and '0' are identified as Perturbed Flash Bits (PFB). A subset of these bits with the number of fluctuations exceeding a certain threshold are identified as Strongly Perturbed Flash Bit (SPFB) and the bits that do not exceed the threshold are identified as Weakly Perturbed Flash Bit (WPFB). Third, we feed the SPFBs into an algorithm that produces a random bit sequence as described in Section 4.3.

4.1 Algorithm for Perturbing Flash Cells

Figure 4.1 shows a flowchart that outlines main step in perturbing a Flash memory segment. The first step involves erasing the selected Flash memory segment. After that, each word (or byte) in the segment is partially programmed. Partial programming involves initiating a Flash program operating and aborting it before it is completed. This causes the removal of the programming voltage abruptly,

thus leaving the Flash memory cells in a perturbed state where their threshold voltage is close to the read reference voltage.

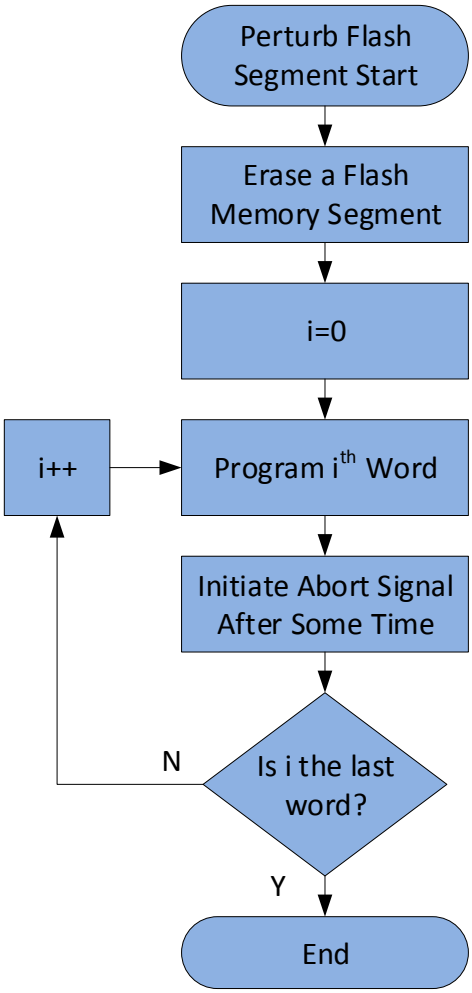

Figure 4.1. Steps for Perturbing a Flash Memory Segment

But there are certain implementation constraints concerning the approach presented in Figure 4.1. The constraints are presented below.

a. When a write operation is being performed by the Flash memory controller, the CPU is stalled and cannot issue an abort signal.

b. The time for issuing the abort signal should be carefully chosen relative to the beginning of the program cycle to maximize the number of Flash cells in the perturbed state.

As a solution to the above listed constraints, the following adjustments to the flowchart in Figure 4.1 are performed.

a. The program performing the partial programming is copied to the RAM and executed from the RAM. This way, the abort signal can be issued by the CPU after a certain delay after the beginning of the programming cycle.

b. The software delay between the moment programming cycle starts and the moment abort signal is issued can be determined by characterizing the behavior of the Flash memory.

Figure 4.2 shows the algorithm for perturbing Flash memory segment. Figure 4.3 shows a C subroutine for same operation. The selected segment at the starting address SEGMENT_ADDR is first erased (line 6 in Figure 4.3). After this step each word in the segment reads as 0xFFFF. Next, the function that carries a partial programming of a given word, *paritalProgramWord()*, is copied into the RAM (line 8 in Figure 4.3). For each word in the segment, the function for partial programming is invoked (line 12 in Figure 4.3).

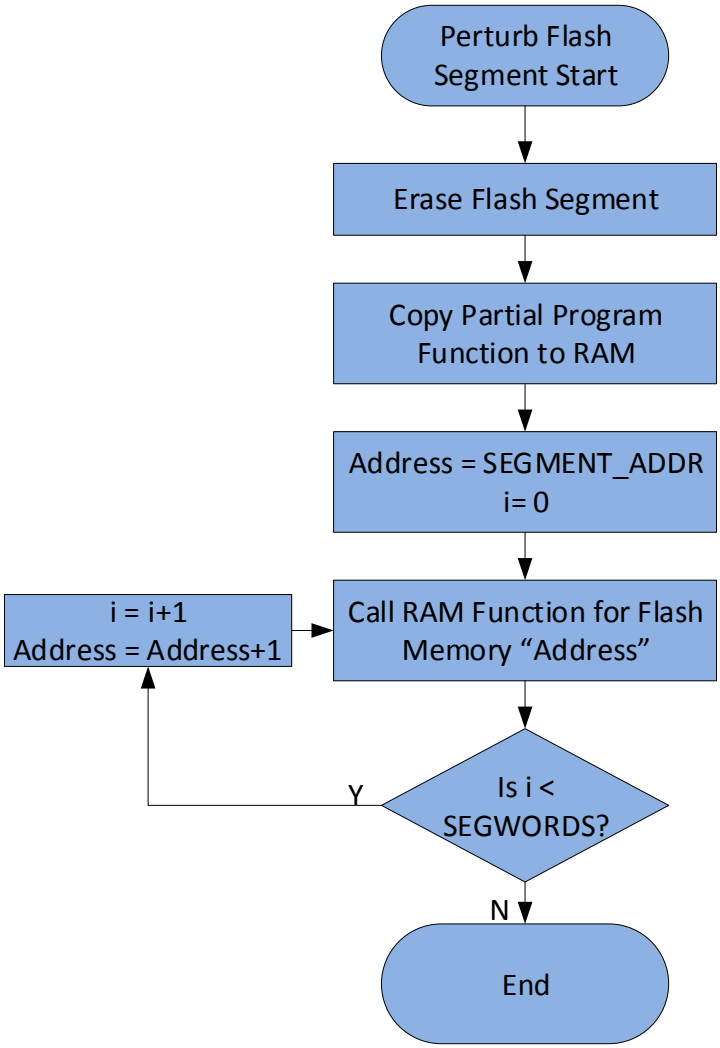

Figure 4.2 Algorithm for Perturbing Flash Memory Segment

CPU Subroutine For Perturbing a Flash Memory Segment

```
 1.   uint16_t *pFlashAdr = SEGMENT_ADDR;
 2.   void perturbFlashSegment(){
 3.     //RAM address to copy function
 4.     uint16_t *ramAddr = RAM_ADDR;
 5.     //erase the segment
 6.     eraseFlashSegment(pFlashAdr);
 7.     //copy function to RAM
 8.     copy2Ram(partialProgramWord,ramAddr);
 9.     //For each word
10.     for(uint8_t i=0;i<SEGWORDS;i++){
11.       //invoke RAM function
12.       asm("CALLA #ramAddr");
13.       //move to next word
14.       pFlashAdr++;
15.     }
16.   }
```

Figure 4.3 Subroutine for Perturbing Flash Memory Segment

Figure 4.4 shows the algorithm for partial programming operation to be performed from RAM. Figure 4.5 shows the C subroutine *partialProgramWord()* implemented to perform partial programming of a word in the Flash memory pointed to by *pFlashAdr*. The Flash memory control registers are first properly initialized for a write operation (lines 3 and 5). FCTL1 and FCTL3 are the Flash memory controller registers - FCTL1 is used to specify the operation to be performed, while FCTL3 is used to unlock the Flash memory and check the status of the Flash memory controller. The actual programming (or writing) of the word starts when a value 0 is written to the word at address *pFlashAdr* as shown in line 4. But the major objective here is to perform a partial programing. Thus, the abort signal is created by setting the emergency exit (EMEX) bit in the control register FCTL3 as shown in line 12.

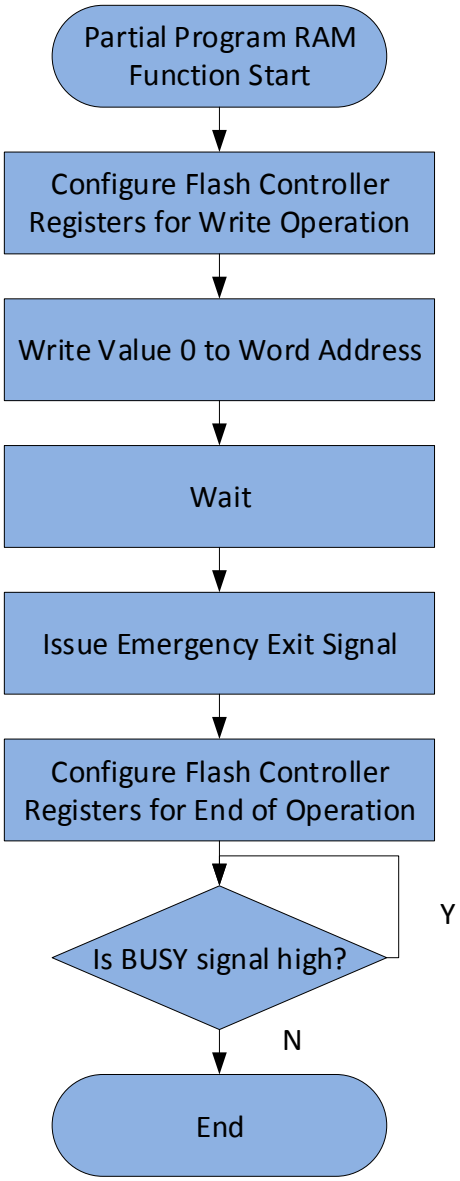

Figure 4.4 Algorithm for Partial Programming

Subroutine For Partial Programming Copied To RAM

```
 1.   void partialProgramWord(){
 2.     //clear lock bit
 3.     FCTL3 = FWPW;
 4.     //enable write
 5.     FCTL1 = FWPW+WRT;
 6.     //write 0 to the address
 7.     *pFlashAdr = 0;
 8.     //waiting..
 9.     _NOP();
10.     ...
11.     //issue emergency signal
12.     FCTL3 = FWPW+EMEX;
13.     //clear WRT bit
14.     FCTL1 = FWPW;
15.     //set lock bit
16.     FCTL3 = FWPW+LOCK;
17.     //wait while BUSY=1
18.     while(FCTL3&BUSY);
19.   }
```

Figure 4.5 Subroutine for Partial Programming

Setting the EMEX bit right after the program command would cause the program operation to be aborted early during the program cycle. This would leave all the bits in the given word in the erased state (they read as logic '0'). Issuing program abort too late would result in all the bits of the given word to be in the programmed state (they read as logic '1'). The software delay between the beginning of programming cycles and the moment emergency exit is issued is controlled by the NOP() instructions. These instructions burn time (1 NOP is equivalent to 1 processor clock cycle) and by adding a series of these instructions between lines 9 and 11 we can adjust the duration of the partial programming as shown in Figure 4.6. Alternatively, to reduce a memory footprint of the software delay code section, a series of NOP instructions can be replaced by an empty loop.

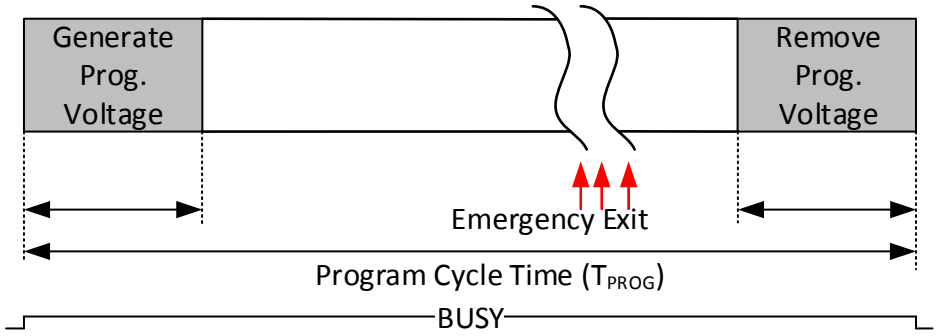

Figure 4.6 Program Cycle with EMEX Signal

4.2 Algorithm for Identifying Strongly Perturbed Flash Bits (SPFB)

After partial programming operation is completed, each of the word address is read a certain number of times. The value read is stored in a buffer. The values are read in a continuous sequence, and operation for characterizing the bit as SPFB, WPFB or none is done later. This is done so as not to interfere or obstruct the reading values from perturbed location.

After the word address is read to fill the buffer, the number of fluctuations in each of the bit position is calculated. A fluctuation is defined as the transition of the bit value from '1' to '0' or from '0' to '1'. The threshold set for classifying a Flash bit as SPFB is $1/8^{th}$ of the total number reads. If the number of fluctuations is larger than that value, the bit is considered an SPFB. If the number of fluctuations is 0, the bit is ignored. If the fluctuation value is greater than 0 but less than the threshold value, the bit is considered to be a WPFB.

Figure 4.7 describes the algorithm for determining the SPFBs and WPFBs after perturbation is performed, while the C subroutine is presented in Figure 4.8. The total number of reads that is performed is *nStep* times. The read values from the ad-

dress *(pFlashAddr+j)* are kept in the buffer at the starting address BUFFER_ADDRESS. Then each bit position value is compared with the same bit position value obtained from next read value. If these two values are same, this means that the bit has not changed its value. If they are different, then the number of fluctuations is increased. This is done by increasing the k^{th} item of the *countFluctuationArr* array.

After finding the number of fluctuations per *nStep* values read from a word address, each of the fluctuation values that are stored in array *countFluctuationArr* is checked to see if it is greater than or equal to *SPF_Threshold* (i.e. *nStep/8*). If it is, then the bit satisfies the condition for SPFB. The information about the word address as well as the bit position is kept in a global array. In case of the above algorithm, the address information is placed in *SPFBAddr* and the bit information is kept in *SPFBit*. If the number of fluctuation is less than *SPF_Threshold* value, but is greater than 0, then the bit satisfies the condition for a WPFB. The address information is kept in another global array named *WPFBAddr* and the bit information is kept in *WPFBit*.

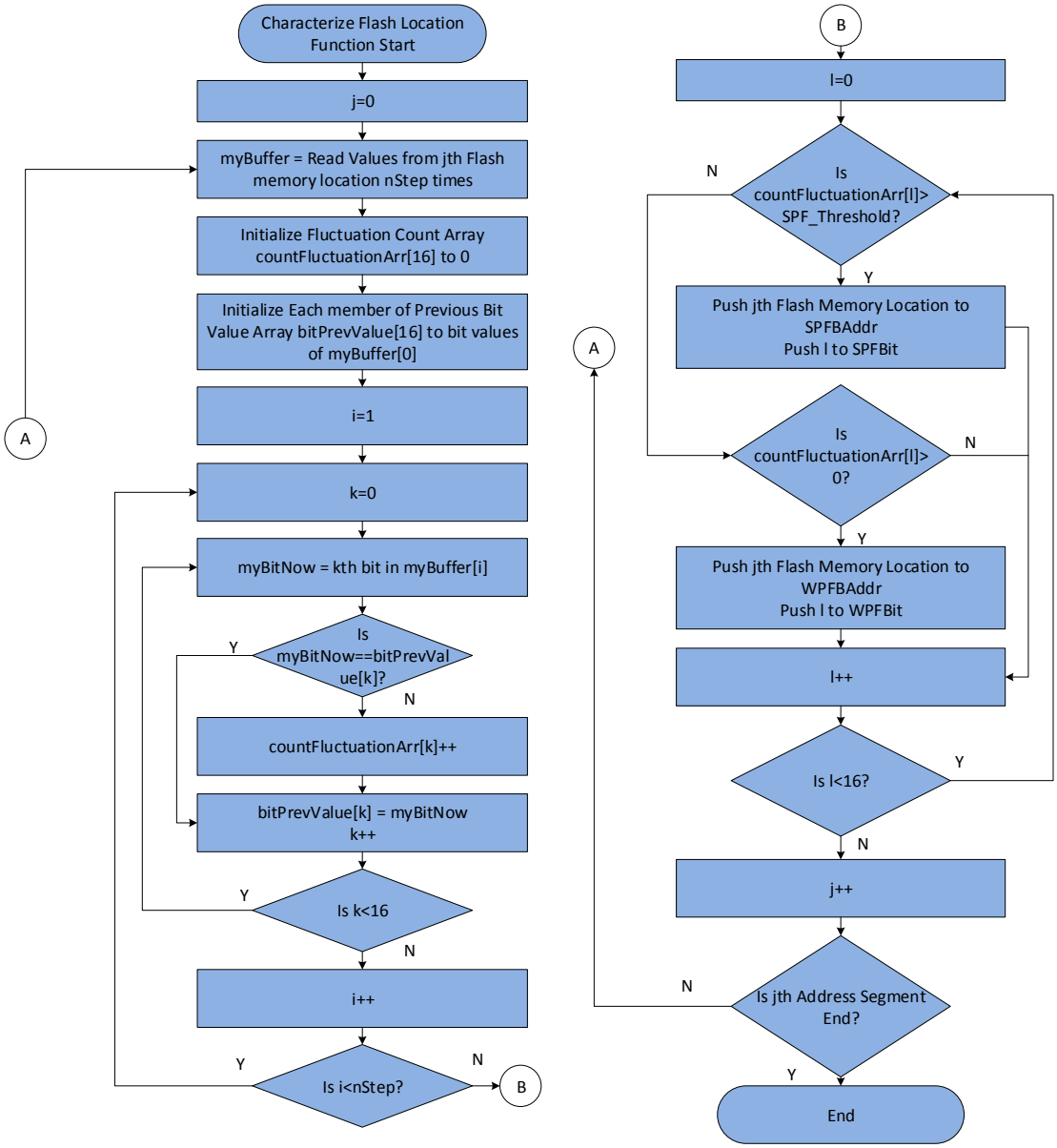

Figure 4.7 Algorithm for Determining SPFB and WPFB

Subroutine For Characterization Of Flash Bits

```
 1.   uint16_t* pFlashAdr = SEGMENT_ADDR;
 2.   //For each word in the segment
 3.   for(int j=0;j<SEGWORDS;j++){
 4.     //storage buffer
 5.     uint16_t* myBuffer = (uint16_t*)BUFFER_ADDRESS;
 6.     //reading each address nStep times
 7.     for(uint16_t i=0;i<nStep;i++){
 8.       //j is the word number in the flash segment
 9.       *myBuffer++ = *(pFlashAdr+j);
10.   }
11.     //reset storage buffer
12.     myBuffer = (uint16_t*)BUFFER_ADDRESS;
13.     //fluctuation count for each of 16 bit positions
14.     uint16_t countFluctuationArr[16] =
      {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
15.     uint8_t bitPrevValue[16];
16.
17.     //read the first value in buffer
18.     uint16_t myTempVal = *myBuffer++;
19.     //initialize the bit count
20.     for(uint8_t i=0;i<16;i++){
21.       bitPrevValue[i] = myTempVal&(0x01<<i);
22.     }
23.     //for each value in buffer
24.     for(uint16_t i = 1;i<nStep;i++){
25.       //read the value
26.       myTempVal = *myBuffer++;
27.       //for each bit in each value read
28.       for(uint8_t k=0;k<16;k++){
29.         uint8_t myBitNow = myTempVal&(1<<k);
30.         //compare if bit is equal to previous bit at same position
31.         if(myBitNow!= bitPrevValue[k]){
32.           //increase count if it is
33.           countFluctuationArr [k]++;
34.           //save for next iteration
35.           bitPrevValue[k] = myBitNow;
36.         }
37.       }
38.     }
39.     //store the address and bit
40.     for (uint8_t i=0;i<16;i++){
41.       //if they are SPFB, store them
42.       if(countFluctuationArr [i]>=SPF_Threshold){
43.         *SPFBAddr++ = pFlashAdr;
44.         *SPFBit++ = i;
45.       }else if (countFluctuationArr [i]>0){
46.         *WPFBAddr++ = pFlashAdr;
47.         *WPFBit++ = i;
48.       }
49.     }
50.   }
```

Figure 4.8. Subroutine for Determining SPFB and WPFB

4.3 Algorithm for Generating True Random Numbers

Initial algorithms to generate true random numbers considering just a single SPFB as a source of entropy failed. Thus, we opted for an algorithm that combines multiple entropy sources. In this case, the bit values read from more than one SPFB are combined using XOR operation which is equivalent to performing addition of individual bit values. Figure 4.9 illustrates a simple XOR operation on three bit sequences.

```
seq1 = [1 0 0 0 1 1 0 1 0 1 0]
seq2 = [0 1 0 0 0 1 0 0 1 0 1]
seq3 = [1 1 1 0 0 1 0 0 1 1 0]
-----
Sum  = [2 2 1 0 1 3 0 1 2 2 1]
XOR  = [0 0 1 0 1 1 0 1 0 0 1]
```

Figure 4.9 XOR Operation in Bit Sequences

Figure 4.10 shows the algorithm for generating *N*-bit random number. Figure 4.11 shows the subroutine for implementation of the algorithm. Information about *M* strongly perturbed bit in a Flash segment is kept in arrays *SPFBAddr* and *SPFBit*. *SPFBAddr* contains the address of the word that contains SPFB while the corresponding bit index is kept in array *SPFBit*.

A variable *k* is used to index the SFPB being read. Another variable *m* is used to indicate the number of SPFBs that are considered for generating a set of random sequence. If $M > m$, then out of *M* available SPFBs, *m* SPFBs are read in a sequence (indexed by *k*) *nStep* times each, the bit position indicated by respective value in *SPFBit* are XORred together and random sequence of length *nStep* is thus generated. Remaining *m* SPFBs are used to produce next *nStep* length of random

sequence and the process is continued in a cycle until random bits of desired length *N* is produced. In the case when *M*<*m*, *M* SPFBs are read in a sequence (indexed by *k*) and the bit positions indicated by respective value in *SPFBit* are read and XORred and again this process is continued in cycle starting from first SPFB address until *k*=*m*. This produces a single sequence of random bits of length *nStep*. For next random sequence of length *nStep*, the process is started from the SPFB *k* is currently pointing to.

Sorting out the bit value pointed by *SPFBit* for creation of *readVal* is not presented in algorithm and subroutine to make the illustration simple. The bit values at the respective positions can be extracted by using a bitmask once the word is read.

countFluctuation() function implementation is not shown in the subroutine presented in Figure 4.11 for simplicity. The concept can be derived from line 13-line 38 form subroutine presented in Figure 4.8. This is additional cautionary step that is adapted in the implementation to make sure that the Flash memory cell identified as SPFB still holds the property. This can be removed from the actual implementation for increasing throughput since our experiments show that once a bit is identified as SPFB, it maintains the property indefinitely.

Legend:
N: total number of random bits to generate
n: number of random bits generated until now
nStep: number of random bits generated in a single round
M: number of SPFB locations
m: number of SPFB location to use for a single round
k: variable to index SPFB being read in SPFBAddr and SPFBit
RNGVector: Array to hold random bits generated in a round
readVal: Array to hold bit vector read from SPFB
SPF_Threshold: Threshold for SPFB
outputRNGVec: Final output vector

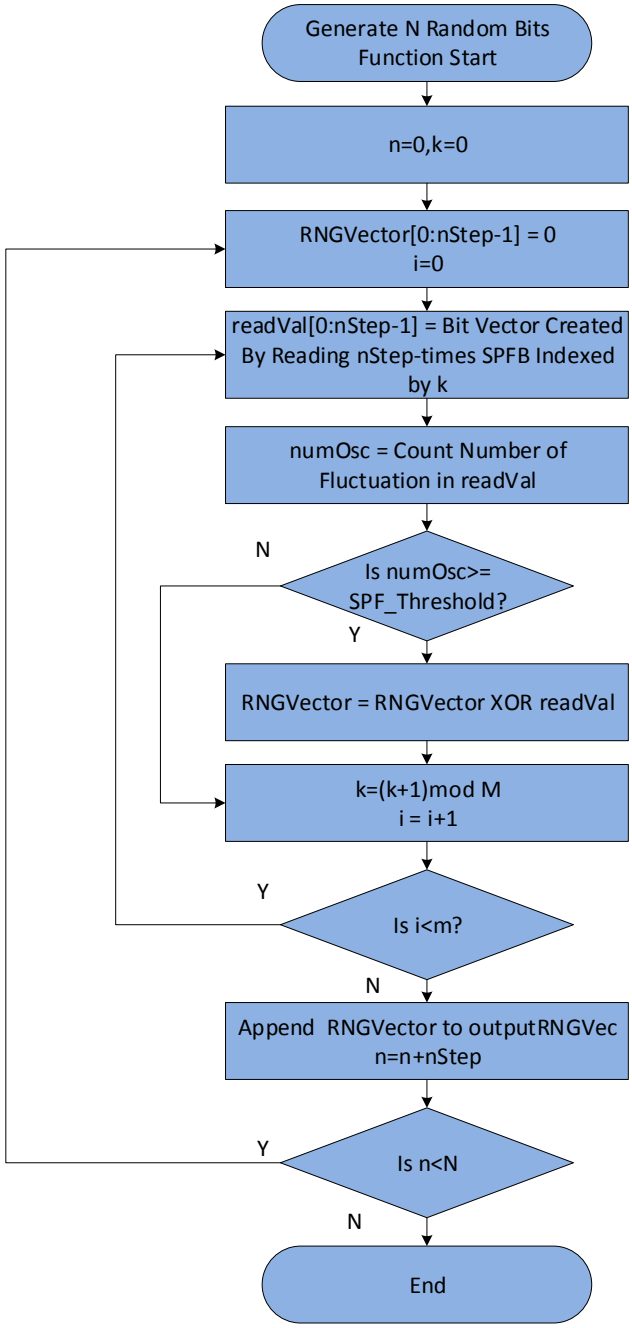

Figure 4.10 Algorithm for Generation of N-bit Random Sequence

Subroutine For Generating Random Numbers

```
1.   //generate N bit random sequence
2.   void generateRandom(N){
3.     uint8_t k=0;
4.     //generates in a chunk of nStep
5.     for(uint32_t n=0;n<N;n+=nStep){
6.       //initialize the value to 0
7.       uint16_t RNGVector[nStep-1] = 0;
8.       //for each bit position SPFB
9.       for(uint8_t i=0;i<m;i++,k=(k+1)%M){
10.        //read the kth location pointed by k in SPFBAddr and SPFBit
11.        //generate a bit vector
12.        uint16_t readVal[nStep]  = getBitVector();
13.        //check fluctuation count
14.        uint16_t numOsc = countFluctuations(readVal);
15.        if(numOsc>=SPF_Threshold) {
16.          //if still maintains fluctuation
17.          RNGVector = RNGVector XOR readVal;
18.        }
19.      }
20.      //push to output sequence
21.      outputRNGVec[n*nStep:(n+1)*nStep] = RNGVector;
22.    }
23.  }
```

Array notation arr[n:k] *is borrowed for simplicity from standard notation which is not available in C.* arr[n:k] *refers to elements of* arr *from* k *to* n-1

Figure 4.11. Subroutine for Generating *N*-bit Random Sequence

Optionally, the value obtained as the random sequence can be de-biased using Von Neumann de-biasing [40]. The algorithm for Von-Neumann de-biasing is presented in Figure 4.12 and the C subroutine is shown in Figure 4.13. Here, *inVec* is the vector that is supplied as input. *outVec* is the vector that is obtained as output. Von Neumann de-biasing works by comparing the two consecutive input bits. If the two bits are same, both of them are discarded. If they are not the same, the first one is considered as output and the second bit is discarded. Von Neumann de-biasing reduces the length of the sequence by at least half in the best case scenario, so for analysis as many data has to be gathered. However, our experiments have shown that given sufficient value of *m* i.e. the number of SPFB being considered for genera-

tion of random numbers, Von Neumann de-biasing is not necessary for quality random number generation.

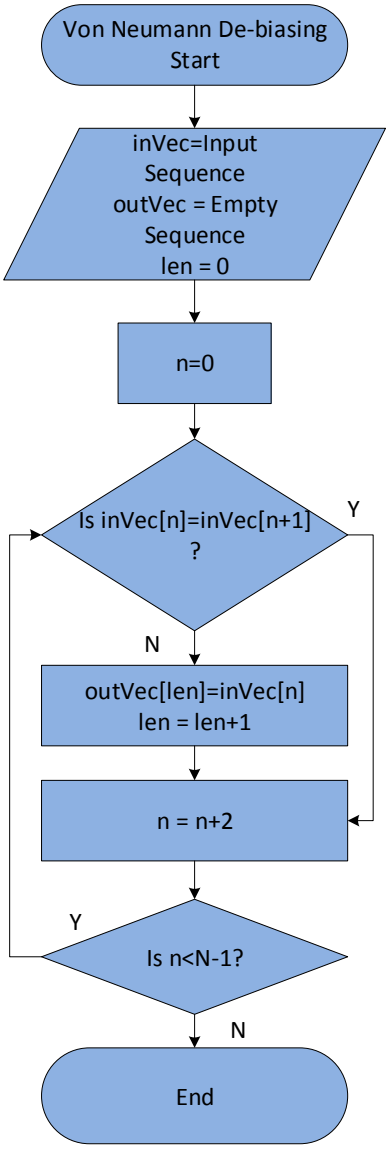

Figure 4.12 Algorithm for Von-Neumann Debiasing

Subroutine For Von Neumann De-Biasing

```
1.   void debais(uint8_t* inVec,uint8_t* outVec,uint16_t &len){
2.     //for  each bit in sequence
3.     for(uint16_t n=0;n<N;n+=2){
4.       uint16_t i=0;
5.       //check if the bit is equal to next bit
6.       if(inVec[n]!=inVec[n+1]){.
7.         //output if not equal
8.         outVec[i++] = inVec[n];.
9.       }
10.    }
11.    //output the length
12.    len = i;
13.  }
```

Figure 4.13 Subroutine for Von Neumann Be-biasing

CHAPTER 5

EXPERIMENTAL ENVIRONMENT

Experimental evaluation of the proposed Flash TRNG is performed on a TI's MSP430 family of microcontrollers. This chapter describes the experimental flow. Section 5.1 gives a system view of the experimental flow. Section 5.2 describes the experimental platform used. Section 5.3 describes the standard testing module, namely NIST Statistical Test Suite, used for verification of the generated random number sequences.

5.1 System View

Figure 5.1 shows a high level diagram of our experimental flow. An experimental board is used as a testbed. Selected segments of the MSP430's Flash memory are perturbed and then repeatedly read during characterization phase. The state of the Flash memory cells are exported through a serial communication interfaces to a workstation for further analysis. Matlab programs running on the workstation are used to characterize individual Flash memory bits and produce random sequences. This setup allows us to explore effectiveness and quality of random sequences produced by a set of candidate TRNG algorithms. The following subsections describe the experimental platform (Section 5.1.1) and characterization and verification programs running on the workstation (Section 5.1.2).

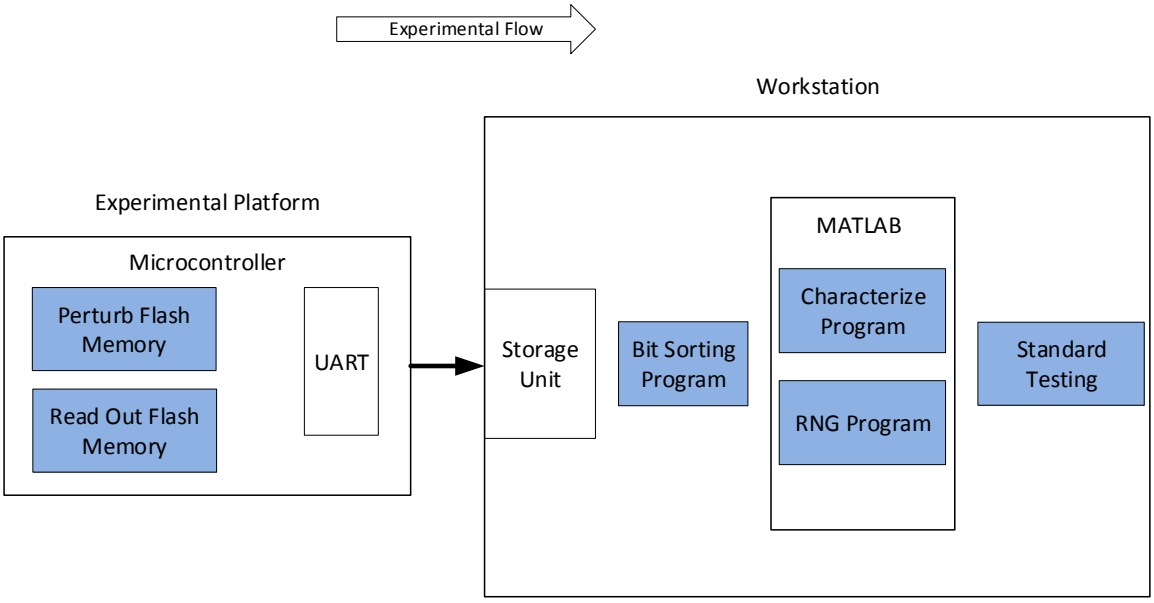

Figure 5.1 System View of Experimental Flow

5.1.1 Experimental Platform Flow

Our experimental platform consists of an experimenter board featuring an MSP430 microcontroller. A selected portion of the Flash memory in the microcontroller is deliberately taken into a perturbed state using algorithms described in Figure 4.3 and Figure 4.5. Perturbing of a Flash memory segment is a one-time process. Once perturbed Flash memory segments will remain in that state indefinitely.

A simple initial profiling mechanism is implemented in the microcontroller to identify words that contain at least one perturbed bit (this process is referred to as *Initial Profile* in Figure 5.3). Here, each word is read into a RAM buffer for a predefined number of times. The states of that word are then analyzed to identify any fluctuations in individual bits. If the sequence of read states show some fluctuations, and the number of fluctuations exceeds a certain threshold, the word is marked as a

potential good source of entropy and fluctuating bits are marked as Strongly Perturbed Flash Bits (SPFBs). One word may include one or more SPFBs.

After this step a list of SPFBs in the selected segment is created. Each SPFB is uniquely described by its word address and its bit position within the word. The next step is to traverse words with SPFBs, extract state changes of each SPFB into a bit vector, and send out the bit vector over the serial communication interface to the workstation. For example, reading a word with a single SPFB 1,024 times and extracting states of the SPFB into a bit vector creates a message that looks like the one shown in Figure 5.2. This message is sent to the workstation for further processing. Algorithms for creating true random numbers benefit from combining multiple sources of entropy, and each SPFB bit is considered a good source of entropy.

This process is repeated in a cyclic fashion so that we have at least a million bits of data which is essential for NIST test. Figure 5.3 describes of these steps on one contiguous chunk of Flush memory.

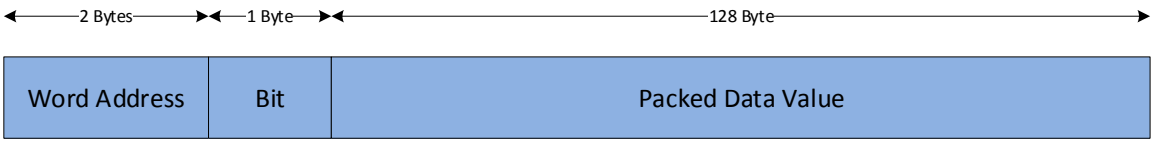

Figure 5.2 Packet Format Generated for Perturbed Bit

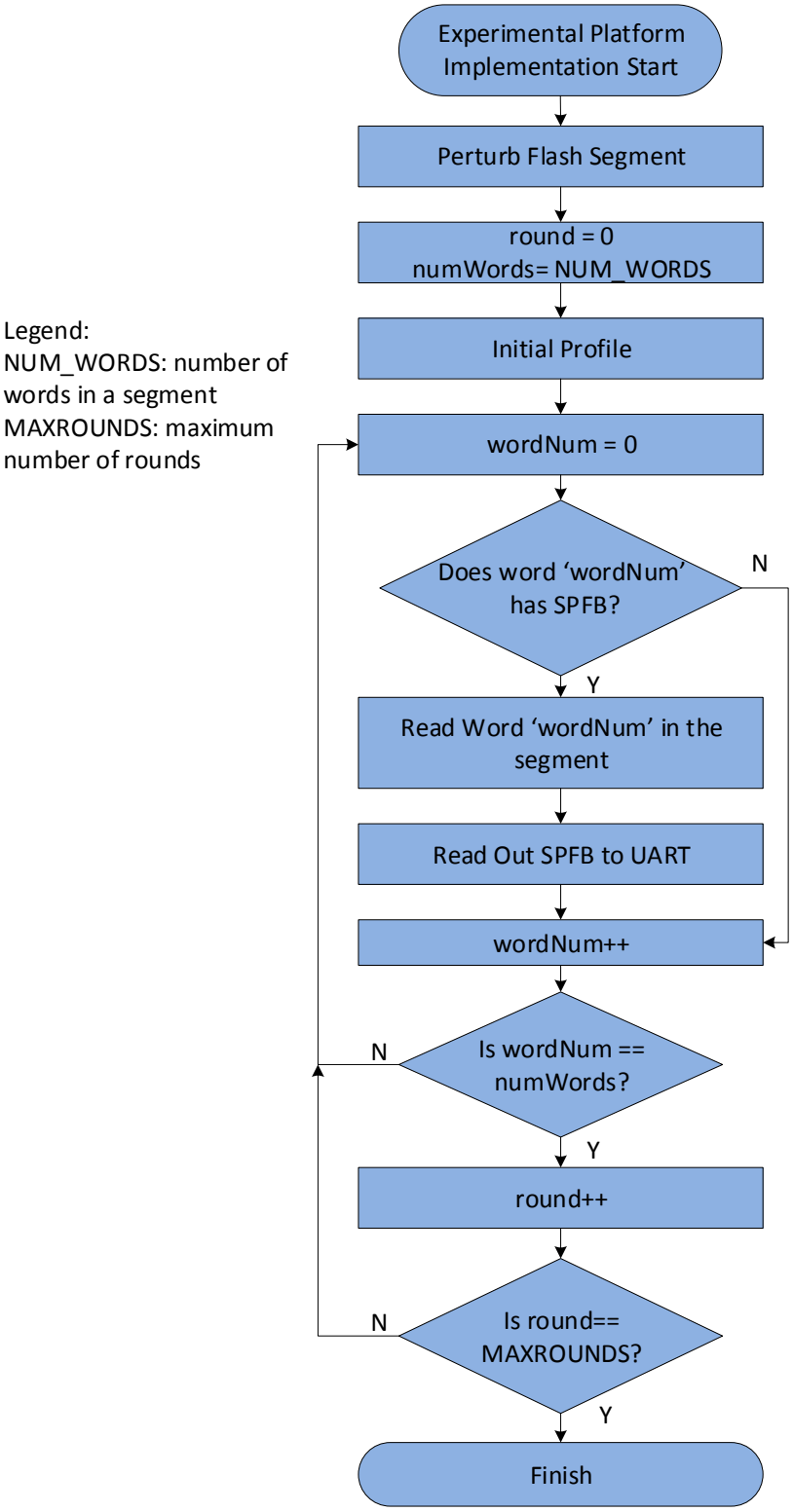

Figure 5.3 Experimental Platform Flow

5.1.2 Workstation Experiment Flow

Figure 5.4 illustrates the workstation experiment flow. The packets obtained from the experimental platform are logged into a file in the workstation. This file is read by the *Bit Sorting Program* that creates a separate file for each SPFB bit.

A characterizing program is run that determines whether the bit that is initially flagged as SPFB still meets the requirements, i.e., the number of state changes is still above a certain threshold. The files containing states of SPFBs that meet the requirement are then used in exploring various algorithms for generating true random numbers that combine information from multiple SPFBs. The last step involves testing the generated random sequences using standard tests designed by National Institute of Standard and Technology (NIST) and defined in the NIST SP 800-22 [3].

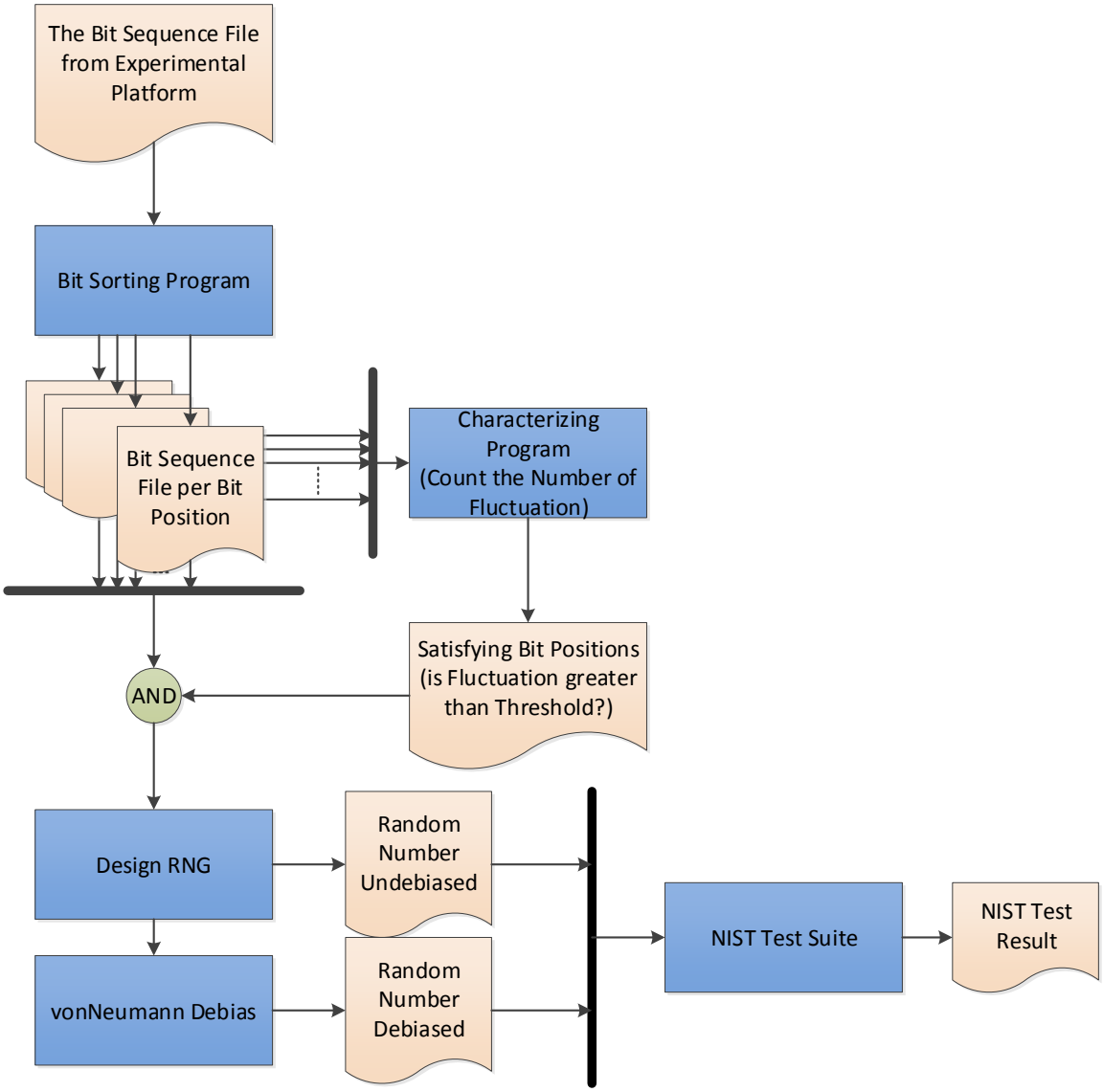

Figure 5.4 Workstation Experiment Flow

5.2 Experimental Platform

Our experimental evaluation is based on Texas Instrument's MSP430 microcontrollers. MSP430 is a family of mixed signal microcontrollers. It features 16-bit RISC CPU with on-chip SRAM, Flash memory, oscillators and peripherals like digi-

tal I/O, power management module, real time clock, watchdog timer, analog-to-digital converter, digital-to-analog converter, DMAs, LCD controllers etc.

The particular MSP430 microcontroller used in the experimental evaluation is MSP430F5438 shown in Figure 5.5(a). This microcontroller contains 256KB of Flash memory that is composed of 4 banks each of 64KB. It also has 16KB SRAM memory, timers, parallel ports and an AD controller. Figure 5.5(b) shows the TI's Experimenter EXP430F5438 board used in our experiments. It contains a 100-pin drop in socket allowing quick changes of microcontroller chips.

(a) Microcontroller MSP430F5438 (b) TI Experimenter's Board EXP430F5438

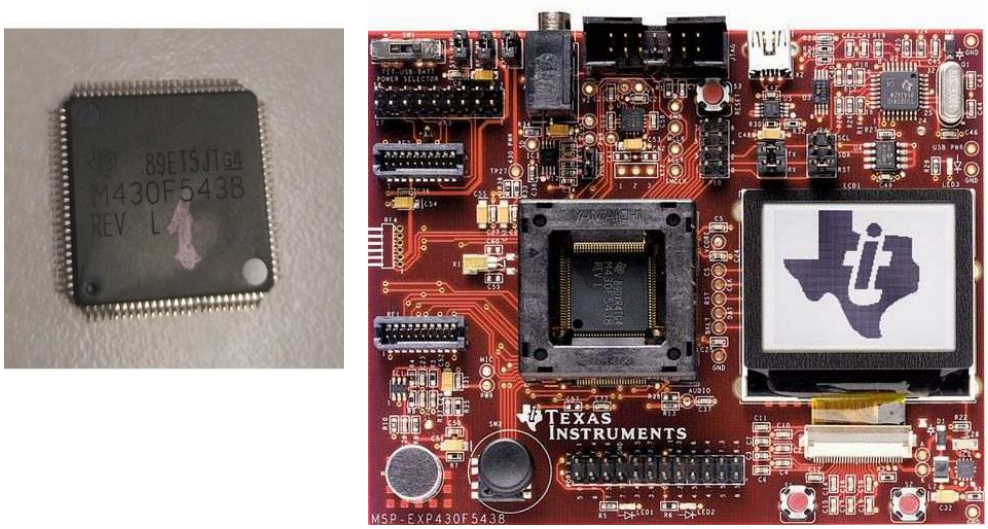

Figure 5.5 Experimental Platform Used

(a) Microcontroller MSP430F5438 (b) TI Experimenter Board

5.3 NIST Tests

National Institute of Standards and Technology (NIST) defines a suite of tests [3] that can be applied to a sequence of random bits to test whether they meet the expectations for truly random numbers. The suite consists of a number of tests

and the following section gives a short description of individual tests. These test treat randomness as probabilistic property and try to find the probability that the sequence shows randomness.

Each of these tests assume a Null Hypothesis that the sequence provided is random. Correspondingly the alternate hypothesis is that the sequence is not random. A parameter based on the sequence is calculated. If it is higher than a predefined critical value, Null Hypothesis is rejected. Otherwise, Null Hypothesis is accepted. However, these are internal to the test. For a user's point of view, each of the test will output a *P-Value*. Each *P-Value* denotes the probability that a truly random generator will generate a sequence less random than the sequence tested. The higher the *P-Value* is, the better. For the case of NIST test suite, a sequence is considered to be sufficiently random if the sequence produces a *P-Value* greater than 0.0001.

The source code for each of these tests is available publicly [41], and thus is easier to download, compile and run for test purpose.

The 15 tests that NIST defines are discussed in brief in the following section.

1. The Frequency Test

The Frequency Test, also called monobit test, determines the proportion of '1's and '0's in the sequence of number produced. As the requirement in each random number generator is to produce equal number of '1's and '0's, this test is the first and basic test on which all other tests depend. Internally, this test converts 0s to -1s and 1s to 1s and calculates the sum of the bits in sequence. Closeness of this value to 0 defines the quality of sequence produced.

2. Frequency Test within Block

Random number generators should produce sequence in which number of '1's and '0's should be equal. This not only applies for the total sequence but also applies to any chosen sub-sequence. Frequency Test within a Block tests this property. It tests if the frequency of 1s in a sequence of *M* bit sequence is *M/2*. Here the total sequence of length *n* is divided into *floor(n/M)* non-overlapping sequences. Here *floor(x)* means highest whole number lower than *x*. Proportion of 1s in each of the sequence is calculated, and the Chi-Squared statistic value is calculated. Using in-complete gamma function *igmac*, *P-Value* is calculated.

3. Runs Test:

A Run is a continuous sequence of a single bit that is bounded on both sides by bit of opposite value. The Runs test finds the runs of 1s and 0s in a sequence and determines if the transition from runs is fast or slow.

4. Test of Longest Runs of Ones:

The purpose of this test is to check the longest run of ones in an *M*-bit block in the sequence generated. This test checks if the longest runs of 1s is consistent with the longest runs of ones in a sequence that is truly random. Each *M*-bit block is categorized in the basis of the longest runs of ones and Chi-Squared statistic value is calculated. *P-value* is calculated using incomplete gamma function.

5. Binary Matrix Rank Test:

The purpose of this tests is to determine the linear dependency among the substrings in the original sequence. The sequence is divided into matrices of certain defined dimension. Rank of each matrices is calculated. Based on these rank, matri-ces are categorized. Chi-Squared statistic value is calculated and *P-value* based on this is calculated.

6. Discrete Fourier Transform Test:

This test detects the periodic features in the sequence. This test detects if the number of peaks in DFT that reach higher than or equal to 95% is significantly different than 5%.

7. Non-Overlapping Template Matching Test:

This test checks for the occurrence of a periodic sequence. Here a window of certain length is used to search bit-pattern of same length. If the pattern is not found, the window slides one bit position. Else, the window is reset to bit after the pattern found, and new search begins.

8. Overlapping Template Matching Test:

This test also uses a window of certain length to find the matching pattern in the sequence. The difference between the Non-Overlapping Template Matching Test and this test is that the window slides only one-bit position if the matching pattern is found in this case.

9. Maurer's Test (Universal Test):

This test checks if the sequence can be compressed significantly without loss of information. The whole sequence is divided into two segments, an initialization segment and a test segment. A sum value is calculated based on the repetition of the certain bit-length values in sequence. This sum is used to calculate test statistic, and thus calculate *P-value*.

10. Linear Complexity Test:

This test finds the complexity of the sequence by finding the length of Linear Feedback Shift Register (LFSR). If the length if too short, it means non-randomness. The original sequence is broken into certain number of blocks of specified length

each. Using Berlekamp-Massey algorithm, the linear complexity value of each of the sequence is calculated. The mean value is calculated, and test statistic is calculated from which *P-value* is calculated

11. Serial Test:

This test determines whether the number of occurrences of the 2^m m-bit overlapping patterns is approximately the same as would be expected for a random sequence. Here, the original sequence is padded with first *m-1* bits to the end, and the frequency of all possible overlapping *m*-bit block is determined. Frequency of possible overlapping *m-1* bits and *m-2* bits also calculated. Using these frequency values, test statistic is generated and *P-value* is calculated.

12. Approximate Entropy Test:

This test is similar to Serial Test, but in this test the frequency of the overlapping blocks of consecutive length are compared with the expected truly random sequence.

13. Cumulative Sum Test:

Cumulative sum for each position is calculated in forward and backward direction after the sequence is converted to a sequence of -1 and 1 replacing 0 with -1. Test statistic is computed as the largest absolute cumulative sum value. This is used to compute the *P-value*. The goal of this test is to see if the cumulative sum of partial sequence is too large or too small as compared to a truly random sequence.

14. Random Excursion Test:

This test checks the number of cycles having exactly K visits in a cumulative sum random walk. Here, the 0 in original sequence is replace by -1, and cumulative sum is calculated. The sequence is padded with 0s at the either end and the number

of zero crossing is calculated, which gives the number of cycles. This test consists of series of eight tests.

15. Random Excursion Variant Test:

This test checks the number of visits to a particular state in a cumulative sum random walk, and detect deviation from what is expected of a truly random sequence. This test is a series of eighteen sub-tests.

CHAPTER 6

RESULTS

In order to properly identify the time for generating a maximum number of SPFBs, we first characterize the Flash memory in microcontroller chips. Section 6.1 presents the results of this characterization. In Section 6.2, we present the results obtained from the standard NIST Statistical Tests on the generated random sequences. The performance of our technique focusing on throughput and size of our algorithm implementations are discussed in Section 6.3.

6.1 Characterization of Flash Memories for Perturbed States

The first task in generating perturbed Flash memory bits is to identify the appropriate time for issuing EMEX signal. The goal is to obtain as much perturbed Flash memory bits as possible. Section 6.1.1 presents the data and discussion regarding optimal duration of partial programming. Section 6.1.2 discusses characteristics and types of perturbed Flash bits, whereas Section 6.1.3 describes characteristics of perturbed Flash segments as a function of processor clock frequency.

6.1.1 Characterization of Partial Programming Duration

The time for issuing emergency exit signal for partial programming is important component in maximizing the number of perturbed Flash bits. By setting

the EMEX bit too soon results in having all Flash bits in the erased state. By setting the EMEX bit too late results in having all Flash bits in the programmed state. Our ability to control duration of partial programming is a function of the processor clock frequency. By increasing the processor frequency, we increase time resolution with which we can abort the ongoing Flash program operation.

Table 6.1 shows the number of SPFB and WPFB bits in sample chip 1 when the processor is running at frequency F_{CPU}=1,048,576 Hz. Each segment consists of 256 words each of 16 bits. The time column indicates the duration of partial programming i.e. the time after which the EMEX bit is set. Clock Cycles column indicates the number of processor clock cycles corresponding to the duration of partial programming. With increase number of clock cycles, there is a sudden increase the number of bits that qualify as SPFB and WPFB. But this is only true for one instance. Increasing the time again would cause all the bits to be in programmed state, thus giving no bits that qualify as SPFB or WPFB. But the condition is slightly different when we increase the processor frequency.

Table 6.2 shows the number of SPFB and WPFB in sample chip 1 when the processor is running at frequency F_{CPU}=4,194,304 Hz. There is an increase in the number of bits that qualify as SPFB and WPFB with increase in partial programing time. This trend continues for two more clock cycles. After the peak value is reached, the number of bits decreases. Further increase in the time would cause all the bits to be in programmed state thus giving no bits in SPFB and WPFB state.

Table 6.3 shows the number of SPFB and WPFB for sample chip, but the processor is running at frequency F_{CPU}=8,388,608 Hz. Here the duration for which the number of SPFB and WPFB continue to increase for a larger number of clock cycles.

But the trend of reducing the number of SPFB and WPFB from the peak point can also be seen here. This indicates that with the increase in clock cycles, the number of bits that reaches the programmed state also increases. After certain number of clock cycles, all the bits will be at programmed state, thus the number of SPFB and WPFB is noticed as 0.

In data presented for F_{CPU}=4,194,304 Hz and F_{CPU}=8,388,608 Hz, same number of clock-cycles and thus same value in time column are presented multiple times for some cases. This is because the time is controlled using a *for* loop and additional *NOP* commands following the *for* loop. This results in same number of clock-cycles for multiple cases. Although we have same clock-cycles at two such instances, the number of SPFBs and WPFBs are different in these cases essentially acting as being different by at least one clock-cycle.

The distribution of SPFBs and WPFBs follow similar trend for sample chip 2 as well. Table 6.4 presents the data for sample chip 2 when the clock is running at frequency F_{CPU}=1,048,576 Hz. The number of bits observed in this case are relatively small than that observed for other sample chips. But the trend of sudden increase in number of SPFB and WPFB can be seen here as well. Table 6.5 presents the number of SPFB and WPFB for sample chip 2 when the processor clock frequency is F_{CPU}=4,194,304 Hz. Table 6.6 presents the number of SPFB and WPFB for sample chip 2 when the processor clock is running at F_{CPU}=8,388,608 Hz.

Observation similar to sample chip 1 and sample chip 2 can be seen for sample chip 3. Data for sample chip 3 is presented in Table 6.7 for processor running at frequency F_{CPU}=1,048,576 Hz, Table 6.8 for processor running at frequency

F_{PCU}=4,194,304 Hz and Table 6.9 for processor running at frequency F_{PCU}=8,388,608 Hz.

Table 6.1 SPFB and WPFB count for Sample Chip 1 at 1,048,576 Hz

Sample Chip1 at 1,048,576 Hz																	
Time	Clock-	Seg 0		Seg 1		Seg 2		Seg 3		Seg 4		Seg 5		Seg 6		Seg 7	
(µs)	cycles	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB
24.80	26	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
25.75	27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
26.70	28	85	100	79	118	85	93	89	132	102	100	85	94	75	94	59	100
27.66	29	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 6.2 SPFB and WPFB count for Sample Chip 1 at 4,194,304 Hz

Sample Chip1 at 4,194,304 Hz																	
Time	Clock-	Seg 0		Seg 1		Seg 2		Seg 3		Seg 4		Seg 5		Seg 6		Seg 7	
(µs)	Cycles	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB
19.31	81	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19.55	82	1	2	0	0	1	1	3	1	1	2	1	1	2	4	3	1
19.79	83	44	44	16	22	40	41	56	73	36	44	42	44	53	63	42	47
20.02	84	159	189	162	208	148	208	160	193	166	202	176	200	140	198	164	214
20.02	84	90	130	28	43	32	64	41	48	39	41	58	52	35	49	34	34
20.26	85	6	6	3	5	7	7	3	3	4	7	2	2	5	4	3	5
20.50	86	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0

Table 6.3 SPFB and WPFB count for Sample Chip 1 at 8,388,608 Hz

Sample Chip1 at 8,388,608 Hz																	
Time	Clock-	Seg 0		Seg 1		Seg 2		Seg 3		Seg 4		Seg 5		Seg 6		Seg 7	
(µs)	Cycles	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB
15.61	131	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15.73	132	3	2	2	2	0	5	2	3	4	1	2	5	2	7	2	4
15.73	132	142	171	157	184	168	187	148	170	131	189	156	191	145	198	147	184
15.84	133	153	189	126	186	145	193	154	187	125	183	156	208	160	191	139	200
15.97	134	91	114	89	144	93	110	106	129	96	121	91	102	108	122	94	109
16.09	135	38	48	36	35	32	37	28	46	43	38	34	36	31	35	27	35
16.21	136	9	5	4	10	8	14	9	15	5	10	5	6	10	11	6	3
16.33	137	2	4	6	4	2	4	3	7	2	6	2	0	2	1	1	2
16.45	138	0	1	0	0	1	0	1	1	1	2	0	0	1	1	0	0
16.57	139	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 6.4 SPFB and WPFB count for Sample Chip2 at 1,048,576 Hz

Sample Chip2 at 1,048,576 Hz																	
Time	Clock-	Seg 0		Seg 1		Seg 2		Seg 3		Seg 4		Seg 5		Seg 6		Seg 7	
(µs)	Cycles	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB
24.80	26	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
25.75	27	3	4	5	9	4	7	1	2	4	7	3	2	1	1	0	2
26.70	28	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 6.5 SPFB and WPFB count for Sample Chip2 at 4,194,304 Hz

Sample Chip2 at 4,194,304 Hz																	
Time	Clock-	Seg 0		Seg 1		Seg 2		Seg 3		Seg 4		Seg 5		Seg 6		Seg 7	
(μs)	Cycles	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB
18.59	78	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18.84	79	97	107	85	104	104	116	90	140	84	113	141	157	99	132	119	127
19.07	80	41	61	53	75	64	79	48	59	42	68	53	65	40	46	34	43
19.31	81	2	3	7	6	4	9	2	3	3	9	0	3	2	1	0	1
19.55	82	1	2	2	1	0	2	0	0	1	2	0	0	0	1	0	0
19.79	83	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20.02	84	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0

Table 6.6 SPFB and WPFB count for Sample Chip2 at 8,388,608 Hz

Sample Chip2 at 8,388,608 Hz																	
Time	Clock-	Seg 0		Seg 1		Seg 2		Seg 3		Seg 4		Seg 5		Seg 6		Seg 7	
(μs)	Cycles	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB
14.90	125	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15.02	126	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0
15.02	126	135	154	142	208	140	179	144	194	134	197	136	158	163	184	148	178
15.13	127	81	94	124	128	93	119	71	88	104	116	75	97	79	106	81	95
15.25	128	20	25	37	44	28	42	21	30	29	35	23	30	18	24	11	19
15.38	129	5	7	13	16	9	10	7	5	8	18	8	3	3	7	6	5
15.49	130	3	4	3	7	1	3	1	2	5	6	1	2	0	3	0	1
15.61	131	2	0	3	2	2	3	1	0	0	2	0	1	1	0	0	0
15.73	132	0	0	2	1	0	0	0	0	0	0	0	1	1	0	0	0
15.73	132	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 6.7 SPFB and WPFB count for Sample Chip3 at 1,048,576 Hz

Sample Chip3 at 1,048,576 Hz																	
Time	Clock-	Seg 0		Seg 1		Seg 2		Seg 3		Seg 4		Seg 5		Seg 6		Seg 7	
(μs)	Cycles	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB
25.75	27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
26.70	28	13	12	10	10	4	15	4	12	8	12	9	19	14	10	20	20
27.65	29	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 6.8 SPFB and WPFB count for Sample Chip3 at 4,194,304 Hz

Sample Chip3 at 4,194,304 Hz																	
Time	Clock-	Seg 0		Seg 1		Seg 2		Seg 3		Seg 4		Seg 5		Seg 6		Seg 7	
(μs)	Cycles	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB
22.64	95	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
22.88	96	3	2	1	5	1	1	3	2	4	113	141	157	99	132	119	127
23.12	97	135	139	135	192	133	167	152	202	146	68	53	65	40	46	34	43
23.36	98	5	12	0	3	11	9	4	5	3	9	0	3	2	1	0	1
23.6	99	0	0	0	0	0	1	0	0	0	2	0	0	0	1	0	0
23.84	100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 6.9 SPFB and WPFB count for Sample Chip3 at 8,388,608 Hz

Sample Chip3 at 8,388,608 Hz																	
Time	Clock-	Seg 0		Seg 1		Seg 2		Seg 3		Seg 4		Seg 5		Seg 6		Seg 7	
(μs)	Cycles	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB	SPFB	WPFB
23.24	195	0	0	0	1	0	0	0	0	0	0	1	1	0	1	1	1
23.36	196	5	4	3	7	5	5	7	5	7	7	4	7	1	11	4	6
23.48	197	39	54	46	53	46	58	23	58	23	33	32	38	42	50	47	65
23.6	198	133	185	159	211	150	179	119	152	119	152	160	196	125	135	151	194
23.72	199	75	92	52	63	110	129	44	56	44	56	52	62	50	73	57	71
23.84	200	15	35	10	7	15	16	16	14	16	14	13	17	13	16	20	23
23.96	201	0	1	4	6	3	5	4	2	4	2	0	0	0	0	1	0

The number of SPFB and WPFB in a particular segment of a sample chip increases with increase in frequency. This is one important observation that can be noticed in data presented from Table 6.1-Table 6.9. It can be concluded that the increase in frequency allows finer time resolution when controlling the duration of partial programming. This also allows more bits to attain the perturbed state thus increasing giving larger number of SPFB and WPFB as the frequency is increased.

Table 6.1-Table 6.9 also indicate that the total number of SPFB is similar over multiple segments in a single chip. This case is true for all the tested frequencies. Table 6.10, Table 6.11 and Table 6.12 summarizes the number of SPFB and total number of SPFB and WPFB, jointly referred to as Perturbed Flash Bits (PFB) for sample chip1, sample chip 2 and sample chip 3 respectively. These data show that there exists a optimal time to achieve the best result in terms of the number of

SPFB for a chip at a particular frequency regardless of the segment. This indicates that the number of SPFB (and WPFB) is actually a function of partial programming time and the chip itself. However, the optimal time to achieve the maximum number of SPFB and WPFB for a single chip over different frequencies are different. The MSP430 family, although reveal the internal timing details to software interface, do not reveal this information in case of MSP4305438 which is used in our work. Thus, according to our result we conclude that the optimal time to achieve the maximum number of SPFB is the function of time after which emergency exit bit is set, clock frequency and the chip.

Table 6.10 Number of PFBs in Sample Chip1

Clock rate [Hz]	Partial program time		Average number of perturbed bits ± stdev	
	#CPU Cycles	[µs]	SPFB	Total (SPFB+WPFB)
1,048,576	26	24.80	0±0.0	0±0.0
	27	25.75	0±0	0±0
	28	26.70	82±12.3	186±19.8
	29	27.66	0±0	0±0
4,194,304	81	19.31	0±0.0	0±0.0
	82	19.55	2±1.1	3±1.8
	83	19.79	41±12.1	88±26.9
	84	20.02	159±11	361±14.3
	84	20.02	45±20.4	102±49.5
	85	20.26	4±1.7	9±3.3
	86	20.50	0±0	0±0.5
8,388,608	131	15.61	0±0.0	0±0
	132	15.73	2±1.1	6±1.6
	132	15.73	149±11.1	334±15.3
	133	15.84	145±13.6	337±18.7
	134	15.97	96±7.1	215±16.1
	135	16.09	34±5.3	72±7.8
	136	16.21	7±2.3	16±5.4
	137	16.33	3±1.5	6±3.2
	138	16.45	1±0.5	1±1.1

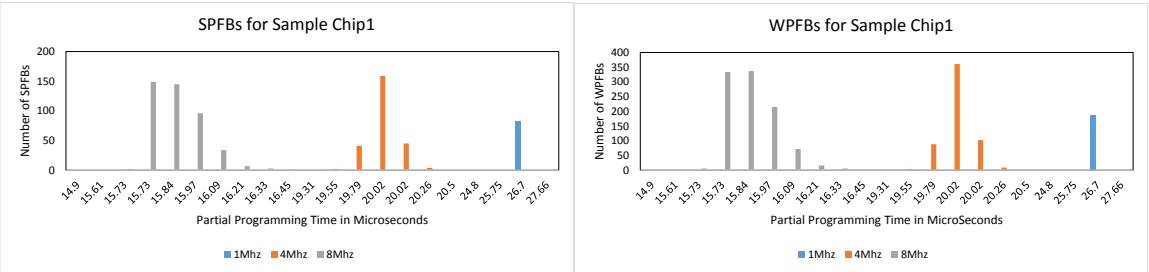

Figure 6.1 SPFBs and WPFBs distribution over time for Sample Chip 1

Table 6.11 Number of PFBs in Sample Chip2

Clock rate [Hz]	Partial program time		Average number of perturbed bits ± stdev	
	#CPU Cycles	[µs]	SPFB	Total (SPFB+WPFB
1,048,576	26	24.80	0±0.0	0±0.0
	27	25.75	3±1.8	7±4.6
	28	26.70	0±0.0	0±0.0
4,194,304	78	18.59	0±0.0	0±0.4
	79	18.84	102±19.3	227±34.5
	80	19.07	47±9.6	109±21.4
	81	19.31	3±2.3	7±5.0
	82	19.55	1±0.8	2±1.4
	83	19.79	0±0.0	0±0.0
8,388,608	125	14.90	0±0.0	0±0.0
	126	15.02	0±0.5	0±0.5
	126	15.02	143±9.5	324±22.7
	127	15.13	89±17.8	194±31.3
	128	15.25	23±7.9	55±16.4
	129	15.38	7±3.0	16±7.5
	130	15.49	2±1.8	5±3.7
	131	15.61	1±1.1	2±1.9
	132	15.73	0±0.7	1±1.1
	132	15.73	0±0	0±0

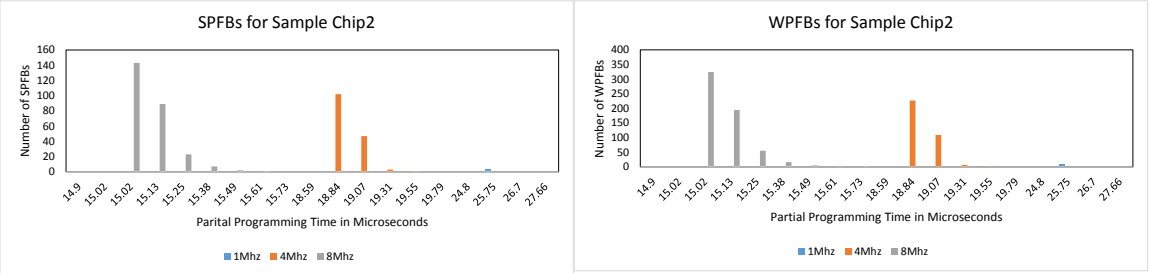

Figure 6.2 SPFBs and WPFBs distribution over time for Sample Chip 2

Table 6.12 Number of PFBs in Sample Chip3

Clock rate [Hz]	Partial program time		Average number of perturbed bits ± stdev	
	#CPU Cycles	[µs]	SPFB	Total (SPFB+WPFB
1,048,576	27	25.75	0±0.0	0±0.0
	28	26.70	10±5.3	24±7.5
	29	27.66	0±0.0	0±0.0
4,194,304	95	22.65	0±0.0	0±0.0
	96	22.89	2±1.3	4±1.6
	97	23.13	135±19.1	301±32.2
	98	23.37	5±3.0	10±5.6
	99	23.60	0±0.4	0±0.5
	100	23.84	0±0.0	0±0.0
8,388,608	195	23.25	0±0.4	1±0.8
	196	23.37	5±2.0	11±1.7
	197	23.48	38±9.1	88±18.9
	198	23.60	142±15.4	324±39.2
	199	23.72	63±21.6	139±44.7
	200	23.84	14±3.1	33±9.9
	201	23.96	2±1.9	4±4.1

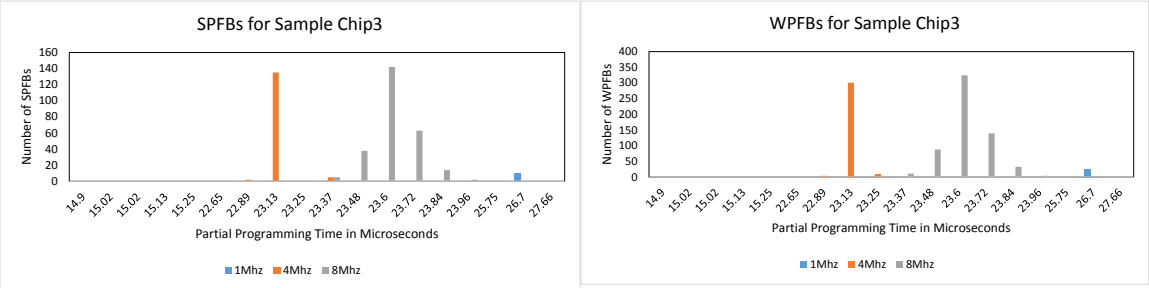

Figure 6.3 SPFBs and WPFBs distribution over time for Sample Chip 3

6.1.2 Characterizing Bits: SPFB or WPFB

Figure 6.4 shows the state of a sample WPFB as a function of the read number. The bit is biased towards logic '0'. Figure 6.5 shows another sample WPFB bit that is biased towards logic '1'. For the SPFB bit shown in Figure 6.6, the fluctuation

of the state is very high, whereas the WPFBs shown in Figure 6.4 and Figure 6.5 are biased and the number of state changes is relatively small.

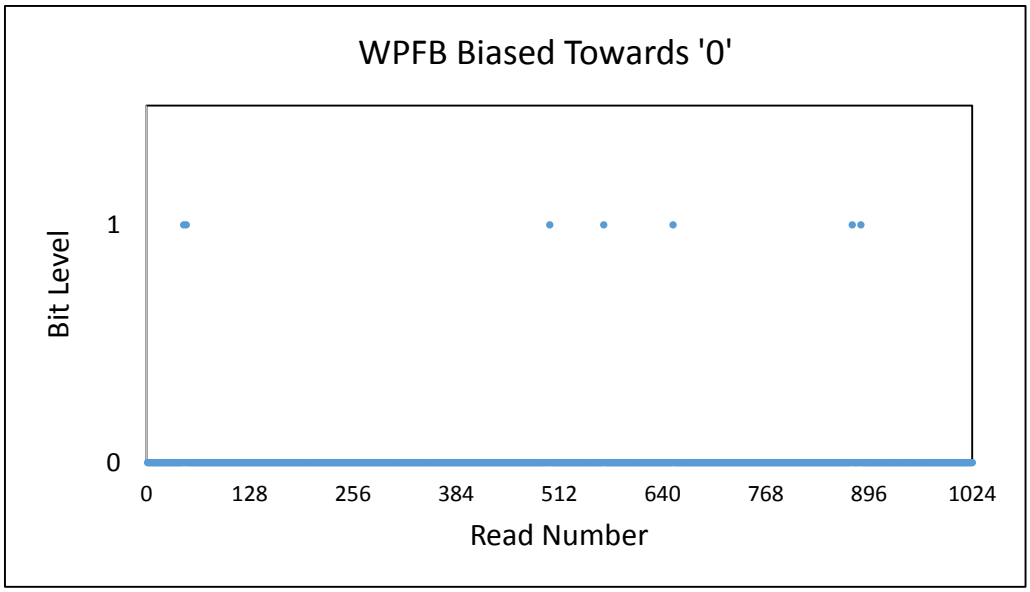

Figure 6.4 Sample WPFB Bit Biased Towards '0'

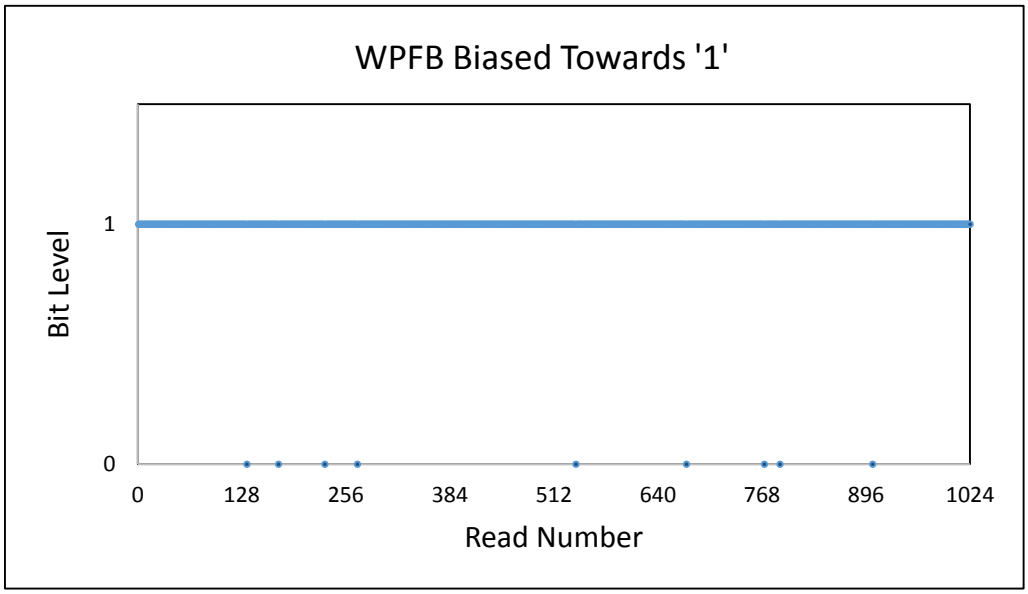

Figure 6.5 Sample WPFB Biased Towards '1'

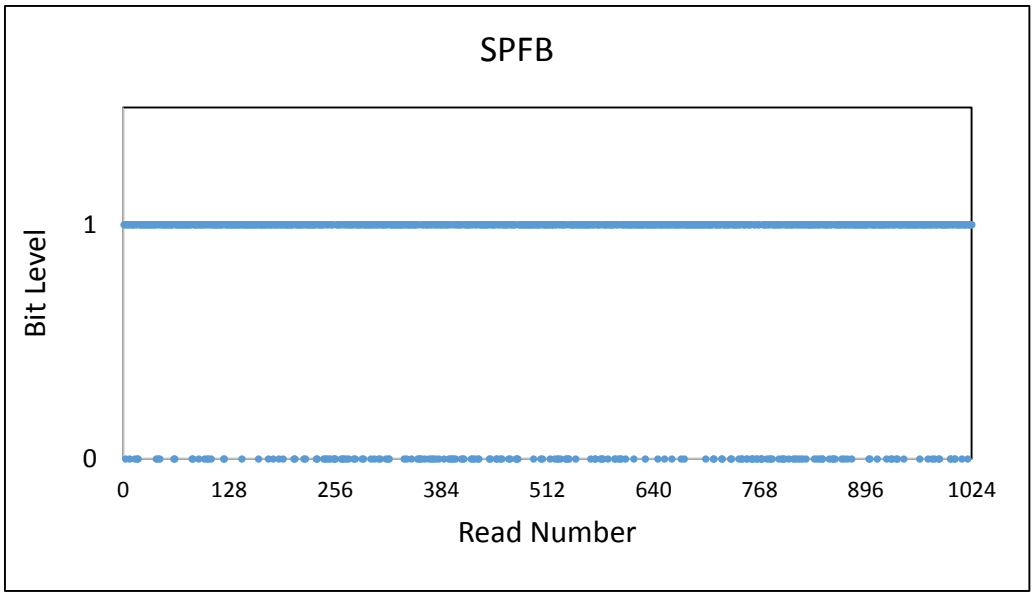

Figure 6.6 A Sample SPFB Bit

Figure 6.7 shows the number of state changes for several exemplary perturbed bits as a function of read count. The number of state changes (fluctuations) is counted for every 256 states of the perturbed bit read from the bit position. In our experiments, we set the SPFB threshold to 1/8th of the total number of reads, that is, a perturbed bit is considered strongly perturbed if the number of state changes is at least 1/8th of the total number of reads. The blue line in each of the diagram is the minimum threshold value for being considered as SPFB. For perturbed bits characterized in Figure 6.7(a) and Figure 6.7(b), the number of state changes is always below the threshold, so these bits are just WPFBs. For perturbed bits characterized in Figure 6.7(c) and Figure 6.7(d), the number of state changes occasionally dips below the threshold for brief periods of time. Finally, for perturbed bits characterized in Figure 6.7(e) and Figure 6.7(f), the number of state changes is always above the threshold, so these bits are just SPFBs.

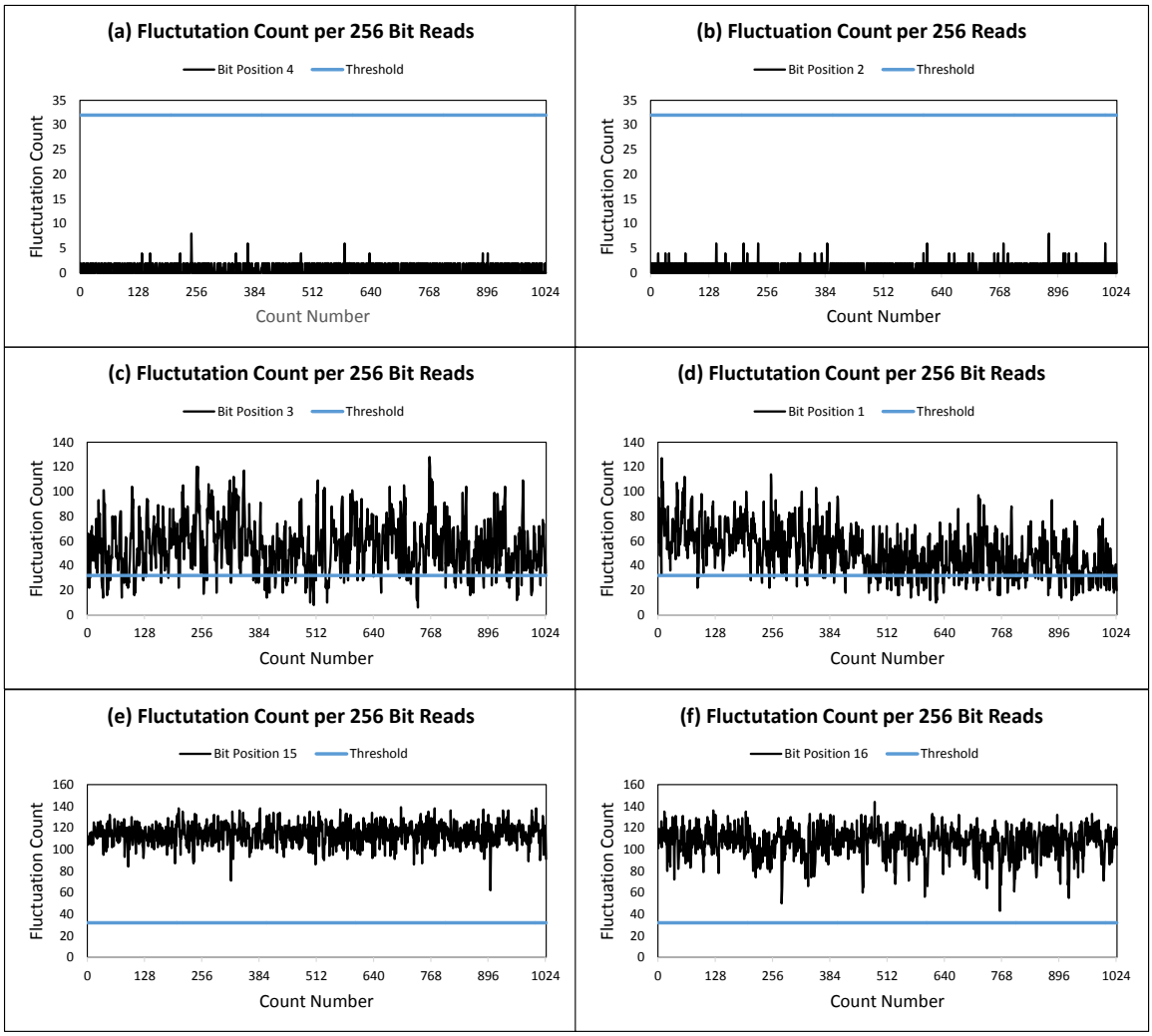

Figure 6.7 Fluctuation Count for Different Perturbed Flash Bits

6.1.3 Characterization of Flash Segment

Each Flash memory segment in MSP430F5438 has 256 16-bit words. This means there is the total of 4096 bits that are either unperturbed, weakly perturbed, or strongly perturbed.

Figure 6.8 Characterization of Bits in a Segment as SPFB, WPFB, Logic 0 or Logic 1 shows the state of individual bits in a perturbed region of the Flash memory. Columns represent individual words and 16 rows in each image represent individual

bit positions. The top most strip shows the state of the Flash bits when the partial programming is done at 8 MHz clock frequency, the middle strip is at 4 MHz, and the lower one is at 1 MHz clock frequency. The yellow pixels represent fully programmed bits, while pixels represent erased bits, green pixels represent weakly perturbed bits, and red pixels represent SPBs. The visual inspection of the images allows us the following conclusions.

a. Running at higher clock frequency allows for a better control of the partial programming duration. This in turn results in an increased number of weakly and strongly perturbed Flash bits. This observation holds true regardless of chip sample.
b. Some chips may produce a relatively small number of perturbed Flash bits when the processor clock frequency is low (sample chips 2 and 3 at 1 MHz processor clock).
c. The majority of unperturbed Flash bits appear to be in the programmed state rather than in the erased state in sample chips 2 and 3. An exception to this rule may be sample chip 3 when higher clock frequency is used during partial programming.
d. Flash bits that are SPFBs are not concentrated to any particular area of the segment thus allowing almost uniform distribution over the segment.

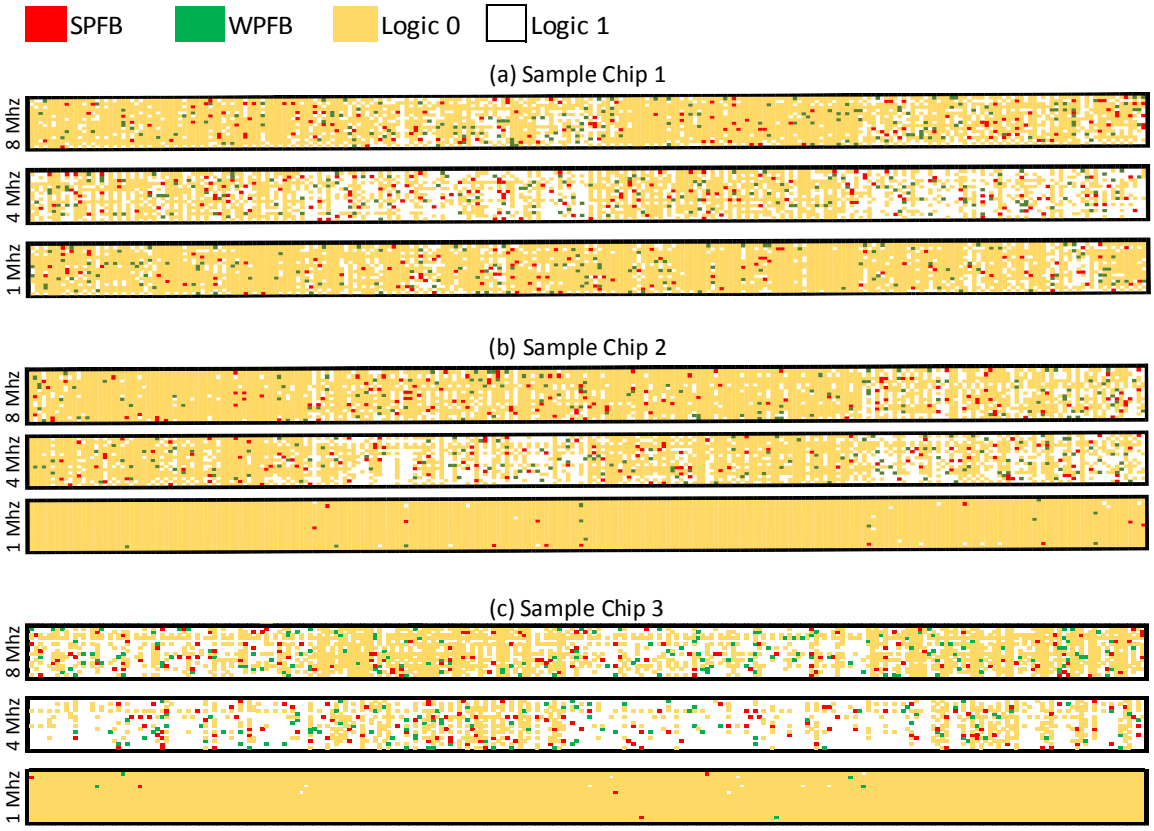

Figure 6.8 Characterization of Bits in a Segment as SPFB, WPFB, Logic 0 or Logic 1

6.2 Random Number Generation and NIST Tests Results

Random Numbers are generated from the values read from SPFB. The generated random number sequence are packed into binary form and written to a file, which are then later fed to the NIST Test suite [3]. We generate random number for different chips at different conditions, and save them in individual files. The different conditions are defined by the number of SPFB taken for generation of random numbers, which is indicated by *m* in random number generation algorithm discussed in Figure 4.10. In each of the case, we make sure that the number of random bits that is at least 10 million. We specify the number of bit-streams as 10 for each case for NIST Test with each bit-stream of length 1 million bits.

The result of NIST Test is given briefly as the *P-Value* and *Proportion*. *P-Value* is displayed as a decimal value while *Proportion* is indicated as a ratio. In order to be considered a random sequence, the *P-Value* should be greater than 0.0001. In case when 10 sequences are provided, the whole sequence is considered a random sequence if at least 8 sequences out of 10 sequences pass the test. This implies, in our case the sequence is considered random is *P-Value* is 0.0001 and *Proportion* is greater than or equal to 8/10.

In our case, *m* is varied from 3 to 10 to find the optimal number of SPFB that would satisfy the randomness requirement. The motivation behind doing this is to find the minimum number of bits that would generate random bits most efficiently. Our experiments show that random number generated by combining at least 5 SPFBs satisfy the NIST Test, but it comes at a cost of application of Von-Neumann De-biasing. Using 10 SPFBs would produce random numbers that pass the NIST Test without application of Von-Neumann De-biasing. This means the original sequence generated using *m=10* would pass the NIST Test. However, for the case when m=3 and 4 or below, the generated bit sequence would not pass the NIST Test for all the cases even after application of Von-Neumann De-biasing.

NIST Test result for these three cases, i.e. using *m*=3, 5 and 10 are presented in Table 6.13, Table 6.14 and Table 6.15 respectively. For the case when 3 SPFBs are used, each chip fails all the test mentioned for the original sequence. The de-biased sequences however pass all the tests, except for sample chip 2. So using 3 SPFBs might not be sufficient for the proposed algorithm. Using 5 SPFBs, however, result in passing all the tests after de-biasing. Thus, we can say that the proposed algorithm that uses 5 SPFBs with de-biasing passes produces quality random se-

quences. With 10 SPFBs, the proposed algoritham produces random sequences that pass all the NIST tests with original bit sequences (without de-biasing).

Table 6.13 NIST Statistical Test Result for case using 3 SPFBs (*m=3*)

NIST Test	Sample 1				Sample 2				Sample 3			
	Original		De-biased		Original		De-biased		Original		De-biased	
	P-value	Propo-rtion	P-value	Propo-rtion	P-value	Propo-rtion	P-value	Propo-rtion	P-value	Propo-rtion	P-value	Propo-rtion
Frequency	0	3/10	0.740	10/10	0	2/10	0.000	6/10	0	2/10	0.035	9/10
BlockFrequency	0	0/10	0.0351	10/10	0	3/10	0.740	10/10	0	0/10	0.534	10/10
CumulativeSums	0	3/10	0.740	8/10	0	2/10	0.018	6/10	0	1/10	0.350	9/10
Runs	0	0/10	0.018	10/10	0	2/10	0.213	10/10	0	0/10	0.035	8/10
LongestRun	0	0/10	0.534	10/10	0.911	9/10	0.350	9/10	0	3/10	0.534	9/10
Rank	0.534	10/10	0.350	10/10	0.122	10/10	0.035	10/10	0.911	10/10	0.740	10/10
FFT	0.067	7/10	0.122	10/10	0.350	8/10	0.534	10/10	0	6/10	0.122	9/10
NonOverlappingT.	0.534	10/10	0.911	10/10	0.213	10/10	0.740	10/10	0.740	9/10	0.911	10/10
OverlappingTemplate	0	0/10	0.350	10/10	0.035	7/10	0.740	10/10	0	2/10	0.740	10/10
Universal	0	0/10	0.122	9/10	0.066	7/10	0.350	10/10	0	0/10	0.740	10/10
ApproximateEntropy	0	0/10	0.350	10/10	0	5/10	0.740	10/10	0	0/10	0.350	9/10
RandomExcursions	--	2/3	--	7/7	--	--	--	6/6	--	3/3	--	4/4
RandomExcursionsV.	--	3/3	--	7/7	--	--	--	6/6	--	3/3	--	4/4
Serial	0	0/10	0.740	10/10	0.534	9/10	0.350	10/10	0.035	8/10	0.740	10/10
LinearComplexity	0.122	9/10	0.534	10/10	0.740	10/10	0.911	10/10	0.740	10/10	0.740	10/10

Table 6.14 NIST Statistical Test Result for case using 5 SPFBs (*m=5*)

NIST Test	Sample 1				Sample 2				Sample 3			
	Original		De-biased		Original		De-biased		Original		De-biased	
	P-value	Propo-rtion	P-value	Propo-rtion	P-value	Propo-rtion	P-value	Propo-rtion	P-value	Propo-rtion	P-value	Propo-rtion
Frequency	0	3/10	0.122	10/10	0.001	6/10	0.534	9/10	0.069	6/10	0.122	10/10
BlockFrequency	0	2/10	0.534	10/10	0.001	6/10	0.069	10/10	0.069	9/10	0.035	9/10
CumulativeSums	0	2/10	0.911	10/10	0.001	6/10	0.740	9/10	0.000	5/10	0.350	10/10
Runs	0	7/10	0.350	10/10	0.067	6/10	0.534	10/10	0.000	7/10	0.911	10/10
LongestRun	0.740	10/10	0.067	10/10	0.534	8/10	0.534	9/10	0.018	10/10	0.213	10/10
Rank	0.911	9/10	0.534	10/10	0.911	10/10	0.122	10/10	0.740	10/10	0.350	10/10
FFT	0.534	10/10	0.351	10/10	0.213	10/10	0.018	10/10	0.534	10/10	0.213	10/10
NonOverlappingT.	0.740	10/10	0.911	10/10	0.911	10/10	0.740	10/10	0.213	10/10	0.740	10/10
OverlappingTemplate	0.350	10/10	0.035	10/10	0.740	8/10	0.911	10/10	0.740	10/10	0.740	10/10
Universal	0.534	10/10	0.350	10/10	0.213	8/10	0.350	9/10	0.534	10/10	0.740	10/10
ApproximateEntropy	0.534	10/10	0.740	10/10	0.004	6/10	0.911	10/10	0.350	8/10	0.911	10/10
RandomExcursions	--	1/1	--	5/5	--	2/2	--	6/6	--	3/3	--	7/7
RandomExcursionsV.	--	1/1	--	5/5	--	2/2	--	6/6	--	3/3	--	7/7
Serial	0.350	10/10	0.740	10/10	0.122	10/10	0.740	10/10	0.350	10/10	0.740	10/10
LinearComplexity	0..911	10/10	0.122	10/10	0.740	10/10	0.213	10/10	0.213	10/10	0.911	10/10

Table 6.15 NIST Statistical Test Result for case using 10 SPFBs (*m=10*)

NIST Test	Sample 1				Sample 2				Sample 3			
	Original		De-biased		Original		De-biased		Original		De-biased	
	P-value	Propo-rtion	P-value	Propo-rtion	P-value	Propo-rtion	P-value	Propo-rtion	P-value	Propo-rtion	P-value	Propo-rtion
Frequency	0.740	10/10	0.534	10/10	0.534	9/10	0.534	10/10	0.534	10/10	0.122	10/10
BlockFrequency	0.740	10/10	0.534	10/10	0.350	10/10	0.911	10/10	0.740	10/10	0.122	10/10
CumulativeSums	0.740	10/10	0.740	10/10	0.534	9/10	0.018	10/10	0.122	10/10	0.350	10/10
Runs	0.213	10/10	0.740	10/10	0.740	9/10	0.018	10/10	0.911	9/10	0.534	10/10
LongestRun	0.213	10/10	0.534	10/10	0.534	10/10	0.911	9/10	0.911	10/10	0.122	10/10
Rank	0.740	10/10	0.534	10/10	0.122	10/10	0.350	10/10	0.350	10/10	0.122	10/10
FFT	0.740	10/10	0.740	10/10	0.534	10/10	0.534	10/10	0.350	10/10	0.350	10/10
NonOverlappingT.	0.911	10/10	0.911	10/10	0.534	10/10	0.740	10/10	0.911	10/10	0.911	10/10
OverlappingTemplate	0.534	10/10	0.213	10/10	0.534	10/10	0.534	10/10	0.740	10/10	0.035	10/10
Universal	0.740	10/10	0.911	9/10	0.213	10/10	0.350	10/10	0.350	10/10	0.122	10/10
ApproximateEntropy	0.534	10/10	0.350	10/10	0.534	10/10	0.911	10/10	0.740	10/10	0.350	10/10
RandomExcursions	--	8/8	--	3/3	--	3/3	--	6/6	--	5/5	--	6/6
RandomExcursionsV.	--	8/8	--	3/3	--	3/3	--	6/6	--	5/5	--	6/6
Serial	0.534	10/10	0.911	10/10	0.534	9/10	0.740	10/10	0.122	10/10	0.740	10/10
LinearComplexity	0..534	9/10	0.911	10/10	0.911	10/10	0.740	10/10	0.350	10/10	0.350	10/10

Although already verified by the Frequency test and Block Frequency Test, 2D binary image representation of a random bit sequence generated is visually illustrated in Figure 6.9. The uniform distribution of black and white pixels can be observed in both the sequences presented. Here, the black pixels represent a bit '0' while the white pixels represent a bit '1' in a random bit sequence of length 65536 that is presented as a 256x256 matrix. Of the many random sequences generated, Figure 6.9 presents random bit sequence generated using 10 SPFBs.

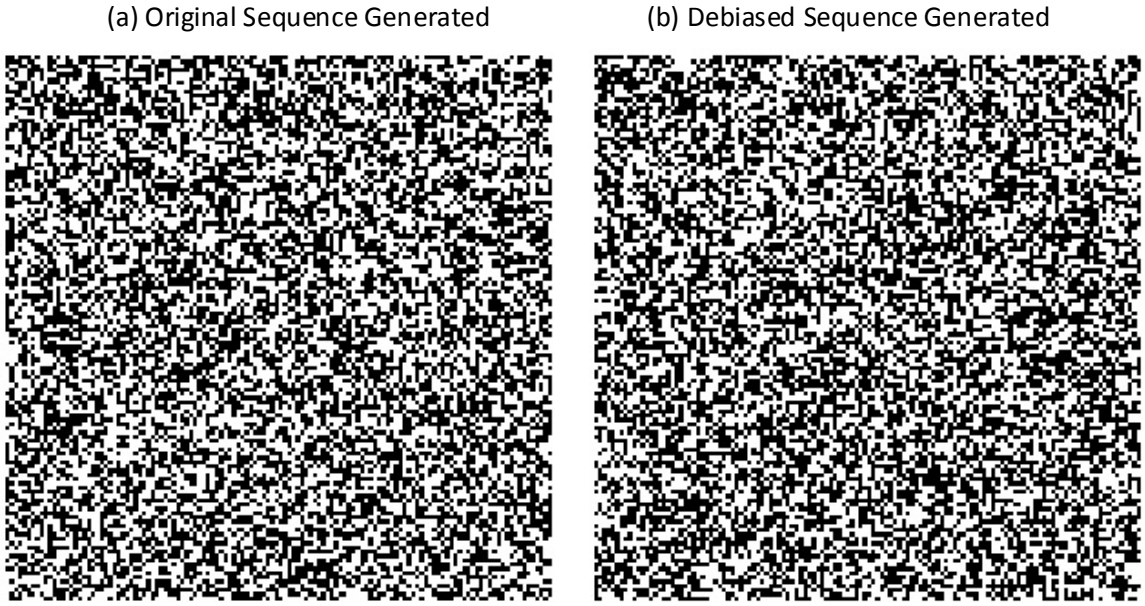

Figure 6.9 Visual Representation of a TRN generated using 10 SPFBs

(a) Original Sequence (b) De-biased Sequence

6.3 Performance

One metric that qualifies our technique as best suited approach for low end embedded systems is its throughput. Throughput refers to the number of bits produced per unit time. In our case, for a random number generator, throughput can be

defined as the number of good random bits generated per unit time. Since the algorithm for generation of random number is a function of *m*, the number of SPFBs that are combined to produce random number and the number of reads performed at a time *nStep*, our throughput is also the function of these parameters (as mentioned in algorithm presented in Figure 4.10). The result presented in Section 6.2 indicate that using *m*=10 would produce quality random bits. So the optimal case for generating *nStep*=1024 random bits using *m*=10 would require 123 processor clock cycles in MSP430F5438. This would correspond to 8,525 random bits when the processor is run at F_{CPU}=1, 048,576 Hz (~1 Mhz). This would scale to 68,200 random bits per second for the case when the processor is run at 8,388,608 Hz (~8 Mhz). This throughput by far exceeds typical requirement for random numbers on this type of low-end embedded systems.

Size requirement is another metric that determines the quality of any software implementation. The memory footprint occupied by our code is very small, thus making it best suited for resource constrained systems that will empower IoT lowest-tier infrastructure. The code for perturbing Flash memory segment occupies 93 bytes of RAM memory at worst case. Worst Case is the case with maximum number of NOPs required for perturbing. The code for identifying the SPFB would occupy 238 Bytes of memory while the generation of random number would occupy 232 Bytes of memory.

The amount of memory in RAM for data while calculating the random bits is a function of number of bits required. The algorithm can be tuned by changing the value of *nStep*. At any point in generation of *nStep*-bit random number using *m* SPFBs, the memory required will be (*nStep * 2)* bits of RAM memory. In case when

enough memory is not available in RAM, as can be case in some low-end resource constrained system, the value of *nStep* can be configured to a smaller value to produce smaller random number sequence. Many such smaller random number sequence can be combined to form a larger random number sequence as post processing.

CHAPTER 7

CONCLUSIONS

This thesis presented a new technique to generate true random numbers by utilizing read noise from perturbed NOR Flash memory cells. The proposed method is demonstrated on a TI MSP430 family of low-power and low-end microcontrollers. The proposed technique is applicable in a wide range of microcontrollers and offers the following advantages relative to the state-of-the-art approaches: (a) it does not require any additional hardware modifications and/or interfaces; (b) it is robust, cost-effective, and high-throughput; (c) it is implemented in software; and (d) it is flexible and can be tailored to work in resource-constrained devices.

The proposed technique relies on perturbing NOR Flash memory cells using partial programming. In the perturbed state, Flash memory cell's threshold voltage is close to the read sensing voltage and thus the outcome of read operations is uncertain – cells read as either logic '1' or logic '0' depending on read noise. We characterize NOR Flash memory regarding its perturbed state and show how timing control of partial programming can impact the number of perturbed states. The thesis describes algorithms for partial programming, characterization of Flash memory cells, and generating true random numbers using strongly perturbed Flash memory cells.

The experimental evaluation shows that the proposed technique results in a high-quality random sequences that pass all the tests from the NIST statistical suite.

In future work, several new directions can be considered. First, this thesis utilizes partial programming of Flash memory as a way to introduce a perturbed state. An alternative approach is to consider partial erasing – though this operations takes more time, it may have potential to produce more strongly perturbed states. Next, generating true random numbers is closely related with creating Physical Un-clonable Functions (PUFs) that can be used to uniquely identify devices. The goal of this direction is to find an algorithm for generating PUFs from perturbed states of Flash memory. Another venue for future exploration is to broaden the number and type of microcontrollers this approach has been tested on.

REFERENCES

[1] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Cryptanalytic Attacks on Pseudorandom Number Generators," in *Fast Software Encryption*, vol. 1372, S. Vaudenay, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 168–188, 1998.

[2] B. Koerner, "Russians Engineer A Brilliant Slot Machine Cheat—And Casinos Have No Fix," 06-Feb-2017. [Online]. Available: https://www.wired.com/2017/02/russians-engineer-brilliant-slot-machine-cheat-casinos-no-fix/.

[3] L. E. Bassham *et al.*, "A statistical test suite for random and pseudorandom number generators for cryptographic applications," National Institute of Standards and Technology, Gaithersburg, MD, NIST SP 800-22r1a, 2010.

[4] E. B. Barker and J. M. Kelsey, "Recommendation for Random Number Generation Using Deterministic Random Bit Generators," National Institute of Standards and Technology, NIST SP 800-90Ar1, Jun. 2015.

[5] M. S. Turan, E. Barker, J. Kelsey, K. A. McKay, M. L. Baish, and M. Boyle, "Recommendation for the entropy sources used for random bit generation," National Institute of Standards and Technology, Gaithersburg, MD, NIST SP 800-90b, Jan. 2018.

[6] A. Maiti, R. Nagesh, A. Reddy, and P. Schaumont, "Physical unclonable function and true random number generator: a compact and scalable implementation," in *Proceedings of the 19th ACM Great Lakes symposium on VLSI*, pp. 425–428, 2009.

[7] L. Westlund, "Random Number Generation Using the MSP430." Texas Instruments, Oct-2006. [Online]. Available: http://www.ti.com/lit/an/slaa338/slaa338.pdf.

[8] M. Majzoobi, F. Koushanfar, and S. Devadas, "FPGA-Based True Random Number Generation Using Circuit Metastability with Adaptive Feedback Control," in *Cryptographic Hardware and Embedded Systems – CHES 2011*, vol.

6917, B. Preneel and T. Takagi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 17–32, 2011.

[9] H. Hata and S. Ichikawa, "FPGA Implementation of Metastability-Based True Random Number Generator," in *IEICE Transactions on Information and Systems*, vol. E95.D, no. 2, pp. 426–436, 2012.

[10] C. Tokunaga, D. Blaauw, and T. Mudge, "True Random Number Generator with a Metastability-Based Quality Control," in *IEEE International Solid-State Circuits Conference, 2007. Digest of Technical Papers*, pp. 404–611, 2007.

[11] J. Wu and M. O'Neill, "Ultra-lightweight true random number generators," *Electronics Letters*, vol. 46, no. 14, p. 988, 2010.

[12] P. Z. Wieczorek and K. Golofit, "Dual-Metastability Time-Competitive True Random Number Generator," in *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 61, no. 1, pp. 134–145, Jan. 2014.

[13] Y. Wang, W. Yu, S. Wu, G. Malysa, G. E. Suh, and E. C. Kan, "Flash Memory for Ubiquitous Hardware Security Functions: True Random Number Generation and Device Fingerprints," presented at the *IEEE Symposium on Security and Privacy*, San Francisco, CA, pp. 33–47, 2012.

[14] M. J. Kirton and M. J. Uren, "Noise in solid-state microstructures: A new perspective on individual defects, interface states and low-frequency (1/ f) noise," *Advances in Physics*, vol. 38, no. 4, pp. 367–468, Jan. 1989.

[15] Yinglei Wang, Wing-kei Yu, S. Q. Xu, E. Kan, and G. E. Suh, "Hiding Information in Flash Memory," in *IEEE Symposium on Security and Privacy (SP)* , pp. 271–285, 2013.

[16] B. Ray and A. Milenkovic, "True Random Number Generation Using Read Noise of Flash Memory Cells," in *IEEE Transactions on Electron Devices*, vol. 65, no. 3, pp. 963–969, Mar. 2018.

[17] D. E. Holcomb, W. P. Burleson, and K. Fu, "Power-Up SRAM State as an Identifying Fingerprint and Source of True Random Numbers," in *IEEE Transactions on Computers*, vol. 58, no. 9, pp. 1198–1210, Sep. 2009.

[18] F. Tehranipoor, W. Yan, and J. A. Chandy, "Robust hardware true random number generators using DRAM remanence effects," in *IEEE International*

Symposium on Hardware Oriented Security and Trust (HOST), pp. 79–84, 2016.

[19] F. Tehranipoor, N. Karimian, W. Yan, and J. A. Chandy, "DRAM-Based Intrinsic Physically Unclonable Functions for System-Level Security and Authentication," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 3, pp. 1085–1097, Mar. 2017.

[20] C. Pyo, S. Pae, and G. Lee, "DRAM as source of randomness," *Electronics Letters*, vol. 45, no. 1, p. 26, 2009.

[21] L. T. Clark, J. Adams, and K. E. Holbert, "Reliable techniques for integrated circuit identification and true random number generation using 1.5-transistor flash memory," *Integration, the VLSI Journal*, Nov. 2017.

[22] S. Balatti *et al.*, "Physical Unbiased Generation of Random Numbers With Coupled Resistive Switching Devices," *IEEE Transactions on Electron Devices*, vol. 63, no. 5, pp. 2029–2035, May 2016.

[23] H. Jiang *et al.*, "A novel true random number generator based on a stochastic diffusive memristor," *Nature Communications*, vol. 8, no. 1, Dec. 2017.

[24] T. Greg and G. Cox, "Behind Intel's New Random Number Generator." IEEE Spectrum, 24-Aug-2011.

[25] B. Jun and P. Kocher, "The Intel random number generator." Intel Corporation, 22-Apr-1999. [Online]. Available: http://www.cryptography.com/resources/whitepapers/IntelRNG.pdf.

[26] John. M., "Intel® Digital Random Number Generator (DRNG) Software Implementation Guide." Intel, 15-May-2014. [Online]. Available: https://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide.

[27] "AMD Random Number Generator." Advanced Micro Devices, 6/27 /17. [Online]. Available: https://support.amd.com/TechDocs/amd-random-number-generator.pdf.

[28] Z. Tang, X. Zhang, Y. Zhang, and L. Qi, "Portable true random number generator for personal encryption application based on smartphone camera," *Electronics Letters*, vol. 50, no. 24, pp. 1841–1843, Nov. 2014.

[29] H. Guo, W. Tang, Y. Liu, and W. Wei, "Truly random number generation based on measurement of phase noise of a laser," *Physical Review E*, vol. 81, no. 5, May 2010.

[30] J. Zhang *et al.*, "A robust random number generator based on differential comparison of chaotic laser signals," *Optics Express*, vol. 20, no. 7, p. 7496, Mar. 2012.

[31] Y. Terashima, K. Ugajin, A. Uchida, T. Harayama, and K. Yoshimura, "Fast physical random bit generation using a photonic integrated circuit," in *International Symposium on Nonlinear Theory and Its Applications*, Yugawara, Japan, 2016.

[32] "<cstdlib> (stdlib.h)." [Online]. Available: http://www.cplusplus.com/reference/cstdlib/.

[33] "<random>." [Online]. Available: http://www.cplusplus.com/reference/random/.

[34] "Class Random." [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/util/Random.html.

[35] "random — Generate pseudo-random numbers." [Online]. Available: https://docs.python.org/3/library/random.html.

[36] Y. Wang, "Flash Memory For Ubiquitous Hardware Security Functions," Cornell University, Ithaca NY, 2014. [Online]. Available: https://ecommons.cornell.edu/handle/1813/36037.

[37] R. Micheloni, A. Marelli, and S. Commodaro, "NAND overview: from memory to systems," in *Inside NAND Flash Memories*, Dordrecht: Springer Netherlands, pp. 19–53, 2010.

[38] L. Crippa, R. Micheloni, I. Motta, and M. Sangalli, "Nonvolatile Memories: NOR vs. NAND Architectures," in *Memories in Wireless Systems*, R. Micheloni, G. Campardo, and P. Olivo, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 29–53, 2008.

[39] A. R. Duncan, M. J. Gadlage, A. H. Roach, and M. J. Kay, "Characterizing Radiation and Stress-Induced Degradation in an Embedded Split-Gate NOR Flash Memory," *IEEE Transactions on Nuclear Science*, vol. 63, no. 2, pp. 1276–1283, Apr. 2016.

[40] J. von Neumann, “Various Techniques Used in Connection with Random Digits.” [Online]. Available: https://mcnp.lanl.gov/pdf_files/nbs_vonneumann.pdf.

[41] “NIST SP 800-22: Documentation and Software - Random Bit Generation | CSRC.” [Online]. Available: https://csrc.nist.gov/projects/random-bit-generation/documentation-and-software.