



DEPARTMENT OF ELECTRICAL
ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY
MADRAS
CHENNAI - 600036

A Provably Secure True Random Number Generator

A Project Report

Submitted by

CHINDAM BALRAJ

In the partial fulfilment of requirements

For the award of the degree

Of

MASTER OF TECHNOLOGY

June 2021

CERTIFICATE

This is to undertake that the Thesis (or Project report) titled **A PROVABLY SECURE TRUE RANDOM NUMBER GENERATOR**, submitted by me to the Indian Institute of Technology Madras, for the award of M.Tech, is a bonafide record of the research work done by me under the supervision of **Prof Veezhinathan Kamakoti** and **Dr. Anbarasu Manivannan**. The contents of this Thesis (or Project report), in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Place: Chennai 600 036

Date: 18 June 2021

CHINDAM BALRAJ

EE19M044

Prof. Veezhinathan Kamakoti

Project Guide

Dr. Anbarasu Manivannan

Project Co-Guide

ACKNOWLEDGEMENTS

Foremost, I would like to thank my guide **prof. Kamakoti** who gave me an opportunity to be a part of Shakti Project.

I am grateful to Project Associates **Arjun menon** and Sadhana for their encouragement, advice, and help whenever needed. I appreciate their constant involvement in my project from scratch.

I would like to take this opportunity to acknowledge my school friends, B.tech friends and madras batch-mates for helping me out in this pandemic situation. I would like to thank my family for their love and encouragement, without their support this M.Tech program would not have been possible.

Chindam Balraj

ABSTRACT

Good random numbers are required for good cryptography. This study assesses the True Random Number Generator (TRNG), which is based on hardware, for use in cryptographic applications. Each and every cryptographic protocol in need of random numbers, which are unknown to predictor. TRNG generates the random numbers which are used as padding bytes, nonces, challenges. The most important requirement is that attackers, even those who are familiar with the RNG design, must be unable to forecast the TRNG outputs in any practical way.

In this project TRNG is splitted into Entropy source and Deterministic Random number Generator (PRNG). Entropy source is based on sampling jitter in combination of Ring oscillators and PRNG uses Advanced Encryption Algorithm for encrypting. In this paper Random numbers are generated with high speed, which are unable to forecast by the attackers.

TABLE OF CONTENTS

	Page
ACKNOWLEDGEMENTS	i
ABSTRACT	ii
LIST OF TABLES	v
LIST OF FIGURES	vi
ABBREVIATIONS	vii
CHAPTER 1: INTRODUCTION	1
1.1 Motivation	2
CHAPTER 2: Background	3
2.1 Entropy	3
2.2 Deterministic Random Bit Generators	4
2.3 The need for non-deterministic random bit generators	5
CHAPTER 3: Advanced Encryption Algorithm	6
3.1 Cipher	8
3.1.1 SubBytes()	8
3.1.2 ShiftRows()	9
3.1.3 Mix Columns()	9
3.1.4 AddRoundKey()	10
3.1.5 Key Expansion for AES-128	11
CHAPTER 4: Architecture	14
CHAPTER 5: Blocks of TRNG	16
5.1 Ro based Entropy source	16
5.1.1 Combining Ro outputs to exploit randomness of phase jitter	17
5.1.2 Urn model to detect no of Ros	18

5.2	Von Neumann Corrector	21
5.3	Health checks	21
5.3.1	Repetition Count Test	22
5.3.2	Adaptive Proportion Test	22
5.4	Conditioning	24
5.5	Counter based DRBG based on Block Ciphers	26
5.5.1	Instantiate	27
5.5.2	Generation	28
5.5.3	Reseeding	29
CHAPTER 6:	Implementation of TRNG	30
6.1	Implementation using two AES hardware modules	30
6.1.1	Synthesis Report	31
6.2	Implementation using only one AES hardware module	31
6.2.1	Synthesis Report	32
6.3	Results	33
6.3.1	BSV Simulation results	33
REFERENCES	36

LIST OF TABLES

Table	Title	Page
5.1	Number of urns for certain Entropy	19
5.2	Von Neumann Corrector	21
5.3	Health bounds for 256-bits	23

LIST OF FIGURES

Figure	Title	Page
3.1	The state array input and output	7
3.2	Block size key combinations	7
3.3	SubBytes()	8
3.4	S-box to each indice of the State.	9
3.5	Shift rows()	9
3.6	Mix columns()	10
3.7	Add Round Key	10
3.8	Hardware utilization of AES	13
4.1	Block diagram of RNG	14
5.1	RO based Entropy source	16
5.2	Periodic square wave with jitter marked as lines	17
5.3	A single jitter event	19
5.4	Hardware for entropy source	20
5.5	Updating CE	25
5.6	Mechanism of DRBG	26
5.7	Instantiate of DRBG	27
5.8	Generation of DRBG	28
6.1	Number of cycles for 1024*128 random numbers	31
6.2	Hardware utilisation of TRNG(2 AES modules)	31
6.3	Number of cycles for 1024*128 random bits	32
6.4	Hardware utilisation of TRNG(1 AES module)	32
6.5	Simulation results of TRNG	33
6.6	Simulation results of TRNG	34
6.7	Simulation results of TRNG	35

ABBREVIATIONS

TRNG	True Random Number Generator
AES	Advanced Encryption Algorithm
OSTE	Online Self Test Entropy
CE	Conditioning Entropy
DRBG	Deterministic Random Bit Generator
PRNG	Pseudo Random Number Generator

CHAPTER 1

INTRODUCTION

Cryptographic systems rely on secret bits and keys for their security. These bits must be random to avoid guessing, hence this secret bits are generated by TRNGs. The production of random numbers is monitored by several standards and technological requirements as a crucial security function. TRNGs (or RNGs) are digital true random number generators that may be implemented using only digital components in semiconductor technology. They are particularly cost effective and flexible. TRNGs which built by digital components are very easy to construct and takes less power while TRNG with analog components is critical to analyze and consumes more power.

Pseudorandom number generators (PRNGs), which use a deterministic process to extend a brief random string into a set of random looking numbers, are adequate for many purposes. For cryptographic applications, however, it is critical to create pseudo random bits that are unpredictably recognised even by the most powerful attacker. Furthermore, pseudo Random Number Generator with quality output which is tolerance to attacks can be built, which is called TRNG. We need to seed this PRNG with high quality seed which have good entropy (greater than 0.5 bits of entropy for one bit).

For asymmetric methods like RSA and DSA, random number generators are necessary to construct public/private keypairs. Randomly generated keys for symmetric and hybrid cryptosystems are also generated. Many cryptographic systems employ as much key material as ciphertext and require a fully random procedure to produce the keystream.

In this project, we focus on practical design of TRNG, which have possibility of implementation on Asic and fpga. We develop a Trng which splits into three components Entropy source, health checks and post processing. The quality of Trng depend on this

three components. initially the seed is generated from entropy source then it is continuously checked by health tests and if the given seed is healthy then this seed value is used to seed the Prng, which continuously produce high quality random outputs with certain frequency. This outputs Produced by proposed Trng is unpredictable even if knew the previous outputs.

1.1 Motivation

The primary factor considered for carrying out this work is the problem of generation of secure random numbers. Related work in this domain consuming much resources and does not have any stable entropy source, which is producing seed with better entropy. Day by Day electronic gadgets are shrinking in terms of area so my aim is to design a Trng with as less resources as possible and with quality (good entropy) random numbers.

CHAPTER 2

Background

To understand True Random Number Generator (TRNG) we need to know about the terms Entropy, Health checks, Conditioning and DRBG. Different TRNG designs use different types of Entropy sources and different DRBG mechanism to produce random numbers.

2.1 Entropy

The entropy of a process is a measure of how random it is. While there are various approaches to quantify entropy, Shannon entropy is easy to quantify, so we use Shannon entropy here [Shannon (1948)].

$$H_j = \sum_{j=1}^m p_j \log_2 p_j$$
$$H_\infty = \min_{j=1}^m (-\log_2 p_j)$$

The probability of the process being in the j_{th} of n possible states or returning the j_{th} of n potential outputs is given by p_j in the above formulations. We measure entropy in bits by applying a base-2 logarithm. Shannon entropy is a metric that estimates the average amount of data needed to describe a condition. Min-entropy on the other hand, quantifies the likelihood that an attacker can guess the state with a single guess.

p_j is the probability that an output equals j , where $0 < j < 2^k$, where $0 < k < 2$. Let the random number generator produce q binary bits. As a result, $p_j = 2^{-q}$ for a perfect random number generator. The output's Shannon entropy and min-entropy are both equal to q bits in this situation, implying that all conceivable outcomes are likely equally. On average, output information cannot be represented in less than k bits and with a probability larger than 2^{-k} , an output cannot be guessed by the attacker.

To computationally bounded adversaries, a RNG for cryptographic applications should appear as near to a flawless RNG as possible.

2.2 Deterministic Random Bit Generators

Pseudo Random number Generators (PRNGs) use a known algorithm to generate a huge number, which are completely random from a small quantity of unpredictability numbers called seed. PRNGs are split into two parts: cryptographic and non-cryptographic. The linear congruential generators present in many programming libraries are non-cryptographic PRNGs. These libraries produce statistically acceptable random numbers, but they can't be used for cryptographic keying applications. These PRNGs take input as seed and give output. The seed used for PRNG is mathematically derived so we can derive the internal state and output of PRNG easily. An attacker can easily predict the outputs if he knew previous outputs so there are security problems associated with PRNG.

Cryptographic PRNGs are also known as DRBGs. AES and SHA are good examples of a strong cryptographic algorithm, which produces random numbers from a random seed. DRBGs produce large quantities of outputs from an initial seed state using deterministic processes. The entropy of the seed is always greater than the seed entropy since the output is an entirely deterministic process. If a DRBG is seeded well, we can't differentiate between DRBG and ideal RNG.

The attacker who could successfully guess the seed with 100 percent can easily predict the complete output of RNG. The seed of DRBG is 256 bits, which are very tough to predict. The probability of detecting a 256-bit seed is $\frac{1}{2^{256}}$. Cryptographic applications frequently need exceptionally high output quality, demanding meticulous development and testing and algorithms. The specifications required for DRBG mentioned in this NIST paper [E. Barker and J. Kelsey (2012)].

2.3 The need for non-deterministic random bit generators

A nondeterministic irregular bit generator employs a nondeterministic source to produce randomness. Most of the randomness (random seed) produced by unpredictable natural processes such as atmospheric noise, jitter noise and nuclear decay. The major performance of TRNG depends upon the quality of seed produced by entropy source.

We can create randomness from within a Deterministic system. DRBG itself is not secure if it is not randomly seeded. Seeding of TRNG requires complete randomness.

Computers which don't have hardware entropy source try to obtain seed from embedded devices or hardware like keyboard and hard drives. The entropy derived from these sources like keyboard is not at all sufficient, so it is better to have hardware entropy source. The health checks and DRBGs can be software since they are Deterministic algorithms.

Computational bounded test that can assess a RNG's output and authoritatively affirm that the output is random, since no test is perfect to detect randomness so defects are frequently overlooked. Entropy source should produce output regularly with some frequency. Some software system stores a seed value in hardware which have security problems, since there are some problems with entropy source. Hardware that provides a well-designed, efficient, and easy-to-use hardware entropy source is the greatest answer to these problems.

CHAPTER 3

Advanced Encryption Algorithm

The Advanced Encryption Standard (AES) states a cryptographic algorithm, which is utilised to safeguard electronic data. AES algorithm [FIPs (2001)] is used for both decrypting and encrypting by using symmetric cipher. The process of encryption changes the data into unrecognisable form called cipher text and decryption converts the data from unrecognisable form to normal data called plain text. The AES utilises 3 key lengths, which are specifically 128, 256, 192 bits to encrypt and decrypt. The method can be used with any of the three key lengths listed. As it uses 3 lengths we call them as AES-128, AES-192, AES-256.

AES algorithm uses 128-bit sequences (digits with values of 0 or 1) for both input and output. These sequences are frequently referred to as blocks, and their length refers to the amount of bits they contain. The AES algorithm's Cipher Key is of 3 bit lengths, which are 128, 192, or 256 bits. The AES algorithm has restriction of those key lengths. There are standards for AES algorithm to use specified key lengths. In this paper we used AES-128 Algorithm.

AES operations are carried out on a 2D array of bytes known as the State. The state array consists of 4 rows, where each row consists of Nb number of bytes (8 bits for bytes). The Nb value for AES 128 is 4. The byte in the state is denoted by specific row and column by using alphabet s, which consists of two indices, with a row r, which is ranging from $0 < \text{row} < 4$ and a column number c, which is ranging from $0 < \text{cmn} < \text{Nb}(4)$. We can denote state as $s_{r,c}$. The state array input, output is shown in figure 3.1.

As a result, the input array, in, is transferred to the State array at the start of the Cipher, as follows:

$$s_{\text{row}, \text{cmn}} = \text{input}[\text{row} + 4\text{cmn}] \quad \text{for } 0 < \text{row}, \text{cmn} < 4$$

similarly after calculating the AES, the state array is transferred to output array as:

$$output[row + 4cmn] = s_{row,cmn} \quad for \ 0 < row, cmn < 4$$

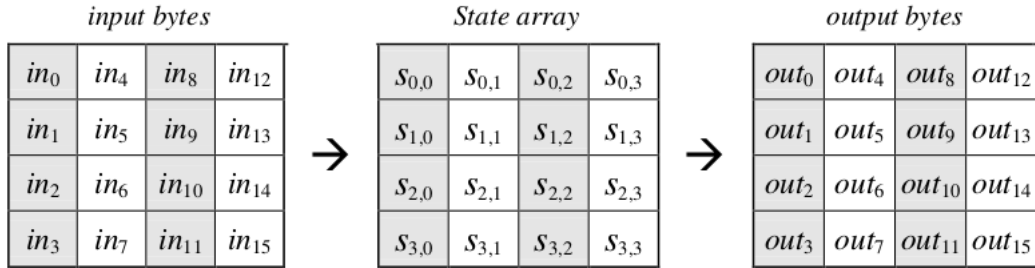


Fig. 3.1: The state array input and output

Then number of rounds that are performed during the AES algorithm is completely depends on the length of the key..when $Nk=4$ we have to perform $Nr=10$ rounds, similarly for $Nk=6,8$ we have to perform $Nr=12,14$ rounds respectively. the number of rounds is denoted by Nr . Number of rounds are explained in below figure.

	Key Length (Nk words)	Block Size (Nb words)	Number of Rounds (Nr)
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

Fig. 3.2: Block size key combinations

The AES algorithm performs a round function comprised of four distinct byte oriented transforms:

- substitution table is used which is called S-box..
- The shifting of rows performed by different offsets according to procedure.
- The xoring operations performed with in the same state.
- The final step is adding round key.

3.1 Cipher

In this section we discuss about encrypting(Cipher) the given data. The input is transferred to the State array by following the rules at the starting of the Cipher. The State array undergo some operations and it is modified by performing a round function 14, 12, or 10 times(it depends upon key length of AES) after an initial addition Round Key .After Nr number of rounds the state array is copied again into output array.

The cipher is performed step wise. The steps include SubBytes(),mixColumns(),shiftRows(),AddroundKey steps are used for key generation and for encrypting. This steps explained below detailly in section wise.

3.1.1 SubBytes()

This a non linear subByte transformation that uses a substitution table to act separately on each and every byte of the S-box. The subByte transformation is caluclated using the below equation. That SubByte is explained in fig 3.3.

$$b'_j = b_j \oplus b_{(j+4)mod8} \oplus b_{(j+5)mod8} \oplus b_{(j+6)mod8} \oplus b_{(j+7)mod8} \oplus c_j$$

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Fig. 3.3: SubBytes()

The state array after performing subbytes is shown in below figure 3.3.

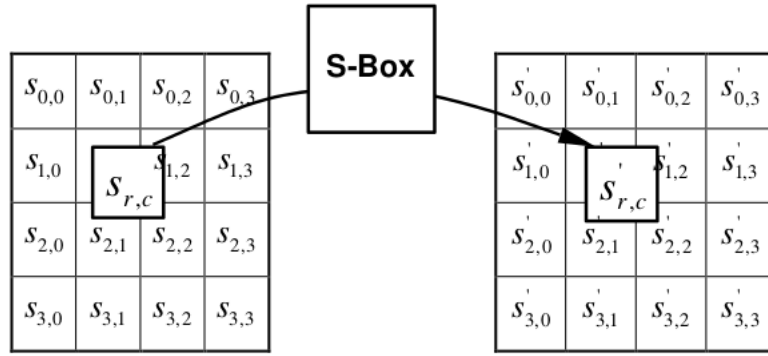


Fig. 3.4: S-box to each indice of the State.

3.1.2 ShiftRows()

Bytes in the State's rows are moved roundly across varying quantities of bytes. The shift rows are performed according to below equation and shown in figure 3.5.

$$S'_{row,cmn} = S_{row,(cmn+shift(row))mod4} \quad for \ 0 < cmn, row < 4$$

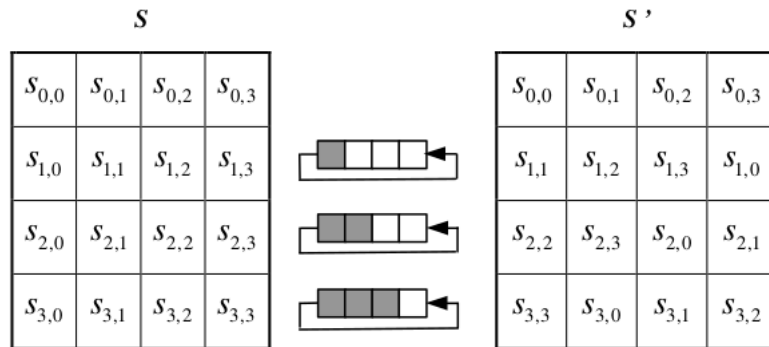


Fig. 3.5: Shift rows()

3.1.3 Mix Columns()

Every column in the State array is treated as a four-term polynomial by the Mix-Columns() transformation, which works column by column. How mixcolumns preformed is explained in below figure 3.6.

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}$$

Fig. 3.6: Mix columns()

After performing dot product between state array and numbers, we have to perform xor like this equation.

$$s'_{0,cmn} = (03.s_{1,cmn}) \oplus (02.s_{0,cmn}) \oplus s_{2,cmn} \oplus s_{3,cmn}$$

3.1.4 AddRoundKey()

AddRoundKey use a xor operations bitwise which are very simple to perform. Nb words (128 bits (4*32) since one word consists of 4 bytes (32 bits)) from the key make up each Round Key. Each of the Nb words is added to the State's columns, resulting in

$$[s'_{3,cmn}, s'_{2,cmn}, s'_{1,cmn}, s'_{0,cmn}] = [s_{3,cmn}, s_{2,cmn}, s_{1,cmn}, s_{0,cmn}] \oplus [K_{round*Nb+cmn}]$$

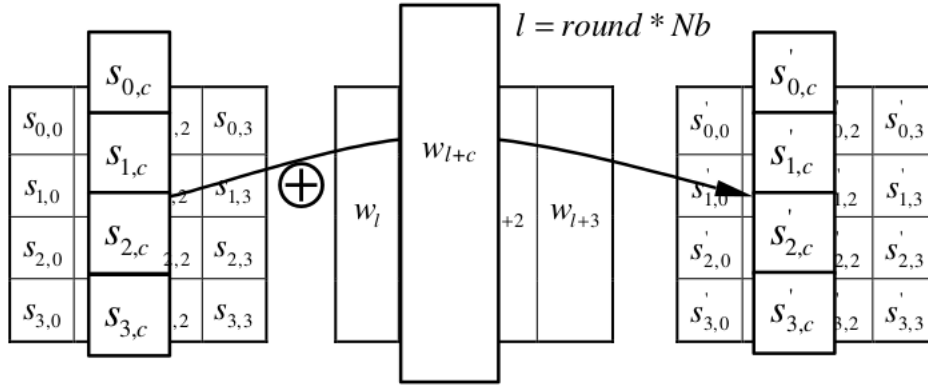


Fig. 3.7: Add Round Key

K_i is the value of key scheduled which is explained later in this section and round is value which is in between $0 \leq \text{round} \leq Nr$. the process of cipher is when $Nr=0$ the initial round value takes place else the process of AddRoundKey starts when Nr in

between 1 and Nr. The process explained in below figure 3.7.

3.1.5 Key Expansion for AES-128

The AES algorithm generates a key schedule using the Cipher Key of 128 bits using a Key Expansion procedure. The Nr and Nb value for key expansion is 10 and 4 respectively. Aes-128 bit totally generates $4 \times (10+1)$ 4-Byte words. One word consists of 32 bits (4 bytes). The 4 words are used for performing one round. Totally we require 41 four byte words, which is denoted by word[j], where j ranges from 0 to 41.

AES key generation code uses terms Sword, Rword, Rowcon. In this paragraph we discuss detailly about this terms. Sword is similar to Subbyte, where as in subbyte we apply s-box for entire state here we apply s-box to one word (4 bytes).

Rword is used in key generation. Rword is a simple shifting of bytes in a words. let assume word = [b0, b1, b2, b3], if we apply Rword to this, the resultant word = [b1, b2, b3, b0]. Rowcon[i] is $= 2^{i-1}, 0, 0, 0$. Rowcon[i] is performed in key expansion only when $i \bmod Nk = 0$.

In key expansion, for AES-128 we fill the word[j], for $0 \leq j < 5$ we fill direct key into word, after filling first 4 words, then we perform xor operations to get word[i], for $4 \leq j < 45$ except multiples of 4. For multiples of 4 we perform additional operations, before doing simple xor. The additional operations are performing Rword to previous word followed by Sword, then we perform xor operation with Rowcon[j/Nk]. This process is repeated for 40 times. The algorithm is explained in below pseudo code.

Algorithm 3.1: pseudo code for key expansion

Key expansion(Word word[44], Byte K[16],)

```

1 begin
2   Word tmp;
3   j=0;
4   while(j<44)
5     begin
6       word[j]=Word[K[4*j],K[4*j+1],K[4*j+2],K[4*j+3]];

```

```

7   j=j++;
8   end
9   j=4;
10  while(j<44)
11    begin
12      tmp=word[j-1];
13      if(j mod 4 ==0)
14        tmp = Sword(Rword(tmp))  $\oplus$  Rowcon[j/4];
15      word[j] = word[j - 4]  $\oplus$  tmp;
16      j=j++;
17    end
18  end

```

The next step after key generation is encrypting the data. Now we have expanded key of Word length 40. We use this key in encrypting the data, which is performed for 10 rounds for AES-128 bit algorithm. In this pseudo code first we copy the input into state array as explained above. The initial Add round key performed when Nr=0. After performing initial add round key, ten rounds of SubBytes, ShiftRows, MixColumns and AddRound key is performed, followed by one additional round of SubBytes, ShiftRows and Additional round Key. This total process completes the encryption of data. The pseudo code for encrypting is explained below in Algorithm 3.2

Algorithm 3.2: Pseudo code for Data encryption

```

Cipher(Word word[44], Byte input[16], output[16])
1  begin
2    Byte st[4,4];
3    st=input;
4    ARK(st,word[0,3])
5    r=1
6    while(r<10)

```

```

7  begin
8    SB(st); 9    SR(st); 10   Mix(st);
11   ARK(st,Word[r*4,(r+1)*(Nb)-1]);
12 end while
13 SB(st);
14 SR(st);
15 ARK(st,Word[40,43]);
16 output=st
17 end

```

In the above Algorithm SR,SB,Mix,ARK stands for ShiftRows, SubBytes, Mix-Columns, AddRoundKey respectively.

The hardware utilisation of AES module is explained in below figure 3.8. This AES module is consuming 3532 LUTs and 2564 slice Registers.

Name	1	Slice LUTs (41000)	Slice Registers (82000)	F7 Muxes (20500)	F8 Muxes (10250)	Bonded IOB (300)	BUFGCTRL (32)
>  aes_(mkAES)		3532	2564	194	64	0	0

Fig. 3.8: Hardware utilization of AES

CHAPTER 4

Architecture

This section describes the complete architecture of True Random Number Generator. This section describes each and every block of Trng briefly. A block diagram is shown in figure 4.1.

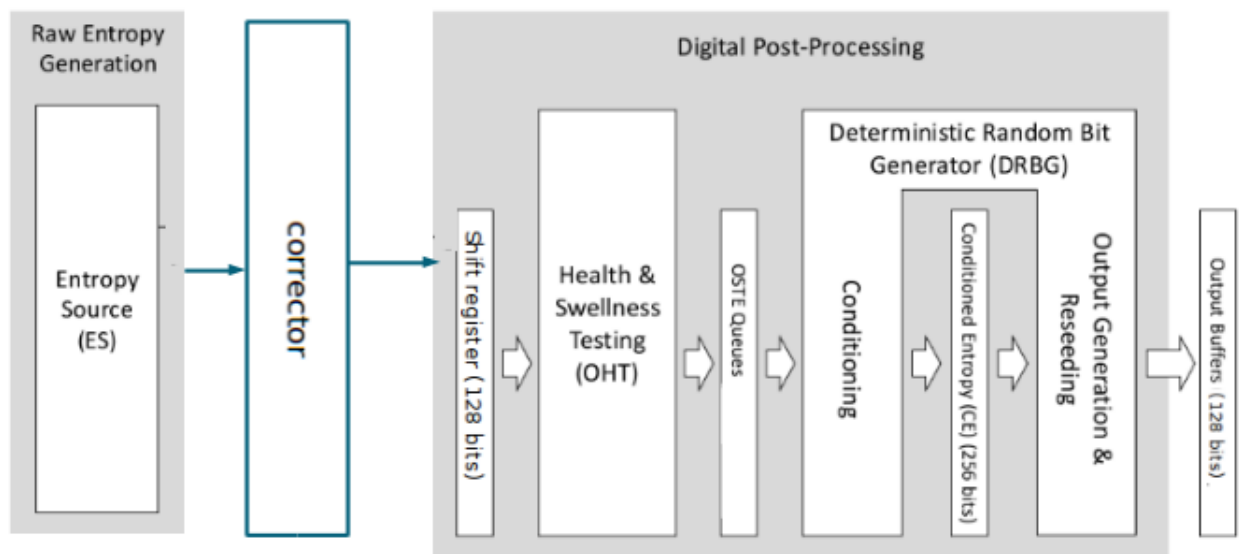


Fig. 4.1: Block diagram of RNG

Most RNGs comprises of an Entropy source followed by corrector and Digital post processing, which includes health tests, conditioning and DRBG. Entropy source output typically has visible biases and other aberrations that differentiate it from random binary data. The goal of the post-processing logic is to turn this raw output into random data with a lower bitrate but greater quality.

The DRBG proposed in this paper is cryptographically very strong. This DRBG is strong in such that it produces quality numbers even if the entropy source is completely degraded. To prevent exploitable holes caused by flaws in the entropy source, the post-

processing employs strong cryptography. The RNG, in particular, keeps an entropy pool that is seeded with a huge quantity of data from the ES.

The proposed RNG work as follows

- The Entropy sources continuously produces bits with constant speed, this bits are used for seeding DRBG.
- Random bits generated by Entropy source are corrected by corrector which approximately decreases the bit rate by $\frac{1}{4}$.
- The bits corrected from the corrector are grouped into 128 bits are moved to lower OSTE[127:0] followed by partially conditioning. Full conditioning needed 256 bits of OSTE.
- The 256 bits collected from corrector (2 times) are processed to full OSTE buffer.
- OSTE buffer undergo health tests and completes conditioning (waiting for upper oste) parallelly. The 256 bit values are moved to CE buffer after conditioning if it passes health test.
- If OSTE buffer passes health test then the conditioned values are used for seeding DRBG, else conditioning process repeats until health test passed or fresh oste collected from corrector.
- The DRBG generates the outputs in blocks of 128 bits after seeding.

CHAPTER 5

Blocks of TRNG

This chapter describes about each and every block in detail.

5.1 Ro based Entropy source

Oscillators are a simple and effective way to generate Entropy source. A digital oscillator is designed by connecting odd number of inverters in ring manner. Ring oscillator output fluctuates in between zero and one and forms a square wave due to the feedback route. hence we obtain a square wave with time cycle T . The diagram of ring oscillator is shown in figure 5.1.

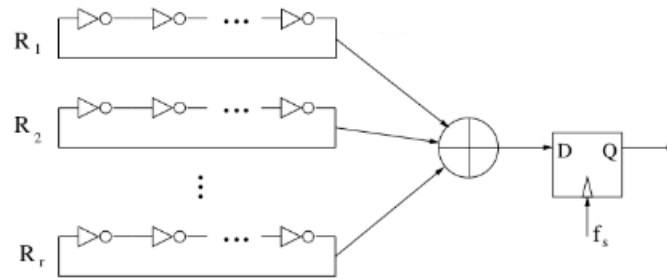


Fig. 5.1: RO based Entropy source

The characteristics of square wave produced by ring oscillator are

- The output signal O should ideally be a square wave with certain period. The period of the generated wave is determined by inverters number and delay of one inverter. That is $T = n\tau$ $O(t) = O(t + T)$, where T represents the time period and τ signifies the one inverter delay.
- Square wave produced by Ro is not exactly square wave since designed inverters are not ideal. The jitter is produced when the square is rising from 0 to 1 or falling from 1 to 0.
- The generated wave behaves in random manner in time domain $t = T + T''$, where T'' is considered as a random variable, which takes noise values when square wave is falling or rising.

- Jitter is the fluctuation(vibration) in the square wave represented by T'' , random variable. T'' is very small compared to T (approximately 4 percent).

The general output of square wave with jitter shown in below figure 4.2.

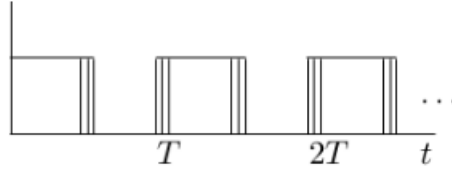


Fig. 5.2: Periodic square wave with jitter marked as lines

5.1.1 Combining Ro outputs to exploit randomness of phase jitter

We have r distinct number of ring oscillators, where each ring consists of fixed odd number of inverters to generate square wave. The r no of ring oscillators produce r square waves. The output of r oscillators Xor-ed to produce a single wave, which is sampled with sampling frequency f_s regularly.

The notion of sampling the output of a Xor-ed wave of ring oscillators is used to create a feasible arrangement for harvesting jitter. The jitter is produced in complete time interval by Xor-ing r number of oscillators based on this below points.

- It's difficult to match the period of the two oscillators exactly, special VLSI layout techniques used for matching
- The two signals may wander relative to one another due to flaws. TRNG designs become extremely vulnerable as a result of this.

Each ring's periodic transition zones are combined in the XOR's output. Because the full waveforms are XORed, the result will contain predictable areas as well as zones where jitter from distinct rings overlaps. Our goal is to fill the full spectrum with transition zones at each and every point in time interval followed by sampling. Sometimes the jitter produced by different Ros may overlap in this cases may lost entropy so we should tackle this problem selecting appropriate number of ring oscillators.

5.1.2 Urn model to detect no of Ros

A random bit stream is generated by sampling the combined signal. On signals that are precisely between 0 and 1, the XOR gate is designed to operate as a smooth interpolation. The generated xored signal is expected to be in the range of low and high volts, with low less than high. Supposing $T=T_j$ is the period of an oscillator ring R_j . Ring oscillator produces a square wave which rises from low to high at time intervals $T/2, 3T/2, \dots$ and high to low from $T, 2T, 3T, \dots$

Assumption: There exists a single moment t , where the signal value cuts $(\text{low} + \text{high})/2$ volts in a time interval ranging from $(mT - T/4, mT + T/4)$. This t behaves as variable, which is random with mean mT and variance σ_j^2

Let us take a time interval $I = [c, d]$ and assume threshold voltage as 2.5, which is average of low and high. The voltage value low and high denote the value of a voltage that is recognised as logic 0 and logic 1 by a certain technology. Our objective is to use a composite signal made up of the outputs of k oscillator rings to fill up this interval with unpredictability. We want to make sure that for every threshold, at some time interval t in I , there exists a ring oscillator R_j such a way that $.25 < P[O = 5/2] < .6476$. It suffices to follow this condition that, for any integer m , $|t - mT_j| < 1.224\sigma_j$ with probability q . The value of q should be 1 in ideal case.

Criterion A: The probability that there exists a two integers j and m with the condition $|t - mT_j| < .6476\sigma_j$ is with q in least case, where t is chosen randomly over the interval I with uniform distribution.

Let us consider a time interval I . This time interval divided into equal length parts (sub intervals) in a way that the criterion A is fulfilled with probability $q=1$. Later we relax the q to less than 1. For any time t in sub interval J there should be at least one ring oscillator which is in transition zone. The time interval should be splitted in a way that it should satisfy condition that whenever a transition occurs in J (sub interval) it should make the criterion A satisfies with probability $q=1$. We call each sub interval j called urn. Let's divide the interval $I = [c, d]$ into subintervals J_1, \dots, J_l with same length. If there exists a ring

oscillator R_j in design, whose signal fulfils $O_j(t) = 5/2$ for some value t in sub interval J , it is called full, else empty. The entropy of entropy source depends upon on number of subintervals called urns explained in table 4.1.

Table 5.1: Number of urns for certain Entropy

Target entropy	Tolerance	number of urns
.98	$.144\sigma$	172
0.96	$.256\sigma$	95
0.94	$.337\sigma$	73
0.91	$.481\sigma$	50
0.79	$.691\sigma$	34
0.51	1.224σ	19

Criterion A is satisfied provided that

- $\text{lenght} > (d-c)/.6471\sigma_j$ for 0.8 bits of entropy.
- The urns filled should be grater than $q \cdot \text{length}$.

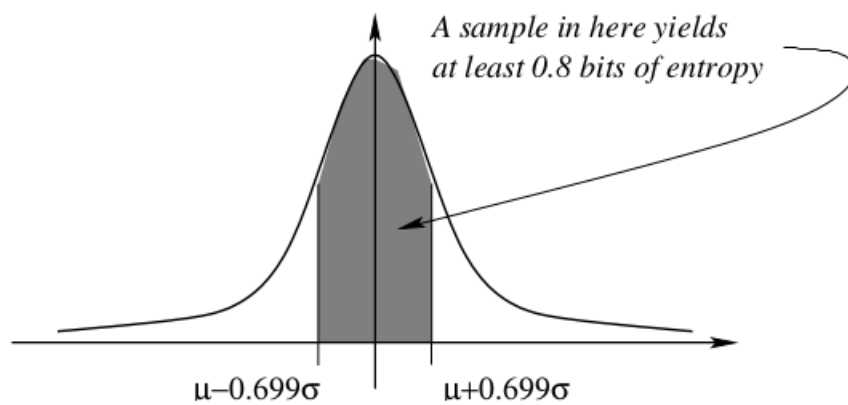


Fig. 5.3: A single jitter event

The ring oscillator R_j fills one time out of each $\Pi_j = lT_j/(d-c)$ urns(Π_j is combinatorial period of ring oscillator). There exists a some probability that overlap of transitions produced by two ring oscillators. Due to the non-deterministic aspects (phase drift and initial time delay), it is exceedingly unlikely that two same ring oscillators would contribute to the jitter in the same way. single transition caused by ring oscillator is shown as normal distribution in figure 4.3.

Filling of jitter in Interval I using similar length ring oscillators is related to combinatorial urn model. This urn model is called coupon collectors problem. There are N distinct coupons, but each one has an endless supply. A coupon collector gathers one coupon every day, selecting one at random from the N coupons, with replacement. The goal is to figure out how many days (r) will complete before the coupon collector (time interval) has filled with at least one copy (one transition) of each and every of coupons (N).

We would want to find the lowest number of rings $r = M(N, f, p)$ required for N number of urns, a specified fill rate which is in range between $0 < f < 1$ and with confidence of probability $0 < p < 1$, so filling N urns with fill rate of f and p probability. For $f = 1$, According to coupon collectors formula We need to use $N \log_2 N$ ring oscillators to fill N number of urns in order to satisfy the criterion A with $q=1$.

For this designed TRNG the digital post processing expects atleast 0.5 bits of entropy. if we consider .9 bits of entropy the urn width should be 0.335σ therefore we need atleast 50 urns. The number of inverters in ring oscillators are 5 in this proposed entropy source. For exact 0.9 bits of entropy the fill rate should be 1, so we need $50 \log_2 50$ no of Ros. We need atleast 0.5 bits of entropy here so let us assume we need 0.5 then we can decrease the fill rate to 0.6, for this fill rate we need 36 Ros. In this entropy source we are using 36 ring oscillators, where each Ro consists of 5 number of inverters.

LUT	FF	BRAMs	URAM	DSP
228	0	0.00	0	0
23	0	0.00	0	0

Fig. 5.4: Hardware for entropy source

This proposed entropy source consuming 228 Luts and 1 flipflop.

5.2 Von Neumann Corrector

The von Neumann corrector is most well-known and widely used postprocessing methods for removing localised biases. It extracts bit pairs from a random bit stream. It eliminates them from the stream of random bit pairs if they have the same value ('11' bits or '00' bits). If pair of bits are different then it takes one of bit from them (starting bit or ending bit). For example if is 10, consider it as 1 if is 01 take it as 0 (considering first bit). This corrector decreasing the speed of the bitrate by some rate if we consider it on average it is decreasing by 1/4. The von Neumann corrector [von Neumann (1963)] has a significant benefit in that it is simple to implement. The corrector is shown in table 4.2.

Table 5.2: Von Neumann Corrector

Random bit pair	corrected bit
00	reject
01	0
10	1
11	reject

5.3 Health checks

Health checks are an important aspect of the TRNG architecture since they guarantee that the noise source and the overall entropy source continue to function properly. The purpose of evaluating the entropy source is to ensure that faults of the ES are detected fastly and with a high likelihood. Noise sources can be delicate, and as a result, they might be impacted by changes in the device's working circumstances, such as temperature, humidity, or electric field, resulting in unanticipated behaviour, so we need to check the health of entropy source regularly.

Before any conditioning, the outputs of a noise source are subjected to health testing. Because these tests are executed continually on all digitised samples collected from the noise source, they must have a very low chance of generating a wrong alarm during correct operation. Health tests includes 2 tests based on this paper[Kelsey (2018)] they are Repetition Count test, and the Adaptive Proportion test.

The data corrected from Von Neumann Corrector is grouped into 256 bits, which is stored in Online Self Test Entropy(oste) buffer. Each 256-bit sample's health is assessed by the health check unit. Here we discuss detailly about adaptive proportion test and repetition count test.

5.3.1 Repetition Count Test

This test is designed to detect whether entropy source is stuck at either 0 or 1. Catastrophic failures which are reason for stuck test are detected by using this test. It's a modernised version of the "stuck test" that was once necessary for RNGs. This test goal is to detect a complete defection of noise source. If a sample is repeated C or more times, the test announces an error.

The samples here we consider are 0 and 1. if we get continuous zeros or ones then test raises alarm that the entropy source is stuck at either 0 or 1. The lowest number meeting the condition $\alpha \geq 2^{\frac{1}{H(C-1)}}$ assures that the chance of receiving a series of identical values from C successive noise source samples is at most. α is acceptable false-positive probability(chance of showing true as false). If we take $H=0.5, \alpha = 2^{-8}$ then from that equation we get C value 16. Therefore, if we get 16 continuous zeros or ones we can raise a alarm.

5.3.2 Adaptive Proportion Test

This test is used to detect a significant loss of entropy caused by a hardware failure or an environmental change that affects the noise source. The test continuously monitors the frequent occurrence of samples, which are generated by noise source to see whether it happens too often. As a result, the test may detect when a value occurs substantially

more frequently than predicted. Rather than the type of total failure found by the Repetition Count Test, this test is designed to detect more environmental failures of the noise source.

The collected samples from noise source are tested and counts how many times the same value appears in the subsequent samples. The test reports an error if the count crosses the cutoff value C . The cutoff value C is calculated using α (acceptable false probability) and no of bits in sample. Let $\text{next}()$ gives the next set of samples from noise source and C is a cutoff value, The adaptive test is performed as explained in algorithm.

Algorithm 4.1: Adaptive Proportion Test

```

1  X=next();
2  Y=1;
3  For i=1 to l(number of n bit samples in 256 bits)
4    if(X=next()) Y=Y+1;
4    if(Y>C) return failure
6  go to step 1.

```

In this paper we selected 6 samples and calculated cutoff values for those 6 specific samples. The 6 samples are 1,01,010,101,0110,1001. The cutoff values are calculated and mentioned in below table 4.3 are referred from intel paper [Mike Hamburg (2012)].

Table 5.3: Health bounds for 256-bits

Sample	Allowable number of occurrences per 256-bit sample
1	$109 < m < 165$
01	$46 < m < 84$
010	$6 < m < 58$
0110	$2 < m < 40$
101	$6 < m < 58$
1001	$2 < m < 40$

The likelihood that a random sample from a homogenous population The percentage of samples who fail their health checkups is around 1 percent. The health checks aren't meant to be a complete entropy measurement. Instead, they wanted to see if the entropy source was severely damaged, and whether it was stuck producing basic repeating patterns like all ones, all zeros. This health tests checks the 256 bit samples if it is healthy, the 256 bit samples transferred to 256 bit CE buffer.

5.4 Conditioning

Noise source are not ideal, therefore they do not produce pure quality random numbers, there is always a chance for existence of biasing, so there should be one method to eliminate biasing called conditioning. Debiasing of biasing bits produced by entropy source called conditioning. Conditioning sometimes reduces the bit rate and entropy, the ultimate aim of conditioning is to make the bits more random. The conditioning uses cryptographic algorithms which includes SHA and AES. In this paper we use a AES-128 bit algorithm for conditioning.

In this project, for conditioning bits are collected from corrector. The conditioning divided into two parts, they are conditioning of upper CE and lower CE. The bits from corrector grouped into 128 bits and transferred to lower OSTE[127:0]. Here after collecting Lower OSTE bits we start conditioning without waiting for OSTE upper bits. We can complete 66 percent of first half conditioning without upper oste. After receiving next group of 128 bits from corrector, we move those 128 bits into upper oste. If OSTE is filled with 256 bits, then we continue the remaining conditioning parallelly we perform the health tests. If the OSTE 256 bits are healthy they are moved to CE, this 256 bits are used for seeding or reseeding the DRBG, else conditioning is repeated 4 more times. Each time after one conditioning health check is performed, if it found healthy it is used for seeding DRBG. If health check fails 4 continuous times after each conditioning, then OSTE value is discarded and wait for fresh OSTE value.

The process of conditioning is explained step wise below. First, the lower half of CE is updated using the OSTE value. The key k' used in this conditioning is non secret fixed

key, which is of 128 bits. This process requires 6 AES-128 bit operations, three each for updating upper and lower CE. For Updating lower CE involves three steps as shown below.

- $\text{Tmp}[127:0] = \text{AES}(k', \text{CE}[127:0]);$
- $\text{tmp} = \text{AES}(k', \text{OSTE}[127:0] \oplus \text{tmp});$
- $\text{CE}[127:0] = \text{AES}(k', \text{OSTE}[255:128] \oplus \text{tmp});$

The conditioning process clearly explained in the below figure 5.5.

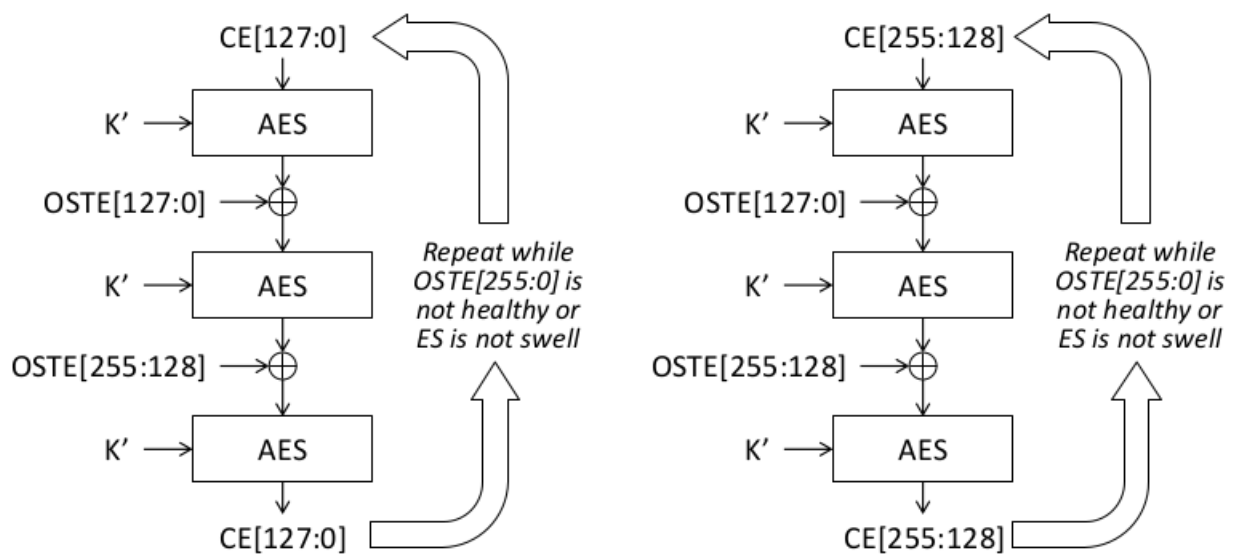


Fig. 5.5: Updating CE

The updating upper CE(CE[255:128]) requires 3 AES operations and 2 bit wise xor operations as explained in below steps.

- $\text{Tmp}[127:0] = \text{AES}(k'', \text{CE}[255:128]);$
- $\text{tmp} = \text{AES}(k'', \text{OSTE}[127:0] \oplus \text{tmp});$
- $\text{CE}[255:128] = \text{AES}(k'', \text{OSTE}[255:128] \oplus \text{tmp});$

The main advantage of updating is make CE more random, if there is any uncertainty in OSTE. let us assume half of the bits of OSTE bits are random (128 of 256 bits). In updating we are applying AES algorithm to FULL OSTE and CE followed by xor operations of lower OSTE and upper OSTE, so by chance if one of the bit of lower OSTE is not random, it may become random since we are xoring it with upper OSTE. Therefore

we can assure that,if one of the similar position of lower or upper OSTE is random(for example 0/128,..5/133..127/255),it becomes random since we are performing AES and xor operations.Therfore we are increasing randomness in conditioning.

If the entropy source producing at least 0.5 bits of entropy(128 out of 256 bits are random) then we can assure that CE is probably random.conditioning never increases the entropy,it just increases the randomness by performing operations.

5.5 Counter based DRBG based on Block Ciphers

Counter DRBG[Elaine Barker (2015)] based on Block cipher involves AES operations.In this DRBG we are using AES-128 encryption Algorithm.The DRBG operates based on DRBG mechanism.The DRBG mechanism involves 3 steps they are instantiate,generate and reseeding function.The mechanism is shown in fig 5.6.

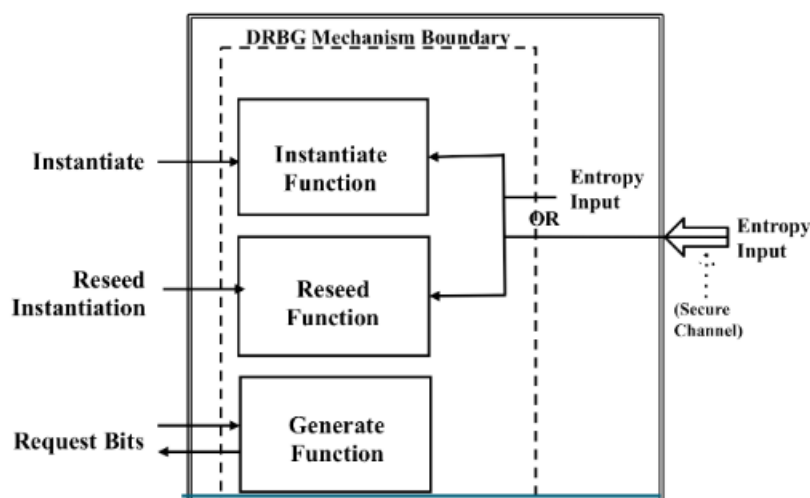


Fig. 5.6: Mechanism of DRBG

The mechanism explained step wise below

- The instantiate function takes the input from entropy source(here it is conditioning),which is used for seeding DRBG.
- The generate function produces output after seeding.It generates number of bits based on request.

- The final stage is reseed. The reseed function takes new input from entropy source and combines it with previous states to produce a new seed or directly entropy source output is used to seed.

The proposed Counter DRBG uses a approved encryption algorithm(Advanced encryption Algorithm). The same cipher algorithm and size of key used for all operations of DRBG. The algorithm and key size should meet the security constraints of DRBG.

The counter DRBG consists of internal state, which is updated with new entropy value while reseeding. The internal state consists of three values.

- V is a input to the cipher(AES) algorithm, V is of Blocklen 128 bits, which is used as internal state.
- K is a key to the cipher algorithm of 128 bits, which is used as internal state.
- The value of reseed counter.

5.5.1 Instantiate

The instantiate function uses different seed value whenever it uses for seeding DRBG. The DRBG continuously need seeding in order to tolerance against attacks. The initial seeding is explained in below equations. The instantiating function is explained in below figure 5.7.

$V[127:0] = CE[255:128]; K[127:0] = CE[127:0];$

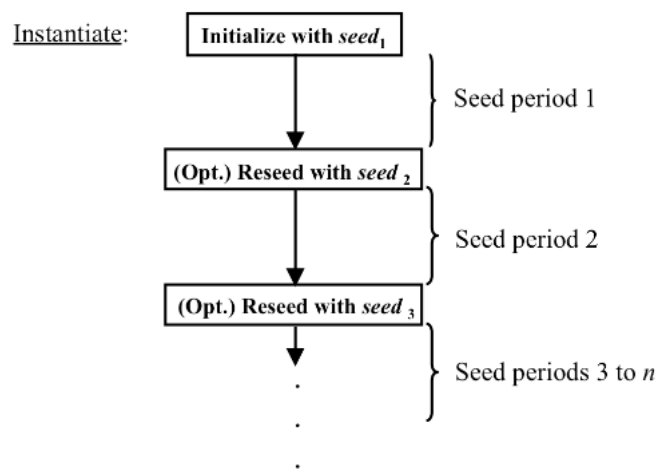


Fig. 5.7: Instantiate of DRBG

5.5.2 Generation

The generation function produces random bits whenever it is instantiated or reseeded. The generate function takes the values V of 128 bits as input to cipher and K as key to the cipher Algorithm. The generate function performs the following steps.

- Takes the V and K (key) value from CE register. The lower $CE[127:0]$ is used as K . The upper $CE[255:128]$ is used as input to the cipher algorithm
- it checks the life of seed value, for this DRBG the life time of seed is 2^{19} bits. For proposed DRBG we are taking 2^{17} as seed life for safety. If generated bits exceed the seed life the DRBG stop producing output.
- Returns the values of the V, K and reseed counter while updating new seed value.
- Returns the Generated output bits.

The pseudo code for Generation algorithm is explained below and shown in figure 5.8.

```

1 Counter=0;
2 if(Counter<512)
3 begin
4   $V[15:0] = (V[15:0] + 1) \bmod 2^{16}$ ;
5  counter=counter++;
6  Output=AES(K,V);
7 endif

```

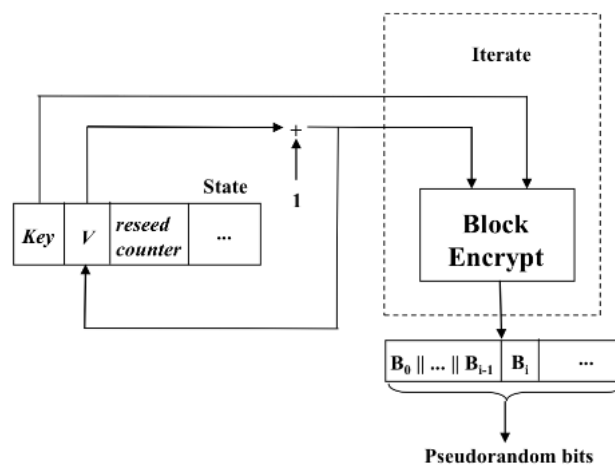


Fig. 5.8: Generation of DRBG

The DRBG may reseed before the seed life(2^{17}) of seed. If every block produced by entropy source is healthy then it reseeds after producing 30 128 bit outputs. If DRBG not seeded after generating 2^{17} bits, then it waits until healthy seed is produced.

5.5.3 Reseeding

The DRBG requires seeding frequently, since there is a chance to predict value based on the previous outputs. The seed life is 2^{17} bits for this proposed DRBG.

The reseed function performs the following steps.

- Takes the input CE[255:0] value from the conditioning.
- Takes the internal state Values from Generate function such as V, K.
- Combines the CE value with V and K to generate new seed using update function. New seed is used for reseeding DRBG.

The pseudo code for updating used is explained below.

```

1 V,K=Fun(Gen)
2 tmp=AES(V,K)
3 V[15:0] = (V[15:0] + 1) mod 65536;
4 K=CE[127:0]  $\oplus$  tmp;
5 V=CE[127:0]  $\oplus$  tmp  $\oplus$  CE[255:128];

```

CHAPTER 6

Implementation of TRNG

The True random number Generator code is designed in Bluespec verilog code. The bsv code is converted into verilog, which is synthesized in vivado tool. Vivado tool calculates the hardware (number of LUTs and Slice registers) required for designed code. The designed code is optimised in terms of area and speed. In this code we optimised one at a time. Optimisation of speed consuming more number of LUTs and FFs. Optimisation of Area consuming more number of cycles (i.e. less speed).

6.1 Implementation using two AES hardware modules

In this design we use two AES hardware module. The conditioning uses one AES hardware module and DRBG uses one hardware module. The conditioning performs 6 AES operations one after other using single AES module. The DRBG uses one AES module, which generates output. While one AES module generates output, parallelly other AES module used for conditioning.

The speed is optimised here, so one can use this TRNG if their main objective is speed. This design is taking more Area as we are using 2 AES modules. The number of cycles for generating 1024×128 random numbers is 40138. LFSR library is used instead of entropy source here since we can't obtain a proper Entropy (noise) source output in software. The number of cycles shown in Figure 6.1.

```

Bluespec Workstation - trng_test
Project Edit Build Tools Window Message
count is 3
value is 81732800481641963971562793032324072905
count is 1021
value is 245305919576011656961706352005666051248
count is 1022
value is 334081374519565158401328384448465942223
count is 1023
value is 334081374519565158401328384448465942223
number of cycles 40138
Simulation finished

```

Fig. 6.1: Number of cycles for 1024*128 random numbers

6.1.1 Synthesis Report

The designed TRNG is consuming 9686 LUTs, 7329 registers and muxes. The number of LUTs for 2 AES hardware modules is 6668 and number of slice registers 2 AES modules consuming is 5127. Most of the Area is consumed by 2 AES modules and remaining part of Area is consumed by entropy source and health checks. The entropy source and corrector is consuming 401 LUTs and 158 registers. The health checks consuming 1384 LUTs and 256 registers. The hardware utilisation shown in figure 6.2.

Name	1	Slice LUTs (63400)	Slice Registers (126800)	F7 Muxes (31700)	F8 Muxes (15850)	Bonded IOB (210)	BUFGCTRL (32)
mkTrng		9686	7329	386	128	136	1
co_ab_ab (mkTop)		1384	256	0	0	0	0
co_aes_ (mkAES)		4014	2563	192	64	0	0
co_aes_1 (mkAES_0)		3543	2564	194	64	0	0
ent (mkEnt)		397	140	0	0	0	0
roo (roo)		249	0	0	0	0	0

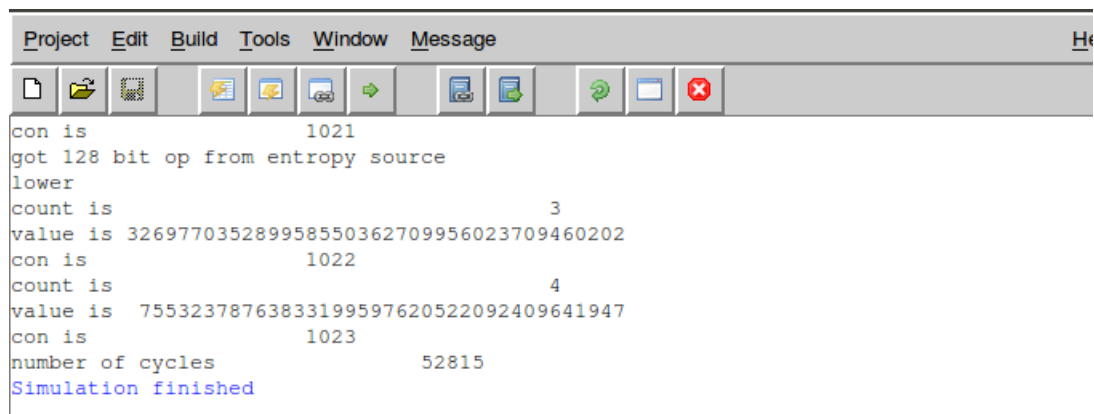
Fig. 6.2: Hardware utilisation of TRNG(2 AES modules)

6.2 Implementation using only one AES hardware module

In this design we use single hardware AES module. The same AES hardware module is used both for conditioning and DRBG. Conditioning performs 6 AES operations where as DRBG continuously uses AES module except when it is used for conditioning. First

priority for AES module here is conditioning. If generated bits from entropy source are healthy, then AES is used for conditioning, else AES module is used for DRBG.

The area is optimised here, so one can use this TRNG if their priority is area. The speed is less compared to above designed module. The number of cycles for generating 1024×128 random numbers is 52815. There is a difference of 12700 cycles for generating 1024×128 bits. The number of cycles to generate 1024×128 bits is shown in figure 6.3



```

Project Edit Build Tools Window Message
con is 1021
got 128 bit op from entropy source
lower
count is 3
value is 326977035289958550362709956023709460202
con is 1022
count is 4
value is 75532378763833199597620522092409641947
con is 1023
number of cycles 52815
Simulation finished

```

Fig. 6.3: Number of cycles for 1024×128 random bits

6.2.1 Synthesis Report

The designed TRNG is consuming 6287 LUTs, 4779 registers and muxes. The number of LUTs and registers consumed by AES module along with all other hardware in conditioning is 4305 and 2564 respectively. The health checks consuming 1362 LUTs and 256 registers. The entropy source and corrector is consuming 401 LUTs and 158 registers. The hardware utilisation is shown in figure 6.4.

Name	1	Slice LUTs (63400)	Slice Registers (126800)	F7 Muxes (31700)	F8 Muxes (15850)	Bonded IOB (210)	BUFGCTRL (32)
mkTrng		6280	4754	194	64	136	1
co_ab_ab (mkTop)		1355	254	0	0	0	0
co_aes_ (mkAES)		4309	2564	194	64	0	0
ent (mkEnt)		397	140	0	0	0	0

Fig. 6.4: Hardware utilisation of TRNG (1 AES module)

6.3 Results

6.3.1 BSV Simulation results

The design was implemented using BSV. The results shown below are using LFSR library in BSV instead of proposed entropy source, since code can't produce random outputs. The results are shown in figures 6.5, 6.6 and 6.7. The results shown below are for design of TRNG using two different AES modules. The results shown in first figure(6.5) are outputs just after stating TRNG. The variable con is counting number of outputs produced by TRNG at given time and the variable count is counting number of outputs after seeding TRNG by new seed. The variable value is output in the blocks of 128 bits produced by TRNG.

```
count is 1
value is 308348967484074539229267218018298001920
con is 0
count is 2
value is 134925386311119610279059570589036750143
con is 1
count is 3
value is 152789450637463034924152365597588037851
con is 2
count is 4
value is 9925761469358416668996341650566993888
con is 3
count is 5
value is 52785568626898782258382415614101409361
con is 4
count is 6
value is 12426340007180766869175312610925508392
con is 5
got 128 bit op from entropy source
count is 7
lower
value is 276499850093950496055723851783682661420
con is 6
count is 8
value is 187175146113381513288903454932199752727
con is 7
count is 9
value is 94528182189631665304957814171097865421
con is 8
```

Fig. 6.5: Simulation results of TRNG

The results shown in figure 6.6 are example of reseeding TRNG. In this picture we could able to see DRBG is successfully seeded after producing 27*128 bits by observing the value of the variable count value in figure as it became 0.

```

con is                20
got 128 bit op from entropy source
count is              22
upper
value is 154677171122746261791740813070297252090
con is                21
count is              23
value is 34438382463691692035638525298101533442
con is                22
count is              24
value is 58899683403174114676428493831742023679
con is                23
count is              25
value is 268660200757936269774402126069655200136
con is                24
count is              26
value is 252153965301122856140079832910513139931
con is                25
count is              27
value is 273314313489873611834161840155635802326
con is                26
cond c7 b84d8db4a8add1452ec16aafel6956fa,1
seedsf
count is              0
value is 35117088738524566713626645867486088141
con is                27
count is              1
value is 183894355873400663391878669653599257963
con is                28
count is              2

```

Fig. 6.6: Simulation results of TRNG

The results shown in figure 6.7 are example of reseeding TRNG. In this picture we could able to see DRBG is successfully seeded after producing 24×128 bits by observing the value of the variable count value in figure as it became 0. The total bits produced by TRNG are 298×128 bits by observing value of variable con value as it is 298.

```
count is 18
value is 69200298843965308704638612211769843895
con is 292
got 128 bit op from entropy source
upper
count is 19
value is 162034627698851319208839205577700826114
con is 293
count is 20
value is 99380398098235567938055021669582180662
con is 294
count is 21
value is 286705270167017234140204258946631943056
con is 295
count is 22
value is 147154473799410419400620811139768918666
con is 296
count is 23
value is 95946555665407788387328457235711324043
con is 297
count is 24
value is 204688446227925970938397652971225089559
con is 298
cond c7 74bcc36d92393c27c2a3a474ec37d00b,1
seedsf
count is 0
value is 284536043072279798928774552591103310933
con is 299
count is 1
value is 236377973776198859407247400322377432427
con is 300
```

Fig. 6.7: Simulation results of TRNG

REFERENCES

1. **E.Barker** and **J.Kelsey** (2012). *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*. NIST Special Publication 800-90A.
2. **Elaine Barker, J. K.** (2015). *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*. NIST Special Publication 800-90A.
3. **FIPs** (2001). *ADVANCED ENCRYPTION STANDARD (AES)*. Federal Information Processing Standards Publication 197.
4. **Kelsey, M. S. T. E. B. J.** (2018). *Recommendation for the Entropy Sources Used for Random Bit Generation*. NIST Special Publication 800-90B.
5. **Mike Hamburg, P. K.** (2012). *ANALYSIS OF INTEL'S IVY BRIDGE DIGITAL RANDOM NUMBER GENERATOR*. Cryptography Research.
6. **Shannon, C.** (1948). *A Mathematical Theory of Communication*. Bell System Technical Journal, vol. 27, pp. 379–423, 623-656.
7. **von Neumann, J.** (1963). *Various techniques for use in connection with random digits, von Neumann's Collected Works*. vol. 5, Pergamon, pp. 768–770.