UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ENGENHARIA DE SÃO CARLOS

LEONARDO BOSCO CARREIRA

GERADOR DE NÚMEROS ALEATÓRIOS DIGITAL, RECONFIGURÁVEL, DE BAIXA LATÊNCIA COM DETECÇÃO E CORREÇÃO DE VIÉS DE SAÍDA

São Carlos
2019

LEONARDO BOSCO CARREIRA

Gerador de números aleatórios digital, reconfigurável, de baixa latência com detecção e correção de viés de saída

Trabalho de conclusão de curso apresentado à Escola de Engenharia de São Carlos da Universidade de São Paulo.

Curso de Engenharia Elétrica com ênfase em eletrônica.

Orientador: Prof. Dr. Maximiliam Luppe

São Carlos
2019

AUTORIZO A REPRODUÇÃO TOTAL OU PARCIAL DESTE TRABALHO,POR QUALQUER MEIO CONVENCIONAL OU ELETRÔNICO, PARA FINSDE ESTUDO E PESQUISA, DESDE QUE CITADA A FONTE.

Ficha catalográfica elaborada pela Biblioteca Prof. Dr. Sérgio Rodrigues Fontes da EESC/USP com os dados inseridos pelo(a) autor(a).

Bosco Carreira, Leonardo
B314g GERADOR DE NÚMEROS ALEATÓRIOS DIGITAL, RECONFIGURÁVEL, DE BAIXA LATÊNCIA COM DETECÇÃO E CORREÇÃO DE VIÉS DE SAÍDA / Leonardo Bosco Carreira; orientador Maximiliam Luppe. São Carlos, 2019.

Monografia (Graduação em Engenharia Elétrica com ênfase em Eletrônica) -- Escola de Engenharia de São Carlos da Universidade de São Paulo, 2019.

1. Gerador de números aleatórios. 2. reconfigurável. 3. baixa latência. 4. detecção . 5. correção de viés. 6. oscilador em anel. I. Título.

Eduardo Graziosi Silva - CRB - 8/8907

FOLHA DE APROVAÇÃO

Nome: Leonardo Bosco Carreira

Título: "Gerador de números aleatórios digital, reconfigurável, de baixa latência com detecção e correção de viés de saída"

Trabalho de Conclusão de Curso defendido e aprovado em 12 / 07 / 2019,

com NOTA 9,0 (nove, zero), pela Comissão Julgadora:

Prof. Dr. Maximiliam Luppe - Orientador - SEL/EESC/USP

Prof. Dr. João Navarro Soares Júnior - SEL/EESC/USP

Mestre Jovander da Silva Freitas - Mestrado/EESC - Instituto Federal de São Paulo/Campus Barretos

Coordenador da CoC-Engenharia Elétrica - EESC/USP:
Prof. Associado Rogério Andrade Flauzino

Dedico essa monografia ao meu pai, Venicio, minha mãe Sirlei, minha irmã Marina e todos familiares que com muito amor e carinho não mediram esforços para me dar todo suporte para que eu chegasse aqui.

Em memória de André Lopes Campos.

AGRADECIMENTOS

Agradeço primeiramente a minha família que esteve ao meu lado durante as mais difíceis decisões desses anos.

Aos meus amigos, que fizeram parte dessa jornada, em especial para república Toca da Raposa onde morei meus melhores anos da graduação e aos amigos de longa data de Ribeirão Preto, Diego, Matheus, Felipe e Leonardo que contribuíram nessa jornada.

A minha psicóloga, Dra. Elizabeth que foi muito importante para me manter focado nos meus objetivos mesmo com os problemas enfrentados esse ano.

Ao meu orientador Maximiliam Luppe e ao Professor Subhanshu Gupta que deram todo suporte necessário para desenvolvimento dessa monografia.

Por último e não menos importante a todos os professores e funcionários da Universidade de São Paulo.

“Segue o teu destino, rega as tuas plantas, ama as tuas rosas. O resto é a sombra de árvores alheias. A realidade sempre é mais ou menos do que nós queremos. Só nós somos sempre iguais a nós-próprios. Suave é viver só. Grande e nobre é sempre viver simplesmente. Deixa a dor nas aras como ex-voto aos deuses”

Fernando Pessoa

RESUMO

CARREIRA, L. B. **Gerador de números aleatórios digital, reconfigurável, de baixa latência com detecção e correção de viés de saída**. 2019. 198 f. Dissertação – Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2019.

O crescente número de dispositivos conectados à rede da internet das coisas (IoT) traz consigo também uma maior preocupação com a vulnerabilidade dos usuários. Geradores de números aleatórios tem um papel fundamental na proteção da comunicação homem-máquina ou máquina-máquina, sendo componentes indispensáveis nos *hardwares* dos dispositivos conectados. Porém, mais do que servir como geradores de chaves para encriptação e compressão de mensagens, há cada vez mais necessidade que estes dispositivos tenham a capacidade de detectar e corrigir qualquer viés induzido na saída, por fontes naturais como variação de temperatura ambiente e envelhecimento de componentes, ou forçadas por terceiros que desejam ter acesso ao dado transmitido garantindo um rápido reestabelecimento de uma conexão segura. Este trabalho apresenta um gerador de números aleatórios reconfigurável, com sistema de detecção e correção de viés desenvolvido em uma FPGA Altera Cyclone V. O projeto proposto utiliza como fonte de entropia o *Jitter* presente em osciladores em anel, realimentado através de um processador hospedeiro com sistema de aprendizado estatístico que garante reconfiguração da fonte de entropia. Resultados demostraram a capacidade de recuperação de até 87ms a 6KHz e um consumo de potência de 80mW do TRGN e 180mW do sistema de detecção e correção.

Palavras-chave: Gerador de números aleatórios, reconfigurável, baixa latência , detecção e correção de viés, oscilador em anel.

ABSTRACT

Bosco Carreira, L. B **Low-latency reconfigurable true random number generator with bias correction and detection**. 2019. – Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2019.

The dramatic growth of the number of devices connected to the internet of things network (IoT) brings more concerns about the users' vulnerability. Random number generators play an important role in the protection of human-to-machine and machine-to-machine communications, being crucial components in all connected devices. However, yet being useful as encryption keys generators and compressive sensing seeds even more is being required that these devices also have the capability to fast detect and correct bias variation introduced by natural sources, as temperature variation and electronic components aging, or introduced by undesirable third parties trying to get access to the transmitted data. This work presents a reconfigurable true random number generator (TRNG) built in with a low-latency bias detection and correction method developed in a FPGA Cyclone V. The proposed system uses a ring oscillator based TRNG controlled by a statistical learning-based host processor. Results have shown correction capability up to 87ms at 6KHz. The measured power consumption of the TRNG and the bias correction HP is 80mW and 180mW respectively at 1.25V with a 18KHz throughput for three random channels.

Keywords: Random number generators, reconfigurable, low-latency, bias detection and correction, ring oscillator.

LISTA DE ILUSTRAÇÕES

Figura 1 - Previsão de crescimento de dispositivos conectados à internet das coisas (IoT).................31

Figura 2 - Distribuição de referência e valores críticos de rejeição da hipótese nula 34

Figura 3 - Distribuição de referência do coeficiente de autocorrelação primário 37

Figura 4 – Diagrama e funcionamento da arquitetura proposta por Yang et al (2016) 41

Figura 5 - Condições de encontro das bordas de subida e descida do oscilador em anel 42

Figura 6 - Gerador de números aleatórios (TRNG) proposto em [7] 42

Figura 7 - Funcionamento do processador hospedeiro(HP) proposto por Yang et al. (2016) 43

Figura 8 - Relação entre média de colapso, desvio padrão e canais de alta entropia na saída do TRNG. 44

Figura 9 - Entropia da saída do TRNG e média de ciclos até o colapso durante ataque de injeção de ruído 45

Figura 10 - Número de oscilações até o colapso para cenários de ataques de injeção de ruído. 45

Figura 11 - Plataforma de desenvolvimento utilizada para programação do processador 48

Figura 12 - Configuração da ponte de comunicação HPS/FPGA 49

Figura 13 - Diagrama de blocos do sistema de coleta de dados 50

Figura 14 - Diagrama de blocos do sistema implementado na FPGA CycloneIV 51

Figura 15 - Processador hospedeiro (HP) implementado na FPGA Cyclone V 53

Figura 16 - Sinais de clock gerados pelo bloco gerador de clocks 54

Figura 17 Diagrama de blocos do teste de frequência implementado na FPGA 54

Figura 18 - Diagrama de blocos do teste de corridas implementado na FPGA 55

Figura 19 - Cenários de oscilação do RO 56

Figura 20 - Diagrama de blocos do teste de máximo 57

Figura 21 - Diagrama de blocos da entidade contadora de sequências aprovadas 57

Figura 22 - Diagrama de blocos da entidade de detecção de viés 58

Figura 23 - Diagrama de blocos da entidade validador de configurações 59

Figura 24 - Comportamento da saída OUT durante o modo de aprendizagem e o modo de operação..61

Figura 25 - Comportamento do HP implementado durante o modo de aprendizagem 63

Figura 26 - Distribuição de probabilidade do coeficiente de autocorrelação primário medido 64

Figura 27 - Comportamento do sistema de detecção e correção de viés 66

Figura 28 - Comportamento do coeficiente de autocorrelação de duas configurações distintas 68

LISTA DE TABELAS

Tabela 1 - Relação do número de estágios inversores com a frequência de oscilação do RO 59

Tabela 2 - Look-up-table implementada na entidade LUT do processador hospedeiro (HP) 62

Tabela 3 - Resultados dos testes estatísticos de aleatoriedade do NIST para Vdd= 1.2v......................65

Tabela 4 - Resultados dos testes estatísticos de aleatoriedade do NIST para Vdd = 1v após correção de viés 67

Tabela 5 - Tabela comparativa de performance..69

Tabela 6 - LUT salva na memória do processador 78

SUMÁRIO

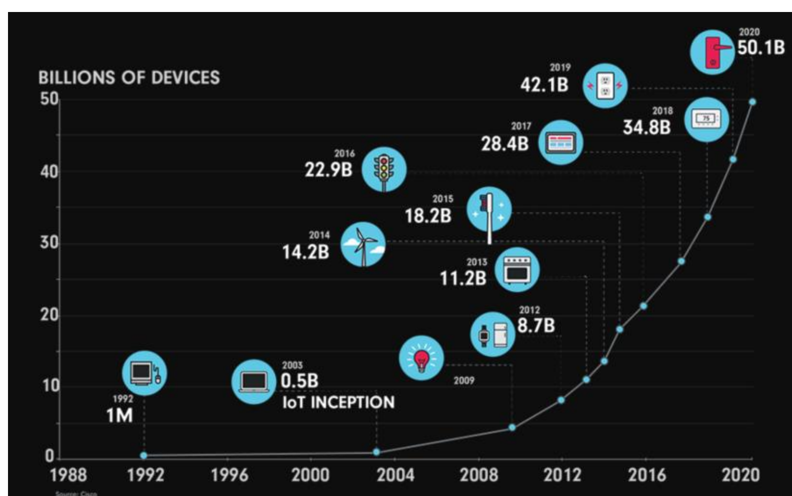
1 Introdução **31**
1.1 Motivação **31**
1.2 Objetivos **32**
2 Revisão bibliográfica **33**
2.1 Conceito: aleatoriedade e gerador de números aleatórios **33**
2.2 Testes de aleatoriedade **33**
2.3 Aplicações de números aleatórios **37**
2.4 História dos geradores de números aleatórios **38**
3 Materiais e Métodos **41**
3.1 Estado da arte **41**
3.2 Objetivo do projeto **46**
3.2 Algoritmo proposto **47**
3.2.1 Modo de aprendizagem 47
3.2.2 Modo de operação 47
3.3 Plataforma de desenvolvimento **48**
3.4 Plataforma de coleta de dados **49**
4 Resultados e discussões **51**
4.1 Implementação do hardware **51**
4.1.1 TRNG 51
4.1.2. Processador hospedeiro (Host Processor) 52
4.2 Configuração do sistema – Estabelecimento das constantes de aprendizagem **59**
4.2.1 Número de estágios (S) 59
4.2.2 Máxima média de colapsos permitida 60
4.2.3 Testes de aleatoriedade NIST 61
4.4 Testes **63**
4.4.1 Funcionamento do algoritmo de aprendizagem em tempo real 63
4.4.2 Eficiência do método de busca guiada 63
4.4.2 Teste de aleatoriedade 64
4.5 Resposta ao enviesamento forçado da saída **65**
4.6 Performance e comparação com trabalhos anteriores **68**
5 Conclusão **71**
REFERÊNCIAS **73**
Apêndice A – Código formato ".m" para cálculo e escrita da LUT em VHDL e LUT completa **77**
Apêndice B – Código em linguagem "C" da rotina de leitura de dados. **81**

1 Introdução

1.1 Motivação

Em meados de 2008/2009 o número de dispositivos conectados à internet ultrapassou a população mundial, cerca de 6.5bi no momento. Foi então que surgiu o conceito de internet das coisas (IoT). Acredita-se que o número de dispositivos seguirá dobrando a cada 5 anos segundo um estudo feito em 2011 [1], chegando a 50 bilhões de dispositivos conectados em 2020, conforme mostrado na figura 1 fornecida pela *Cisco international limited* [2].

Figura 1 - Previsão de crescimento de dispositivos conectados à internet das coisas (IoT).



Fonte: Cisco *virtual network forecast* www.cisco.com/

Porém para seguir o aumento esperado a internet das coisas (IoT) terá de superar diversas barreiras que estão sendo impostas. A de maior destaque é a segurança e privacidade dos usuários, como constatado em um estudo feito pela McKinsey em 2014 [3]. Um exemplo recente ocorreu em 2015, quando um *hacker* tomou controle de uma aeronave da *United Airlines* através do sistema de entretenimento, situado abaixo do banco do passageiro. Conectando seu computador a uma porta USB ele foi capaz de entrar nos sistemas dos motores e configurações de decolagem da aeronave [4].

Para evitar acessos não autorizados a rede, engenheiros do mundo todo tem tentado desenvolver sistemas cada vez mais robustos e capazes de manter visitantes não desejados distante dos dados comunicados entre os dispositivos. Sabe-se hoje que apenas barreiras de segurança programadas nos softwares dos dispositivos, como por exemplo *software firewalls,* sozinhas, não são capazes de garantir transações de dados seguras [5]. A raiz da confiabilidade de um sistema começa pela segurança implementada no hardware, por isso geradores de números genuinamente aleatórios (*True Random number Generators – TRNG)* tem sido cada vez mais estudados. Sua alta entropia, grau de indeterminação [6], os tornam os componentes mais indicados para gerar as chaves de encriptação das comunicações maquina – maquina utilizada em protocolos de comunicação como *oAuth* e *https* .

O aumento de pesquisas na área e a ampla necessidade de sistemas cada vez mais seguros serviram de motivação para o estudo realizado neste trabalho.

1.2 Objetivos

O projeto aqui apresentado foi desenvolvido buscando atingir os seguintes objetivos gerais:

- Compreender o funcionamento e aplicações dos geradores de números aleatórios genuínos, especificamente da arquitetura proposta em [7] baseada no *Jitter* de osciladores em anel.
- Propor um novo método, mais rápido de detecção e correção de viés na saída do TRNGs baseado em um algoritmo de busca e reconfiguração do anel.
- Validar a arquitetura proposta utilizando-se de testes do órgão regulamentador *National institute of standard and technology (NIST)* e comparar os resultados obtidos com o estado da arte estudado.

2 Revisão bibliográfica

2.1 Conceito: aleatoriedade e gerador de números aleatórios

Aleatoriedade é um evento que ocorre de forma independente de qualquer evento ocorrido, com resultado que não apresenta um padrão mesmo quando repetido por um longo período, não podendo assim ser previsto o próximo resultado por maior que seja o tempo de observação [8].

Métodos mecânicos como o lançar de uma moeda não viciada tendem a produzir resultados aleatórios, porém em termos de produção de sequências aleatórias longas a altas frequências não são aplicáveis. Para isso foram desenvolvidos os geradores de números aleatórios, dispositivos capazes de produzir sequências de dígitos aleatoriamente distribuídos a uma alta frequência. Esses dispositivos copiam eventos da natureza conhecidos por possuírem distribuições aleatórias. Suas mais diversas arquiteturas serão discutidas mais a fundo na seção 2.4.

Apesar de o conceito de aleatoriedade definido no primeiro parágrafo parecer claro, a conceito de aleatoriedade absoluta, ou seja, de toda a sequência sob análise ser aleatória, não é definido o que torna a tarefa de rotular um gerador aleatório como bom ou ruim muito difícil. Como exemplo considere a face cara da moeda o valor binário ‘0’ e a face coroa ‘1’, uma série de lançamentos pode produzir a sequência binária ‘010110101’ como poderia também produzir a sequência ‘111111111’. Em ambos os casos o gerador da sequência não mudou, e as chances individuais de cada valor ser zero ou um a cada lançamento são $\frac{1}{2}$, porém analisando-se a segunda sequência como um todo, um padrão é nítido o que pode trazer a dúvida se o lançar da moeda é mesmo aleatório. Porém nota-se que se repetido infinitas vezes o experimento anterior a tendência é que zeros e uns se equalizassem na distribuição, ou seja, por mais que pequenas amostras pudessem apresentar sequências longas de zeros ou uns o evento tende a ser uma variável aleatoriamente distribuída com probabilidade $\frac{1}{2}$.

O exemplo anterior mostra que uma sequência pode ser analisada sob diversos ângulos e apresentar resultados bons e ruins dependendo da amostra que é retirada para análise. Isso mostra que aleatoriedade depende do processo de geração, mas sua análise depende da amostra que será realizada do todo e do critério ao qual ela será testada [8]. A correta escolha das amostras e testes será discutida nesse trabalho na próxima seção.

2.2 Testes de aleatoriedade

Por mais que seja impossível afirmar aleatoriedade absoluta de uma sequência ainda é possível classificá-la, quanto a seu grau de aleatoriedade, baseado em testes estatísticos.

As sequências sob análise devem passar por um extensivo conjunto de testes para serem classificadas quanto a seu grau de aleatoriedade. Mesmo que nunca qualquer conjunto de testes seja completo para atestar aleatoriedade, cada teste independentemente pode rejeitar a hipótese de uma

sequência não ser aleatória sob um determinado critério de avaliação. Por isso um grande conjunto de análises com diferentes critérios permite qualificar o nível de aleatoriedade da sequência reduzindo ao máximo as chances de se atestar uma sequência como aleatória quando a mesma não é. Esse erro em estatística é chamado de falso positivo e o foco de todos os testes propostos é reduzi-lo.

Está seção tem como intuito definir e detalhar alguns dos mais comuns testes de aleatoriedade. São eles: Teste de frequência, teste de corridas, teste de não-superposição de padrões e teste de autocorrelação.

Um teste estatístico tem como propósito testar uma específica hipótese nula (H_o). Seja a hipótese considerada por todos os testes descritos aqui de que a sequência sob avaliação seja aleatória. Cada teste calcula uma variável estatística que é função da sequência sob análise, e a compara com uma distribuição de referência esperada, um valor crítico é predeterminado e serve de comparação para determinar o quão a hipótese se assemelha a referência e se deve ou não ser rejeitada.

A variável estatística sob questão é o P-valor (P-value), que para a hipótese estudada é a chance do gerador produzir sequências menos aleatórias que a sequência sob análise (Força da evidencia contra a hipótese nula), em outras palavras se P-valor = 1 há 100% de chance de o gerador testado produzir um sequência menos aleatória que a testada, logo a sequência sob teste é a mais aleatória possível. O valor crítico α é o grau de significância, e representa as chances de o teste indicar uma sequência como não aleatória quando esta é aleatória, erro de tipo I ou falso negativo [9], A figura 2 apresenta uma distribuição de referência e as condições de rejeição da hipótese.

Figura 2 - Distribuição de referência e valores críticos de rejeição da hipótese nula

Para uma análise fiel é necessário que a amostra a ser testada seja dividida em m sequências de n números, sendo a escolha correta de m e n crucial para a relevância do resultado obtido. No caso em estudo nesse trabalho, a amostra é a saída binaria do TRNG. Um grau de significância $\alpha = 0.01$ indica que se espera que apenas 1 a cada 100 sequências analisadas tenha hipótese rejeitada mesmo que aleatória, logo nesse caso para $\alpha = 0.01$ é prudente que mais do que 100 sequências sejam analisadas ($m > 100$). O mesmo ocorre para o tamanho da sequência n, se a sequência não for grande

o suficiente para traçar a distribuição pode ocorrer um erro ao compará-la com a distribuição de referência, como no exemplo da moeda citado na seção anterior. Logo cada teste possui seu valor mínimo de *n* para garantir confiabilidade do resultado obtido.

O teste de frequência ou teste de frequência *monobit* avalia a sequência binária quanto a proporção de zeros e uns, uma sequência aleatória tende a apresentar proporção ideal de $\frac{1}{2}$. A distribuição de referência para esse teste é uma distribuição meia-normal. Considere a sequência binaria $\varepsilon = \varepsilon_1, \varepsilon_2, \varepsilon_3 \ldots \varepsilon_n$, seja sua soma absoluta $S_n = \sum_{i=1}^{n} 2\varepsilon_i - 1$, o valor P é dado por:

$$P_{value} = erfc(\frac{|S_n|}{\sqrt{2n}}) \tag{1}$$

Onde *erfc* é a complementar da função erro. Para garantir confiabilidade do resultado este teste necessita que *n* seja no mínimo 100 [9].

O teste de corridas pode ser considerado um teste complementar ao teste de frequência uma vez que busca analisar como os bits estão distribuídos na sequência verificando se a oscilação entre zeros e uns é rápida ou lenta. Seu algoritmo contabiliza o número de sequências ininterruptas do mesmo digito. Essas sequências são chamadas de corridas. Considere a sequência binaria $\varepsilon = \varepsilon_1, \varepsilon_2, \varepsilon_3 \ldots \varepsilon_n$. Seja V_{obs} o número de corridas observadas e π a proporção de dígitos '1' na sequência ε dada por $\pi = \frac{\sum_{i=1}^{n} \varepsilon}{n}$.

O P-valor é dado por:

$$P_{value} = erfc(\frac{|V_{obs} - 2n\pi(1-\pi)|}{2\sqrt{2n\pi(1-\pi)}}) \tag{2}$$

Este teste tem como referência uma distribuição chi-quadrada (χ^2), uma distribuição da família *gama* [9] [6]. Assim como o teste de frequência o tamanho recomendado de sequências a ser utilizado é *n* superior a 100 bits.

O teste de não-sobreposição de padrões (*Non overlaping template*) tem como intuito vasculhar a sequência sob teste e procurar por padrões predefinidos. A sequência original é subdividida em N blocos independentes de M bits. Uma janela B de tamanho m =9 ou $m = 10$ bits, desliza bit a bit em busca do padrão sob teste. Para uma janela de 9bits 148 padrões são testados e para 10bits até 248 padrões [9]. Se o padrão procurado é encontrado incrementa-se uma variável W e a janela se desloca m bits para frente e continua o processo até que a sequência de M bits seja avaliada por inteiro. O processo descrito se repete para todas as N sequências calculando-se um W para cada uma delas.

Sob hipótese de aleatoriedade calcula-se o valor médio e a variância da distribuição referência:

$$\mu = \frac{M - m + 1}{2^m} \; e \; \sigma^2 = M(\frac{1}{2^m} - \frac{2m - 1}{2^{2m}}) \tag{3}$$

Calcula-se a soma dos erros quadráticos médios de W para o valor médio μ e divide o resultado pela variância de referência.

$$\chi^2 = \sum_{j=1}^{N} \frac{(W_j - \mu)^2}{\sigma^2} \tag{4}$$

Essa variável estima o quão bem o valor esperado de sobreposições bate com o valor encontrado.

O P-valor é então calculado para cada padrão testado:

$$P_{value} = igamc(\frac{N}{2}, \frac{\chi^2}{2}) \tag{5}$$

Onde $igamc$ é a função gama incompleta. Para este teste o número de bits analisados deve ser tal que:

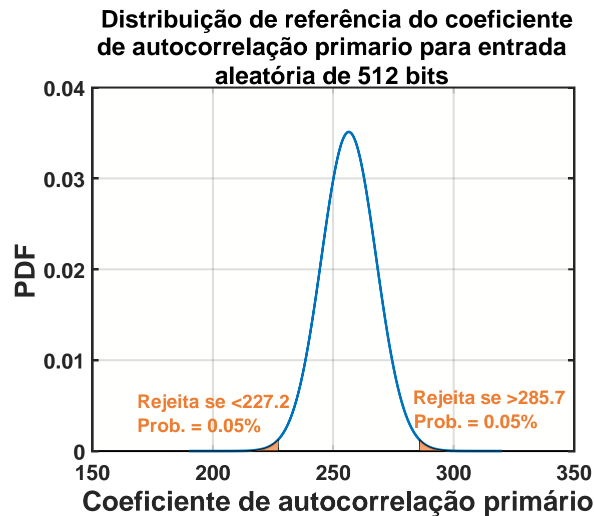
$$\begin{cases} N \leq 100 \\ M > 0.01n \\ N = \dfrac{n}{M} \end{cases}$$

O teste de autocorrelação produz um resultado nomeado coeficiente de autocorrelação que define o quanto os bits da mesma sequência estão relacionados. A relação entre bits adjacentes é dada pelo coeficiente de autocorrelação primário. Este teste foi descrito em [10] como sendo um dos mais indicados para identificar viés na saída de TRNGs e, devido a sua simplicidade, será de grande ajuda no trabalho aqui desenvolvido. O coeficiente de autocorrelação primário é descrito pela seguinte equação:

$$C_i = \sum_{k=1}^{n-i} \varepsilon_k \oplus \varepsilon_{k+i} \tag{6}$$

Onde ε são bits da sequência analisada. A variável i indica a distância entre quais bits da sequência se deseja calcular a correlação, se i=1 calcula-se dos bits adjacentes e assim por diante. Sequências maiores que 512 bits mostraram melhores resultados para detecção do viés [10] após o cálculo o coeficiente de C_1 para 128.000 sequências aleatórias de 512bits. Pode-se ver na figura 3 a distribuição de referência obtida através da análise de 10mil sequências de 52 bits. Esta será usada como referência ao longo deste trabalho.

Figura 3 - Distribuição de referência do coeficiente de autocorrelação primário



Distribuição obtida após aplicação do teste de autocorrelação primário 10mil sequências aleatórias distintas de 512bits.

Fonte: Leonardo Bosco Carreira (2019)

A figura 3 apresenta C_1 para uma sequência aleatória de 512bits deve estar entre 227 e 285 para um $\alpha = 0.01$. Qualquer alteração que leve a sequência a não ser mais aleatória, chamada de sequência enviesada ou com viés, irá alterar o valor para fora dessa margem.

2.3 Aplicações de números aleatórios

Os números aleatórios estão presentes constantemente na vida atual. Por mais que de uso implícito sem eles não seria possível executar diversas tarefas do cotidiano. Alguns exemplos de aplicações são: Criptografia, simulações computacionais, a amostragem de pesquisas, análises numéricas, programação e física experimental [11].

Na criptografia, os números aleatórios são utilizados para criar as sementes de encriptação dos sistemas de comunicação. Apenas os detentores da semente são capazes de extrair a mensagem original. Todo envio de dados hoje é criptografado: E-mails, mensagens de texto, *Tags* de cancelas e comunicações máquina-máquina no geral.

Nas simulações como métodos de Monte Carlo, a utilização de números aleatórios é imprescindível durante a amostragem aleatória garantindo relevância da solução encontrada.

Outra aplicação conhecida é no uso de algoritmos de aprendizado de máquina (machine learning – ML), onde números aleatórios são utilizados para amostragem da sequência de testes garantindo assim uma conversão mais rápida do treinamento[12].

Em experimentos científicos por exemplo a estimativa do resultado esperado é baseada numa seleção aleatória da amostra obtida Assim é possível se traçar uma distribuição que reduza o erro esperado devido a aleatoriedades do processo.

Em telecomunicações números aleatórios são utilizados em diversas frentes, por exemplo na compressão dos sinais, mensagens amostradas aleatoriamente apresentam maior eficiência energética[13]. Outras aplicações são na criptografia de mensagem ou autenticação das entidades comunicantes [14].

2.4 História dos geradores de números aleatórios

O estudo dos números aleatórios ganhou grande incentivo com a necessidade de pesquisas probabilísticas e de amostragem aleatória em experimentos científicos por volta dos anos 30. Até o advento dos computadores, números aleatórios eram gerados em sistemas mecânicos manualmente e transcritos em tabelas conhecidas como tabelas de amostragem aleatória publicadas em tabelas da época [15].

Porém, os métodos de obtenção eram incertos e não havia uma forma documentada de garantir sua eficácia, até que em 1938 Kendall and Babington-Smith propuseram um conjunto de quatro testes para classificar as tabelas publicadas [16]. No mesmo ano Kermack and Kendrick publicaram outros dois testes estatísticos para testar o comportamento periódico das sequências [17] .Alguns destes testes, como os testes de frequência de Kendall and Babington e os testes de corridas e lacunas de Kermack and Kendrick, ainda perpetuam até hoje em softwares destinados a testar RNGs, como o teste do *National institute of standard and technology (NIST)* para classificar geradores de números aleatórios principalmente para aplicações em criptografia [9].

Foi por volta dos anos 40, seguindo o surgimento dos computadores, que geradores de números aleatórios passaram a ser automatizados e desenvolvidos em *hardware*. O primeiro método que se tem conhecimento foi desenvolvido pela empresa RAND em 1949. O dispositivo era baseado em um gerador de pulsos emitidos aleatoriamente. Os pulsos eram contados por um contador de 5 dígitos (módulo 32). Os resultados gerados pelo dispositivo tiveram de ser posteriormente tratados antes de impressos em uma tabela com 1 milhão de números aprovados nos testes propostos em [15]. A tabela virou livro e foi um marco da época para cientistas e estatísticos. Em 1957, uma máquina foi desenvolvida para uso na loteria britânica, nomeada ERNIE (*electronic random number indicator equipment).* O método utilizado na ERNIE se aproximou muito a alguns métodos utilizados em casos recentes. A ERNIE contava o número de partículas dos gases ionizados de válvulas elétricas que colidiam com uma chapa metálica instalada dentro das mesmas. As colisões geravam excitações elétricas que eram transmitidas a um contador [18].

Os eventos do século passado trouxeram uma nova visão de aleatoriedade. Provou-se possível o uso de variáveis imprevisíveis de fácil obtenção para gerar números aleatórios. Essas variáveis eram irreplicáveis, mesmo que garantidas as mesmas condições de operação, sendo assim em sua grande maioria consideradas fontes de alta entropia. Um exemplo é o ruído térmico, uma das primeiras fontes a serem quantificada e utilizada como gerador de números aleatórios com aplicações em criptografia.

Em 2000 os pesquisadores C.S. Petrie e J.A. Connely propuseram a amplificação do ruído térmico de resistores em uma nova arquitetura. O ruído era amplificado por um amplificador de alta impedância de saída com larga banda de amplificação, o sinal amplificado era então comparado através de uma porta XOR com a saída de um oscilador controlado por corrente alimentado pelo mesmo ruído amplificado [19].

Esse método inovou os métodos de amplificação direta de ruído propostos na década de 90, pois se provou muito mais robusto a influências externas não necessitando de proteção contra interferências eletromagnéticas de fontes não aleatórias. Porém o sistema ainda precisava de uma precisa calibração do amplificador e do conversor A/D a fim de evitar qualquer viés na saída.

Em 2004 a ERNIE foi reconstruída em sua quarta versão, utilizando o ruído térmico de transistores, tendo seu uso expandido da loteria britânica para a casa da moeda e trabalhos no ramo da economia [18]. Em 2008 M. Matsumoto [20] propôs à amplificação e quantização do ruído emitido por SiN Mosfets para aplicações em cartões inteligentes (*SIM cards).* Comparado a outros métodos de amplificação direta de ruído este se mostrou mais estável a variações elevadas de temperatura e com maior potencial de integração, permitindo reduzir o tamanho dos circuitos integrados da época. A metaestabilidade foi também fruto de grandes estudos nos anos 2000, pois assim como a amplificação direta de ruído, possui alta entropia e possibilidade de gerar sequências aleatórias em altas frequências, aliado com a vantagem de se tratar de circuitos de baixo consumo de potência e maior nível de integração por se tratar de um circuito inteiramente digital. Entretanto sistemas metaestáveis tem baixa estabilidade e necessitam de refinada calibração para garantir amostragem no estágio metaestável [21].

A natureza continuou inspirando as buscas por novos métodos de se obter sequências aleatórias. O tempo levado para um evento incerto ocorrer foi o estudo de N. Liu et. Al, uma fina camada de oxido alimentada à tensão reversa acima da tensão limite de ruptura tende a romper-se em um prazo não equacionável devido as variações no processo de produção de cada componente. A quantificação desse tempo levou a uma nova arquitetura de TRNG em 2011 [22]. Porém, mesmo que aleatório, o tempo de rompimento do oxido não variava em grande escala o que não permitia produzir palavras aleatórias muito grandes, reduzindo assim sua aplicação em criptografia, também a alta tensão de rompimento tornava o custo de potência muito elevado para uso em sistemas IoT alimentados por baterias.

Geradores de números aleatórios baseados no *Jitter* de osciladores em anéis são assuntos de grandes estudos na atual década devido ao seu alto nível de integração, simplicidade e baixo consumo. Em 2003 os pesquisadores Marco Bucci et. Al [23], propuseram uma topologia com dois osciladores, um oscilando a 10MHz era usado como *Clock* de amostragem de um oscilador de 1GHz. A ideia por trás desse método é que devido ao *Jitter* acumulador durante a oscilação há um grau de incerteza do dado no momento da amostragem, porém a acumulação de *Jitter* em um único oscilador é limitada e

não adequada para uso como fonte de alta entropia o que tornou necessário o uso de pós processamento dos dados para remover sequências longas de zeros e uns obtidas. Métodos baseados no *Jitter* de osciladores foram sendo aperfeiçoados, medindo a diferença de frequência de dois osciladores em paralelo (*beat-frequency)* [24] [25],osciladores caóticos [26], ou mais recentemente o uso do tempo de colapso entre oscilações do primeiro e terceiro harmônico do mesmo oscilador [27]. Todas formas apresentaram alta entropia, alta capacidade de integração e baixo consumo de potência. Porém não foi comprovada a robustez de nenhum dos sistemas contra variações de processo, tensão de alimentação e temperatura (PVT). O trabalho proposto por Yang et Al. (2016). em [7] foi um dos exemplos que visou corrigir esse problema apresentando um método mais robusto. A próxima seção apresenta a fundo o estado da arte base para esse projeto e as melhorias propostas.

3 Materiais e Métodos

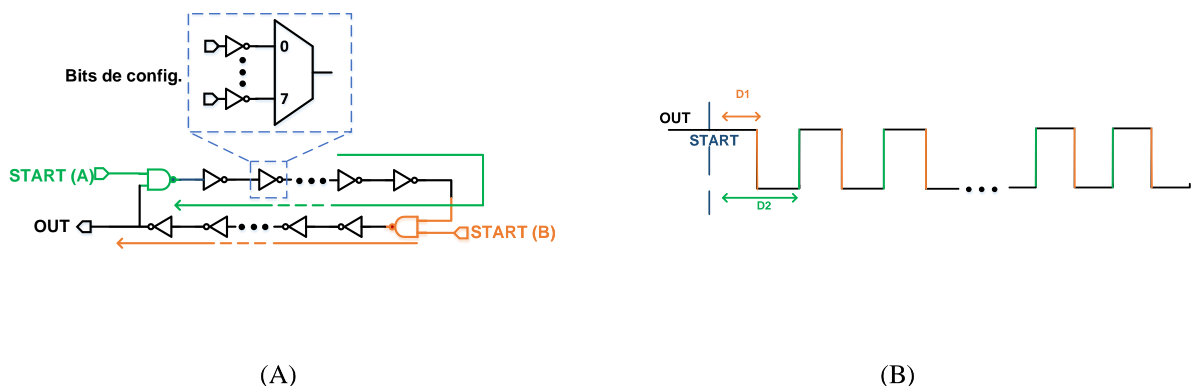
3.1 Estado da arte

Antes da proposta de Yang et Al. (2016) geradores de números aleatórios digitais baseados no *jitter* de osciladores apresentavam dois problemas cruciais, eram instáveis a variações de temperatura e tensão, e tinham a necessidade de métodos adicionais para evitar que o elevado tempo de acúmulo do *jitter* não criasse uma propensão a mais zeros ou uns na saída. Alguns métodos para reduzir o viés na saída foram o pós processamento ou a utilização do *jitter* não apenas da frequência central do oscilador mas também de harmônicos da frequência de operação, ou métodos de interferência de frequências ligeiramente diferentes (*beat frequency)*, todas as soluções comparada à propostas em [7], aumentavam muito os gasto de área e potência do sistema.

Para reduzir os efeitos das variações de processo, tensão e temperatura (PVT) Yang et al. (2016) propôs um circuito fechado com ajuste automático da fonte de entropia. O anel do oscilador é formado por múltiplos caminhos, com cada estágio inversor composto por 8 inversores selecionados através de um multiplexador. A a variação no processo de fabricação faz com que cada caminho oscile a uma frequência ligeiramente distinta por mais que os inversores sejam identicamente projetados. A figura 5 exemplifica o sistema de multi-caminhos. O modelo proposto exclui também a necessidade de detectores de fase e frequência, utilizados em [27] para detecção do colapso, fim da oscilação, dos harmônicos, uma vez que seu método garante fácil detecção do colapso através de um contador binário.

A fonte de entropia proposta é um oscilador em anel com número par de portas inversoras, onde dois sinais são injetados simultaneamente em estágios opostos através de duas portas NAND. O número par de estágios faz com que cada sinal injetado percorra o anel de forma independente e atinja a saída sempre na mesma borda, subida ou descida, nomeando os sinais como A e B. A figura 4 (A) mostra como cada um atinge a saída do oscilador em anel (RO).

Figura 4 – Diagrama e funcionamento da arquitetura proposta por Yang et al (2016) .



(A) (B)

(A) RO proposto em [7] com ênfase no estágio multicaminho. (B) Saída do oscilador onde os sinais de excitação A e B regem as bordas de subida e descida respectivamente.

Uma vez que independentes, o tempo de propagação através do anel dos sinais A e B são ligeiramente distintos sendo descrito pela soma das seguintes parcelas: o atraso ideal, $Ideal_Delay$, a variação de atraso devido as variações intrínsecas do processo de fabricação de cada porta, $\Delta Delay$, a variação aleatória devido ao $Jitter$; e a constante D que representa o tempo levado para a primeira borda atingir a saída. Todas variáveis se acumulam durante as N voltas nos S estágios do anel. Baseado nisso o seguinte modelo do tempo de descida e subida foi proposto em [7]:

$$T_{fall,N} = D_1 + \sum_{n=1}^{N} \sum_{i=1}^{S} \left(ideal_Delay_{i,B} + \Delta Delay_{i,B} + Jitter_{n,i,B}\right) \quad (7)$$

$$T_{rise,N} = D_2 + \sum_{n=1}^{N} \sum_{i=1}^{S} \left(ideal_Delay_{i,A} + \Delta Delay_{i,A} + Jitter_{n,i,A}\right) \quad (8)$$

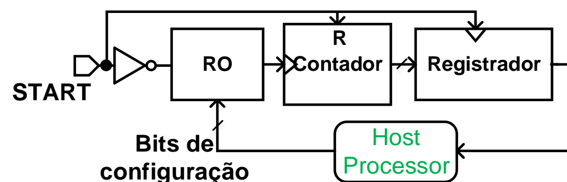
A oscilação do anel cessa assim que a borda de descida encontra a borda de subida, denominado colapso. Sendo possível dois casos: A borda de subida é mais rápida que a de descida ou o caso contrário. A figura 5 exemplifica o colapso das duas bordas em ambos os casos.

Figura 5 - Condições de encontro das bordas de subida e descida do oscilador em anel

(A) Condição de colapso para borda de subida mais rápida que borda de descida. (B) Condição de colapso para borda de descida mais rápida que de subida.

A fim de contabilizar o número de oscilações até o colapso, $Yang$ et. Al propôs o uso de um contador e um banco de registradores para contar e armazenar o número de oscilações respectivamente, como mostrado na figura 6.

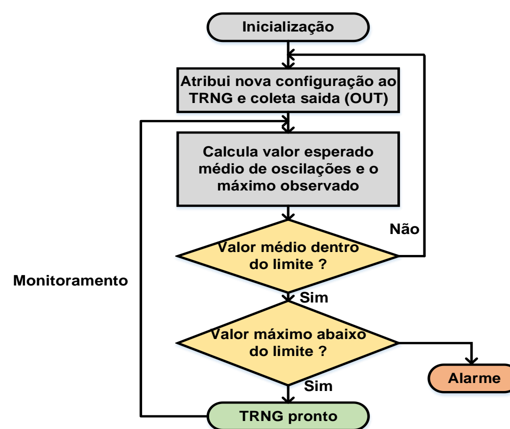
Figura 6 - Gerador de números aleatórios (TRNG) proposto em [7]



Antes de iniciar o processo um processador hospedeiro (host processor – HP) define qual o caminho da oscilação no anel, selecionando o conjunto de inversores a serem utilizados através dos bits de configuração. Os bits de configuração são na verdade a saída de um *Linear feedback shift register* (LFSR). Definido o caminho o sinal *START* é injetado no anel, sua borda de descida da início as oscilações do anel, o contador de modulo 256 (8-bits) conta quantas vezes o anel oscila até o colapso e o valor encontrado é salvo em um banco de registradores na subida do sinal *START* logo

antes do mesmo zerar o contador. O ciclo se repete de 500 a 5000 vezes para os mesmos bits de configuração para que o processador consiga calcular o a média de oscilações que levam ao colapso para aquela configuração. Nomearemos esse valor de μ_{osc} . Uma vez calculado, μ_{osc} é comparado a um limite inferior e superior *pré*-definido (a escolha do *range* é discutida mais a frente nessa seção), se dentro do valor esperado os bits de configuração são mantidos. O ciclo se repete durante todo tempo de operação do TRNG. Caso o valor médio ultrapasse os limites o LFSR no processador recebe um sinal de clock e altera o caminho de inversores até que o sistema volte a operação desejada. O fluxograma apresentado na figura 7 exemplifica o funcionamento do sistema:

Figura 7 - Funcionamento do processador hospedeiro(HP) proposto por Yang et al. (2016)



A escolha dos limites é baseada na função de distribuição de probabilidade (pdf) dos colapsos. Para inferir tal distribuição estatística fez-se uso das equações (1) e (2). O sistema para de oscilar quando $T_{fall,N} = T_{rise,N}$, logo a equação que rege o colapso pode ser obtida igualando-se (1) e (2). Uma vez que os estágios possuem o mesmo tempo de propagação ideal ($Ideal_Delay_{i,B} = Ideal_Delay_{i,A}$) e que para um mesmo dispositivo as variações nas portas devido ao processo de fabricação são imutáveis $\sum_{n=1}^{N}\sum_{i=1}^{S}\Delta Delay_{i} = N \times \sum_{i=1}^{S}\Delta Delay$, a igualdade pode ser simplificada como:

$$D_2 - D_1 = \mathrm{N} \times \sum_{i=1}^{S}\left(\Delta Delay_{i,B} - \Delta Delay_{i,A}\right) + \sum_{n=1}^{N}\sum_{i=1}^{S}\left(Jitter_{n,i,B} - Jitter_{n,i,B}\right) \quad (9)$$

A variação do processo de fabricação, $\Delta Delay$, segue uma distribuição normal de média zero acumulada igualmente durante os N ciclos, podendo ser expressa por $N \times \mathcal{N}(0, \sigma_{processo}^2)$. O *Jitter* majoritariamente é gerado devido ao ruído térmico [28] e também segue uma distribuição normal de média zero independentemente acumulada durante cada estágio e pelos N ciclos $\sum_{n=1}^{N}\sum_{i=1}^{S}\mathcal{N}\left(0, \sigma_{jitter}^2\right)$. A subtração/soma de duas variáveis independente e identicamente distribuídas (iid) é dada pela distribuição com valor médio igual a subtração/soma dos valores médios e a soma das variâncias [6]: $\mathrm{N} \times \sum_{i=1}^{S}\left(\Delta Delay_{i,B} - \Delta Delay_{i,A}\right) = \mathrm{N} \times \mathcal{N}\ (0, 2S\sigma_{processo}^2)$ e $\sum_{n=1}^{N}\sum_{i=1}^{S}Jitter_{n,i,B} - Jitter_{n,i,A} = \sum_{n=1}^{N}\mathcal{N}(0{,}2S\ \sigma_{jitter}^2)$. A expressão final pode ser escrita como:

$$D_2 - D_1 = N \times (\mathcal{N}(0, 2S\sigma^2_{process}) + \sum\nolimits_{j=1}^{N} \mathcal{N}(0, 2S\sigma^2_{jitter}) \quad (10)$$

Na equação (10) o primeiro termo a direita da igualdade é linear com o número de ciclos e o segundo termo é um passeio aleatório em função de N. Como um movimento browniano pode ser visto como o caso limite de um passeio aleatório [6] a equação (10) pode ser comparada a um processo de Wiener com desvio ν e variância σ^2 : $X_t = \nu t + \sigma W_t$, onde W_t é um movimento browniano padrão.

A solução de (10) é então a primeira passagem do processo de *Wiener* pelo ponto $\alpha = D_2 - D_1$. A primeira passagem é distribuída de acordo com uma gaussiana inversa com os seguintes parâmetros:

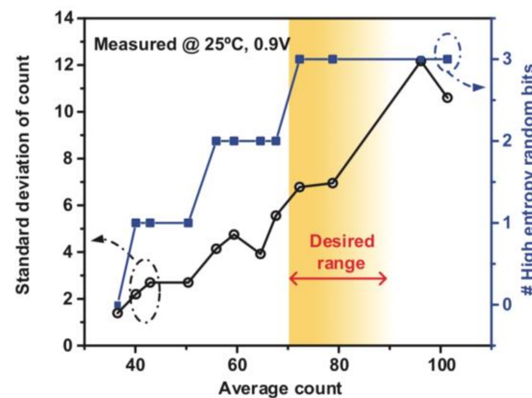
$$IG\left(\frac{\alpha}{\nu}, \frac{\alpha^2}{\sigma^2}\right) \therefore$$

$$\mu = \frac{|(D_2 - D_1)|}{|\sum_{i=1}^{S}(\mathcal{N}(0, 2\sigma^2_{process})|} \quad (11)$$

$$\lambda = \frac{(D_2 - D_1)}{2S\sigma^2_{jitter}} \quad (12)$$

Através das equações (11) e (10) Yang et al. (2016) pode modelar o sistema e definir os limites para gerar a quantidade de canais aleatórios desejados na saída do TRNG. Segundo seus estudos um colapso médio (μ_{osc}) de 80 a 90 oscilações garantiria que os últimos 3LSBs da saída se comportassem de forma aleatória (com alta entropia), como mostrado na figura 8 .

Figura 8 - Relação entre média de colapso, desvio padrão e canais de alta entropia na saída do TRNG.



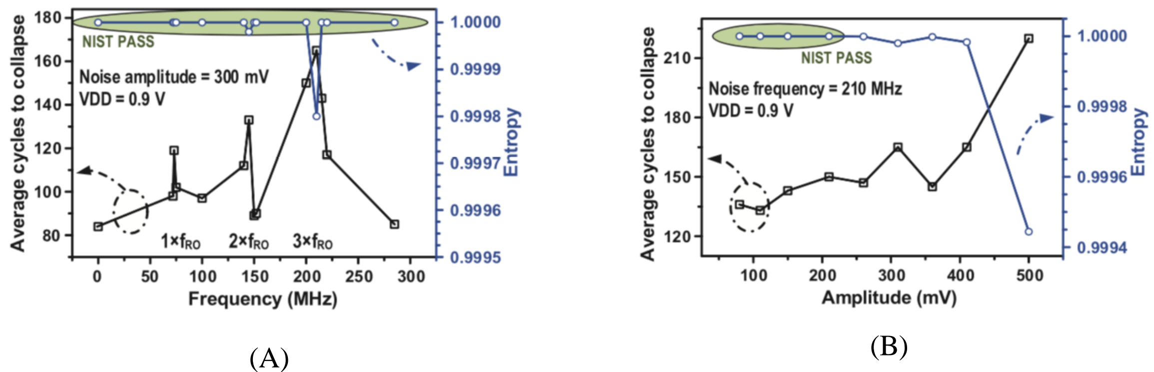
Média de ciclos até colapso **X** desvio padrão doe ciclos até colapso **X** Número de canais com alta entropia na saída do TRNG.

Fonte: An All-Digital Edge Racing True Random Number Generator Robust Against PVT Variations, *K.Yang et al. (2016)*

Nos seus estudos também demonstrou o comportamento da saída sob ataques não invasivos que visam enviesar a saída do TRNG, como é o caso da injeção de ruído na fonte de alimentação do oscilador. Para isso foi adicionada a alimentação do circuito integrado (CI) um sinal AC, e medida a

entropia da saída conforme a frequência e amplitude desse sinal eram alteradas. As figuras 9 (A) e 9 (B) mostram os resultados obtidos:

Figura 9 - Entropia da saída do TRNG e média de ciclos até o colapso durante ataque de injeção de ruído



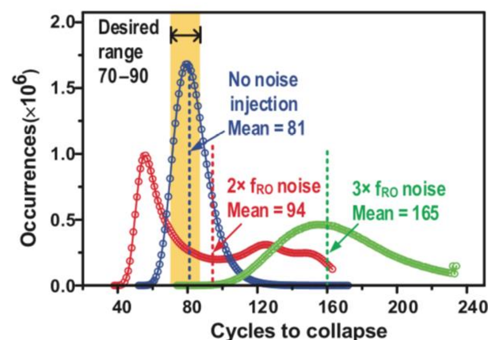
Ruído injetado com (A) Tensão de pico fixada em 300mVe frequência variada de 0 a 300MHz . (B) Frequência fixada em 210MHz e tensão de pico variada de 0 a 500mV

Fonte: *An All-Digital Edge Racing True Random Number Generator Robust Against PVT Variations*, K.Yang et al. (2016)

Notou-se que a injeção de ruídos em frequências harmônicas a de oscilação do anel causavam o efeito chamado de *injection locking* [29], onde o oscilador passa a ressoar e oscilar de maneira quase idêntica ao sinal injetado fazendo com que a saída perca significativamente a aleatoriedade e consequentemente seja identificada pelo intruso.

A relação do ataque de injeção de ruído com a arquitetura proposta em [7] é direta. Como descrito em [30] o *Jitter* da oscilação tende a diminuir substancialmente com a ressonância o que elevaria o tempo médio de colapso do anel a aumentar até o ponto onde a oscilação seria truncada no valor máximo (onde não ocorre colapso) diminuindo consideravelmente a aleatoriedade da saída até que esta seja uma sequência inteira de ‘1’s ou ‘0’s . A figura 10 mostra o efeito da injeção de ruído com frequência igual ao segundo e terceiro harmônico, onde já se nota próximo ao valor 240 no eixo x o truncamento no valor máximo.

Figura 10 - Número de oscilações até o colapso para cenários de ataques de injeção de ruído.



Ruído injetado em frequências múltiplas a frequência do oscilador em anel (f_{RO}) para a mesma configuração

Fonte: *An All-Digital Edge Racing True Random Number Generator Robust Against PVT Variations*, K.Yang et al. (2016)

A figura 10 demonstra como o modelo proposto consegue detectar facilmente o ataque, uma vez que o desvio da média de colapsos pode ser facilmente detectado pelo algoritmo do processador.

Para corrigir o viés na saída causada pelo ruído injetado, o algoritmo da figura 7 propõe que os bits de configuração sejam trocados até que o anel oscile em outra frequência, devido as ligeiras diferenças de propagação dos caminhos como citado, dessa forma saindo do estado ressonante.

3.2 Objetivo do projeto

O algoritmo proposto em [7] leva um grande tempo para detectar e corrigir o sinal de saída se considerado que as chances de encontrar uma configuração que leve a caminhos que colapsem dentro da faixa desejada é muito baixa. No próprio artigo base para esses estudos é citado que aproximadamente apenas 5% das configurações levam a colapsos na faixa de 70-90 oscilações. Dada a probabilidade de ocorrência e considerando que cada evento é independente pode-se calcular a probabilidade de as configurações testadas colapsarem dentro do limite desejado na enésima tentativa:

$$P(n) = (1 - p)^{n-1}p \tag{13}$$

Onde p é a probabilidade de o evento ocorrer (5%), n é a enésima configuração testada e *P(n)* é a probabilidade de o enésimo teste colapsar com média de distribuição dentro do limite. A esperança (E) da função descrita pela equação (13) pode ser descrita como $\frac{1}{p}$. Logo, espera-se que em média na tentativa $n = 20$ o TRNG encontre uma configuração satisfatória.

Levando em conta que o algoritmo pode levar até 5000 colapsos para testar cada configuração o número médio de tentativas de recuperação do sistema é:

$$T_{resposta} = 5000 \times 20 = 10.000\ colapsos$$

No pior dos casos evidenciado em [7] o sistema levou 315 configurações para retornar a operação desejada o que custaria 1.575.00 $colapsos$.

Outro ponto negativo da arquitetura proposta é que a detecção do viés na saída é baseada no número de oscilação até o colapso, tornando o método de detecção dependente do uso da mesma arquitetura de TRNG.

Vistos os pontos discutidos nesse tópico os objetivos desse projeto são:

- Redesenhar a arquitetura proposta por Yang et al. (2016) em uma plataforma reprogramável (FPGA).

- Propor um processador com menor tempo de resposta e mais adaptável a outras arquiteturas de TRNGs e implementá-lo em uma FPGA.

3.2 Algoritmo proposto

A fim de alcançar os objetivos propostos, este trabalho traz a implementação de um algoritmo inovador de rápida resposta e facilmente adaptável a diversas arquiteturas de TRNGs. O algoritmo proposto se baseia em um processo de aprendizagem a partir dos mais importantes testes estatísticos para avaliação da aleatoriedade de sequências binarias. O algoritmo pode ser dividido em 2 macro etapas:

- Modo de aprendizagem
- Modo de operação

3.2.1 Modo de aprendizagem

Durante o modo de aprendizagem, diversas configurações são testadas. Para cada caminho as saídas do TRNG passam pelos testes de corridas (*runs)* e o teste de frequência monobit (*Monobit frequency*) apresentados na seção 2.2 128 sequências de 256bits são testadas para cada configuração seguindo as especificações do órgão regulamentador.

Assim como no trabalho de Yang et al. (2016) foi previamente definido o número de canais aleatórios desejados na saída do gerador, no caso os 3LSBs. Para isso as três saídas do TRNG, OUT [2], OUT[1] e OUT[0] são testadas simultaneamente no algoritmo proposto a fim de garantir aleatoriedade de todas. O processo também calcula o valor médio de oscilações até colapso durante todas as 128 sequências testadas para verificação do máximo, o motivo de aplicar esse teste será abordado mais a fundo no decorrer do trabalho.

Uma vez que as três saídas sob teste são aprovadas em ambos os testes e o valor médio de oscilações até o colapso é menor que o valor máximo permitido, definido nas próximas seções, a configuração sob avaliação é salva em uma memória interna do processador proposto. O processo se repete até que a memória esteja completa fazendo com que o sistema passe ao modo de operação. Esse processo foi denominado pelo autor como “Busca aleatória”, uma vez que pretende buscar de maneira desordenada/aleatória dentro do conjunto finito, porém muito extenso, por configurações que atendam a requisitos desejados.

3.2.2 Modo de operação

Com a memória preenchida com diversos caminhos já testados o TRNG passa ao modo de operação, este é o modo onde o TRNG está pronto para uso. O oscilador em anel (RO) é alimentado com a primeira configuração de caminhos salva.

Nessa etapa usa-se o teste de autocorrelação primária entre os bits de saída, descrito em na seção 2.2, para analisar o resultado de saída em tempo real. Por se tratar de um teste estatístico, cuja

distribuição é gaussiana, o teste de correlação pode porventura apresentar resultados de coeficientes fora da margem desejada, mesmo para uma configuração que não esteja produzindo uma saída enviesada. A fim de evitar a eventualidade de um falso negativo, a configuração observada apenas será rotulada enviesada caso reprove três vezes consecutivas no teste.

O teste é aplicado na sequência composta pela concatenação das saídas TRNG_OUT(2) ... TRNG OUT(0), reduzindo assim a latência de resposta.

Caso seja detectado viés na saída a próxima configuração gravada na memória é testada. O processo se repete até que a o coeficiente de autocorrelação esteja novamente dentro dos limites aceitáveis. Esse processo foi nomeado como “Busca guiada” uma vez que a busca de configurações ocorre dentro de um universo muito mais reduzido e previamente testado e é a grande contribuição para redução do tempo de resposta do sistema.

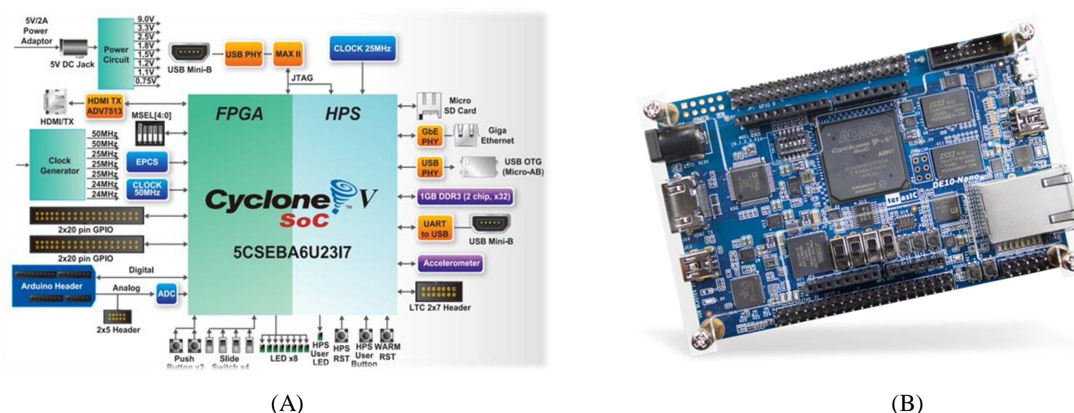
3.3 Plataforma de desenvolvimento

O processador com o algoritmo proposto e o TRNG proposto por Yang et al. (2016) foram implementados em uma FPGA, a escolha da plataforma se deve a facilidade de alteração do projeto durante o desenvolvimento.

Como o enfoque desse projeto é o processador hospedeiro (HP), preferiu-se separar o TRNG e o HP em duas plataformas distintas. O TRNG foi desenvolvido em uma plataforma equipada com a FPGA EP4CE6E22C8N da família Cyclone IV da Altera, já o processador foi implementado no kit de desenvolvimento DE-Nano 10 da TerasIC que conta com uma FPGA Cyclone V 5CSEBA6U23I7 e um *hard processor system (*HPS) baseado em uma arquitetura ARM.

O kit foi escolhido devido a interface HPS/FPGA que facilita a coleta dos dados gravados na FPGA através do Linux rodando no ARM. A figura 11, extraída do site do fabricante [31], apresenta o digrama de blocos do kit e a placa de desenvolvimento.

Figura 11 - Plataforma de desenvolvimento utilizada para programação do processador



(A) (B)

(A) Diagrama de blocos da FPGA Cyclone V instalada no kit de desenvolvimento da TerasIC utilizado nesse projeto (B) Kit de desenvolvimento da TerasIC utilizado nesse projeto

Fonte: www.terasic.com.tw (2019)

A programação da FPGA foi feita na plataforma de desenvolvimento Quartus Prime Lite, gratuitamente fornecida pela Altera em seu site [32] enquanto a programação do ARM foi realizada em editor de código comum como *NotePad++* [33] ou *Sublime Text* [34].

3.4 Plataforma de coleta de dados

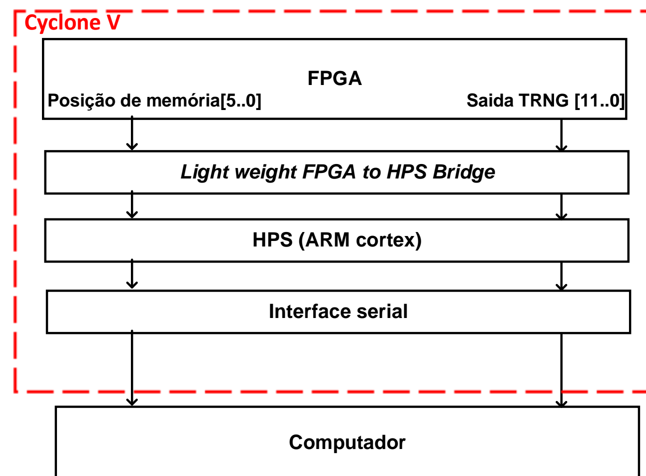
Os dados gerados pelo sistema foram coletados da seguinte maneira: Os GPIOs (*General purpose input/outputs*) das placas foram conectados através de *Jumpers* de forma que as saídas do TRNG fossem enviadas para o processador. No kit de desenvolvimento foi configurada a ponte de interface do HPS com os periféricos da FPGA como mostrado na Figura 12. No processador ARM foi instalado a versão do Linux indicada pelo fabricante [31] onde um programa, escrito em linguagem C, rodava uma rotina para leitura dos GPIOs da FPGA conectados a ponte *Light weight FPGA to HPS e* escrita dos valores obtidos em um arquivo texto. O código pode ser visto no apêndice B.

Os resultados obtidos foram pós processados utilizando o *software Matlab* da Mathworks [35]. As figuras 12 e 13 mostram a configuração da ponte e o diagrama de blocos da interface FPGA-Usuário:

Figura 12 - Configuração da ponte de comunicação HPS/FPGA

Conne...	Name	Description	Export	Clock	Base	End
	clk_0	Clock Source				
	clk_in	Clock Input	**clk**	*export*		
	clk_in_reset	Reset Input	*Double-click to export*			
	clk	Clock Output	*Double-click to export*	clk_0		
	clk_reset	Reset Output	*Double-click to export*			
	hps_0	Arria V/Cyclone V ...				
	memory	Conduit	**memory**			
	hps_io	Conduit	**hps_io**			
	h2f_reset	Reset Output	*Double-click to export*			
	h2f_lw_axi_clock	Clock Input	*Double-click to export*	**clk_0**		
	h2f_lw_axi_master	AXI Master	*Double-click to export*	[h2f_l...		
	read_trngout	PIO (Parallel I/O) I...				
	clk	Clock Input	*Double-click to export*	**clk_0**		
	reset	Reset Input	*Double-click to export*	[clk]		
	s1	Avalon Memory Ma...	*Double-click to export*	[clk]	0x10	0x1f
	external_connection	Conduit	**read_trngout_external_connection**			
	clk_read	PIO (Parallel I/O) I...				
	clk	Clock Input	*Double-click to export*	**clk_0**		
	reset	Reset Input	*Double-click to export*	[clk]		
	s1	Avalon Memory Ma...	*Double-click to export*	[clk]	0x0	0xf
	external_connection	Conduit	**clk_read_external_connection**			
	mem_address	PIO (Parallel I/O) I...				
	clk	Clock Input	*Double-click to export*	**clk_0**		
	reset	Reset Input	*Double-click to export*	[clk]		
	s1	Avalon Memory Ma...	*Double-click to export*	[clk]	0x20	0x2f
	external_connection	Conduit	**mem_address_external_connection**			
	mem_data_serial	PIO (Parallel I/O) I...				
	clk	Clock Input	*Double-click to export*	**clk_0**		
	reset	Reset Input	*Double-click to export*	[clk]		
	s1	Avalon Memory Ma...	*Double-click to export*	[clk]	**0x50**	0x5f
	external_connection	Conduit	**mem_data_serial_external_connection**			
	read_request	PIO (Parallel I/O) I...				
	clk	Clock Input	*Double-click to export*	**clk_0**		
	reset	Reset Input	*Double-click to export*	[clk]		
	s1	Avalon Memory Ma...	*Double-click to export*	[clk]	**0x40**	0x4f
	external_connection	Conduit	**read_request_external_connection**			
	stop_read	PIO (Parallel I/O) I...				
	clk	Clock Input	*Double-click to export*	**clk_0**		
	reset	Reset Input	*Double-click to export*	[clk]		
	s1	Avalon Memory Ma...	*Double-click to export*	[clk]	**0x30**	0x3f
	external_connection	Conduit	**stop_read_external_connection**			

Figura 13 - Diagrama de blocos do sistema de coleta de dados



Fonte: Leonardo Bosco Carreira (2019)

4 Resultados e discussões

4.1 Implementação do hardware

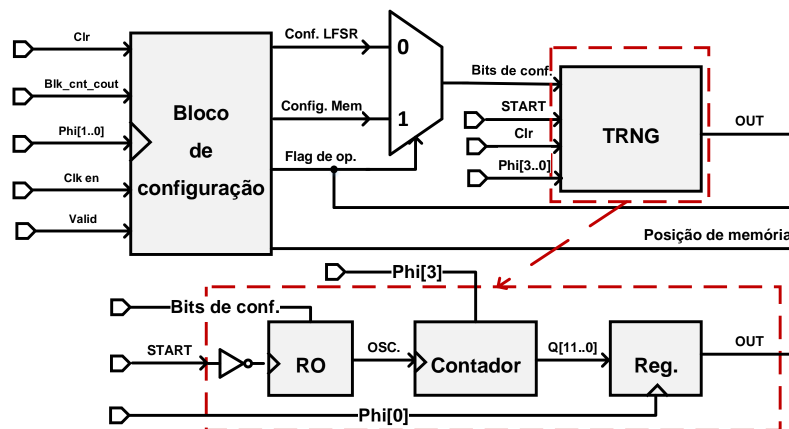
Esta sessão irá apresentar a implementação do sistema proposto nas FPGAs e discutir o funcionamento interno dos circuitos. A fim de facilitar o entendimento do leitor cada circuito, TRNG e o processador, serão divididos em entidades explicadas uma a uma no decorrer dessa sessão. O autor optou por não utilizar o diagrama de blocos fornecido pelo Quartus após a síntese do código para facilitar a leitura, visto que o compilador gera sinais internos que poluem a imagem dificultando a visualização. Para isso os blocos foram redesenhados mantendo apenas as lógicas de interesse.

4.1.1 TRNG

Assim como o circuito proposto por Yang et al. (2016) o TRNG é composto por 4 elementos principais, o oscilador em anel (RO) com multi-caminhos, um contador de 12 bits, um banco de registradores de 12 bits e um sistema para fornecer ao RO os bits de configuração necessários para definir o caminho de inversores.

A figura 14 apresenta o diagrama de blocos do circuito mencionado.

Figura 14 - Diagrama de blocos do sistema implementado na FPGA CycloneIV



Fonte: Leonardo Bosco Carreira (2019)

O TRNG, é idêntico ao proposto em [7]. Um bloco fornece ao oscilador os bits de configuração que indicam o caminho de inversores a ser utilizado, um sinal de início *START* dá início a oscilação do anel cuja saída é utilizada como *clock* de entrada de um contador de 12 bits que conta o número de oscilações até o colapso. O sinal *phi[0]* salva a saída do contador em um registrador. A saída do banco de registradores é enviada ao processador hospedeiro programado na outra placa através dos pinos de entrada e saída (GPIOs) das placas. O processador proposto será discutido a fundo no decorrer dessa seção.

Diferente do sistema proposto em [7] o bloco que fornece ao RO os bits de configuração não é apenas um LFSR, mas sim um conjunto de memória e um LFSR , o bloco de configuração.

Este é composto por um controlador de memória, que coordena a operação de escrita e leitura na memória, um *feedback linear shifter register* (LFSR) que gera os 96 bits de configuração para serem testados durante o modo de aprendizagem, e uma célula de memória de 96 bits x 64 posições com barramento de dados alimentado pelo LFSR.

Durante o modo de aprendizagem o LFSR é quem envia os bits de configuração para o oscilador em anel (RO). Caso a sequência sob teste é aprovada nos testes descritos na seção anterior, o sinal *valid* indica ao controlador de memória que a configuração deve ser salva, este por sua vez coordena o processo de escrita na memória e incrementa a posição de escrita. Uma vez que a posição de escrita chega ao máximo da memória, 64 posições no caso aqui implementado, um *flag de operação* indica ao multiplexador para alternar do modo de aprendizagem para o modo de operação fazendo com que o RO receba agora os caminhos direto da memória e desabilitando o LFSR para economia de energia.

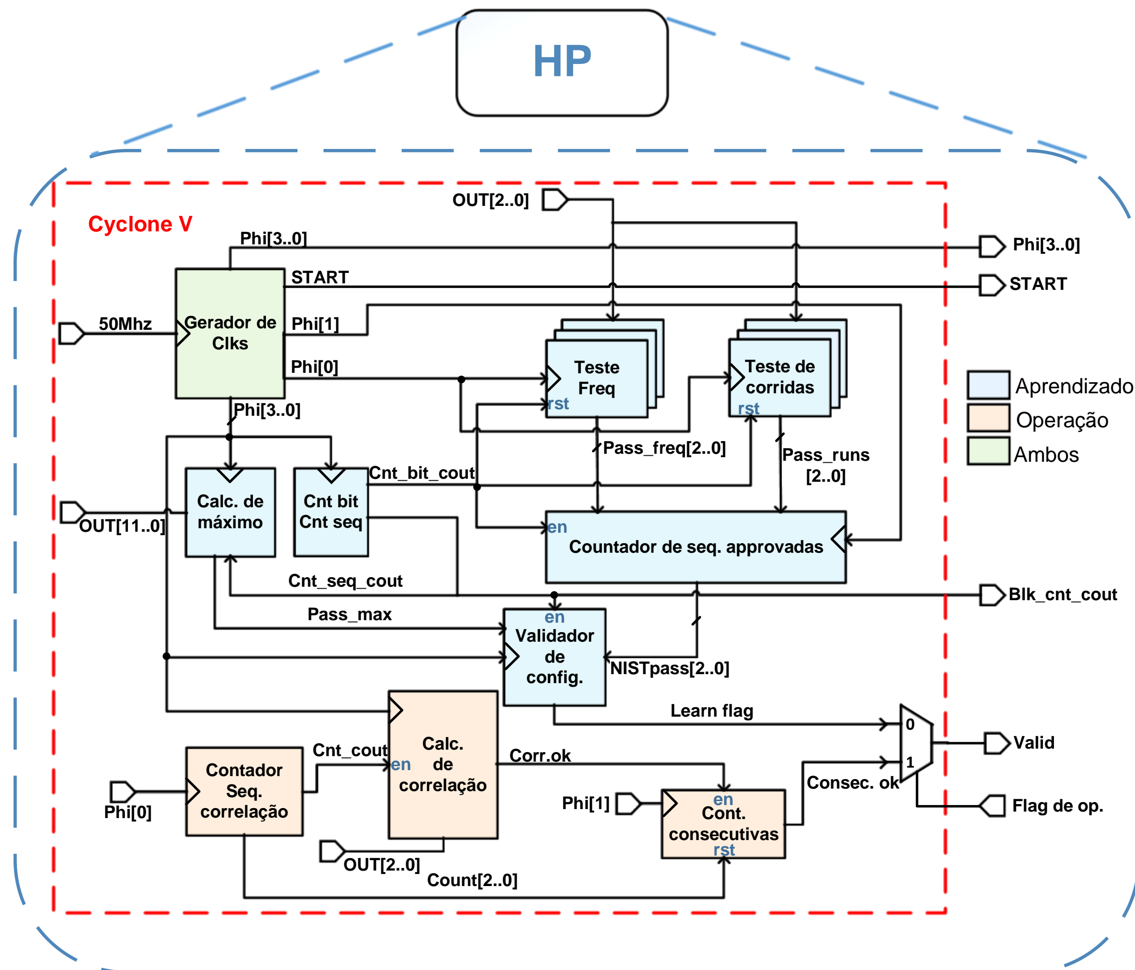
Como descrito na seção 3.2.2 o algoritmo de operação continua monitorando a saída do TRNG. Caso o sistema comece a produzir saídas enviesadas e reprove no teste de autocorrelação, o sinal *valid* é utilizado agora para avisar o controlador de memória que a posição de memória deve ser acrescida em uma posição. Assim o RO recebe a próxima configuração aprendida. Este processo se repete até que a saída analisada volte a ser aprovada no teste de autocorrelação.

4.1.2. Processador hospedeiro (Host Processor)

O processador é composto por 5 blocos de testes: teste frequência, teste de corridas, contador de sequências aprovadas, calculadora de máximo e teste de correlação, e 3 blocos de controle: gerador de *clocks*, contador de bits e sequências e validador de configuração.

A figura 15 apresenta o diagrama de blocos do processador implementado, onde as cores dos blocos dizem em qual estágio: aprendizagem, operação ou ambos, os blocos estão habilitados (medida utilizada para reduzir o consumo de energia do processador).

Figura 15 - Processador hospedeiro (HP) implementado na FPGA Cyclone V



Fonte: Leonardo Bosco Carreira (2019)

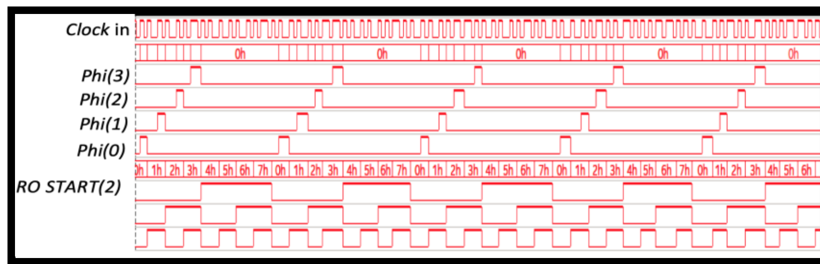
Para facilitar o entendimento do hardware os blocos serão explicados detalhadamente nessa sessão.

4.1.2.1 Gerador de *Clocks*

A esse bloco é atribuído o *clock* interno da placa de 50MHz. Um contador é utilizado para fazer a divisão do *clock* de entrada em até 8192x. O bit mais significativo do contador é utilizado como *clock* de referência para a criação dos *clocks* de sincronismo.

Uma vez que durante a operação de um ciclo o processador possui entidades em cadeia, ou seja, que a saída de uma entidade será o dado processado pela próxima, são necessários sinais de sincronismo para evitar erros de *setup time.* O diagrama de sinais da figura 16 mostra a conversão do *clock* de 50MHz nos *clocks* sincronizados.

Figura 16 - Sinais de clock gerados pelo bloco gerador de clocks



De cima para baixo: *Clock* de referência da placa 50MHz, *clocks* de sincronismo *Phi*[3..0] e o *clock* de início da oscilação START

Fonte: Leonardo Bosco Carreira (2019)

Como pode ser observado, o processador apenas irá trabalhar quando o resultado a ser processado já estiver na saída do TRNG, ou seja, quando *START* for nível logico baixo. As saídas *phi[3 . . . 0]* são os sinais de *clock* sincronizados que irão comandar todo o processamento dos dados.

4.1.2.3 Controlador de bits e sequências

Como explicado no algoritmo da seção 3.2 durante o modo de aprendizagem os testes realizados no processador são aplicados a 128 sequências de 256bits para cada configuração testada, atendendo aos requerimentos mínimos sugeridos pelo órgão regulamentador mostrado na seção 2.2. Este bloco utiliza dois contadores: Um para contar quantos bits de uma sequência já foram analisados e outro quantas sequências. Um sinal de controle é emitido quando os contadores atingem a contagem máxima indicando que uma sequência inteira já foi processada (*cnt_bit_cout*) ou todas as 128 sequências já foram analisadas (*cnt_seq_cout*).

4.1.2.3 Teste de frequência (frequency test)

Como descrito no algoritmo da seção anterior durante a aprendizagem as saídas do TRNG passam por testes para inferir se a configuração sob teste produz ou não saídas com alta entropia. Um dos testes é o teste de frequência descrito na seção 2.2. Esse teste é aplicado independentemente aos 3 LSBs das saídas do TRNG. A figura 17 apresenta o digrama de blocos do teste.

Figura 17 Diagrama de blocos do teste de frequência implementado na FPGA

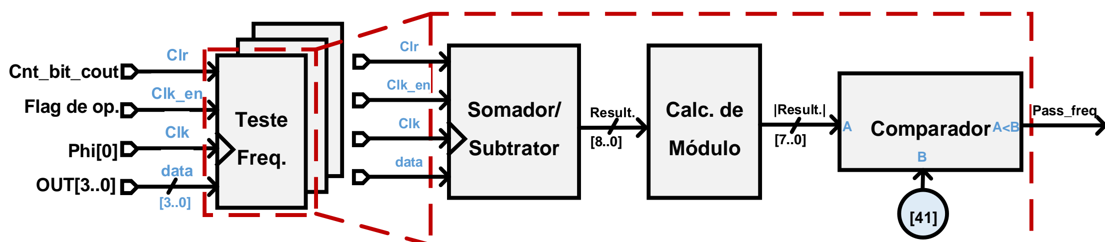


Diagrama de blocos do teste de frequência utilizado no processo de aprendizagem implementado dentro do processador proposto, com ênfase nas etapas internas aplicadas a cada LSB.

Fonte: Leonardo Bosco Carreira (2019)

A figura 17 ilustra o interior do bloco que realiza o teste de frequência em cada uma das saídas testadas. Um somador/subtrator adiciona um ao valor acumulado caso *data* seja '1' e subtrai um para '0'. A lógica combinacional subsequente calcula o valor absoluto de sua saída e o último bloco compara se o resultado é menor que a constante esperada, discutida na posteriormente na seção 4.2, que garante P-Valor maior que o recomendado 0.01.

4.1.2.3 Teste de corridas (runs test)

Assim como o teste de frequência, o teste de corridas também é aplicado aos três LSBs independentemente a fim de garantir que no final da aprendizagem as configurações salvas produzam na saída do TRNG 3 canais com alta entropia.

O conjunto de blocos destacados em azul conta o número de corridas da sequência analisada. Basicamente um flip-flop tipo D guarda o último bit analisado, um comparador compara o bit armazenado com o atual e caso o bit atual for diferente do anterior temos uma nova corrida, acrescenta-se um ao contador de corridas. O acumulador em verde, conta o número de uns na sequência sob análise acrescendo a soma em um toda vez que a saída em análise produz um bit '1'. Note que todas as saídas combinacionais de blocos que engatilham alterações nas entidades sequênciais são utilizadas como sinais de *enable* uma vez que não é boa prática utilizar saídas combinacionais como *clock* de entidades.

A LUT (*look up table*) destacada em laranja é a peça chave do bloco, uma vez que o teste de corridas possui dois graus de liberdade, ou seja, P-valor depende do número de corridas (V_{obs}) e do número de uns na sequência ($n\pi$). A fim de evitar cálculos extensos dentro do hardware foi criada uma LUT que, fornecido o número de '1's na sequência retorna qual a faixa de corridas que irão retornar P-valor maior que 0.01. A criação da LUT foi detalhado em mais detalhes na sessão 4.2.

O quarto e último destaque são os blocos comparadores responsáveis por comparar se o número de corridas encontradas está dentro do esperado para aprovação no teste (figura 18).

Figura 18 - Diagrama de blocos do teste de corridas implementado na FPGA

Diagrama de blocos do teste de corridas utilizado no processo de aprendizagem implementado dentro do processador proposto, com ênfase nas etapas internas aplicadas a cada LSB implementado dentro do HP

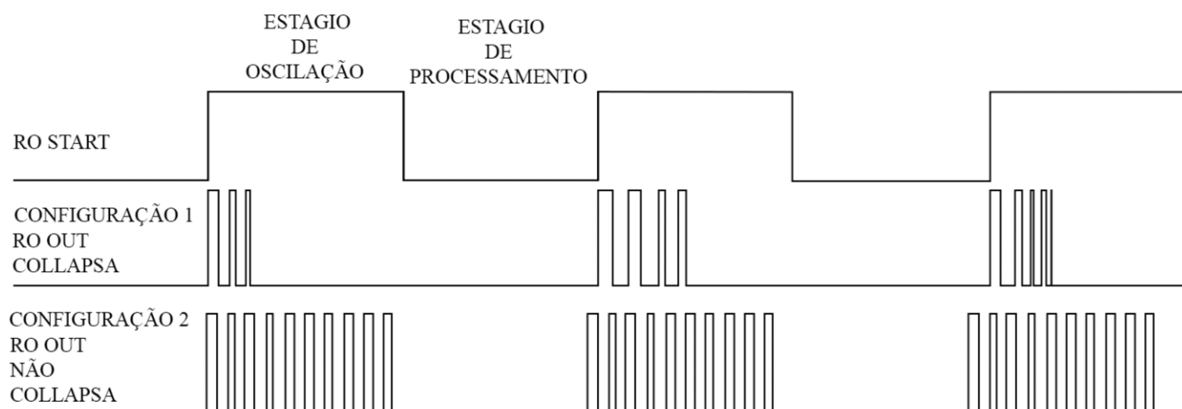
Fonte: Leonardo Bosco Carreira (2019)

4.1.2.3 Teste de máxima média de colapso (Max avg test)

A teste de máximo calcula a média de colapso da configuração em teste, ou seja, dentre os 128 X 256 = 32.768 colapsos observados para cada configuração qual é o valor médio. Dado o valor médio o bloco compara se ele não é maior que o máximo permitido.

O máximo permitido é um valor atribuído na hora do desenvolvimento baseado nos resultados observados do TRNG. Ele leva em consideração o máximo número de oscilações que cabem em meio pulso do *START* uma vez que a oscilação deve colapsar antes de *START* ir para o nível lógico '0', onde já se deve ter no registrador do TRNG a saída para que o processador a análise. Caso o oscilador não colapse dentro desse período a saída do contador será igual ao número máximo de oscilações que cabem dentro de meio período de *START*.

Figura 19 - Cenários de oscilação do RO



Clock de início da oscilação, um exemplo de configuração onde ocorre o colapso e outra onde não ocorre o colapso das bodas de subida e descida do RO mantendo o sistema oscilando por metade do período de *START*

Fonte: Leonardo Bosco Carreira (2019)

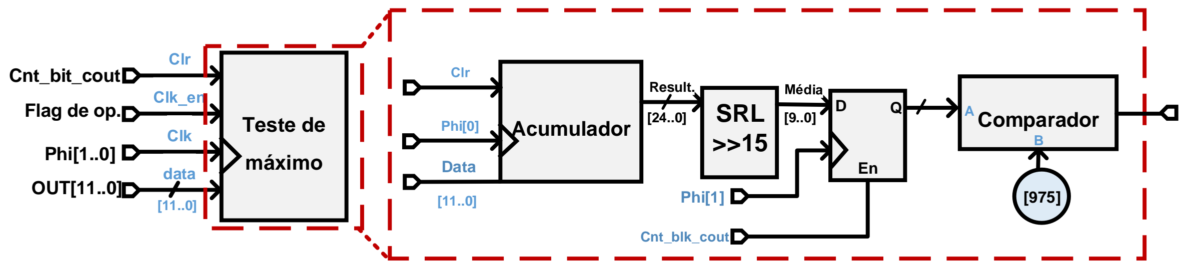
O diagrama da figura 19 ilustra o que esse teste tenta evitar. Uma configuração que tem *Jitter* suficiente para colapsar antes do estágio de processamento produz na saída do TRNG saídas que variam, nesse exemplo TRNG OUT = '011', TRNG OUT = '100' e TRNG OUT = '101'. Já configurações que trabalham no limite podem trancar a oscilação e produzir sempre o mesmo número, o valor máximo de pulsos que cabem em meio período, o que não é útil para geração de números aleatórios.

Para evitar que configurações que apresentam colapsos muito próximos ao máximo, calcula-se quantas oscilações cabem em meio período do pulso:

$$Osc_{max} = \frac{f_{osc}}{f_{start}} \times \frac{1}{2}$$

Essa constante será usada mais para frente para definir as configurações de aprendizagem.

Figura 20 - Diagrama de blocos do teste de máximo



Estão enfatizados os blocos utilizados para cálculo da média de colapso para as 128 sequências de 256 bits e a comparação com o valor máximo permitido durante o processo aprendizagem

Fonte: Leonardo Bosco Carreira (2019)

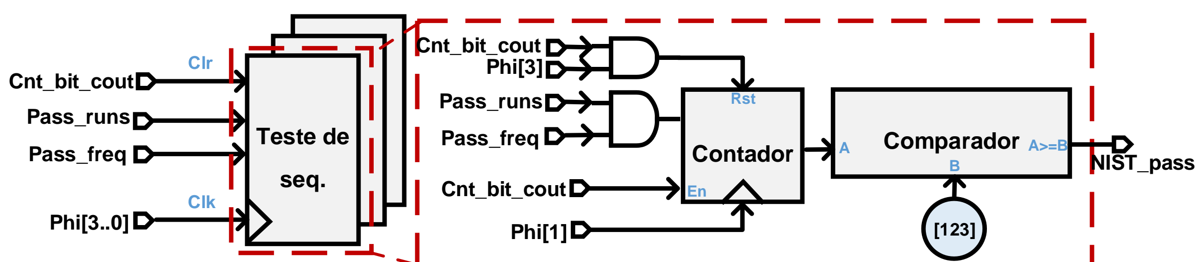
O cálculo da média de colapsos é apresentado na figura 20, usa-se um acumulador para somar os valores dos colapsos a cada corrida, como cada configuração é testada para 128 sequências de 256 colapsos, divide-se no final o valor acumulado por 128x256. A divisão é feita deslocando-se a saída em 15 bits à direita. Um comparador é usado para checar se o valor médio de colapsos está abaixo do máximo estipulado, 975. Mais a frente será discutido o motivo desse limite ter sido escolhido.

4.1.2.3 Contador de sequências aprovadas

Ambos testes, corridas e frequência, são realizados em 128 sequências de 256 bits por configuração testada. O número mínimo de sequências aprovadas deve ser 123 como descrito em [9] para cada canal (LSB). A entidade em destaque na figura 21tem a finalidade de analisar os resultados dos testes e verificar se o número necessário de sequências foram aprovadas.

Um contador acresce seu resultado em um cada vez que a sequência analisada passa em ambos testes. Um comparador compara o resultado com o esperado e caso maior ou igual o necessário a configuração testada é aprovada quanto aos testes de aleatoriedade implementados.

Figura 21 - Diagrama de blocos da entidade contadora de sequências aprovadas



A esquerda cada instanciação da entidade, uma para cada um dos 3LSBs. A direita o interior do bloco

Fonte: Leonardo Bosco Carreira (2019)

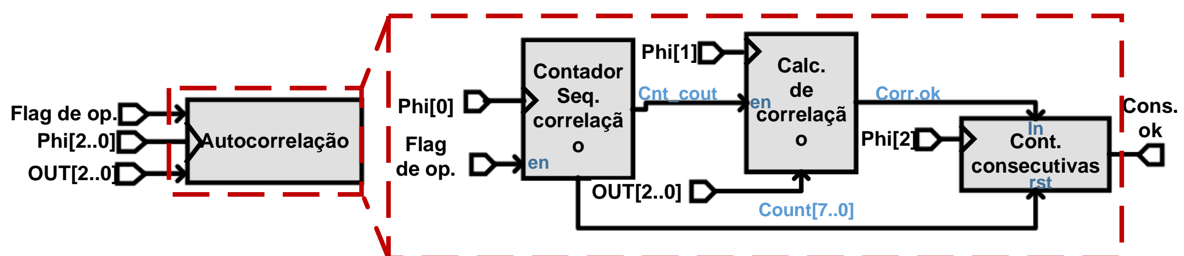
4.1.2.3 Teste de correlação (Autocorrelation test)

O teste de autocorrelação, como já mencionado, é apenas aplicado na fase de operação e utilizado para detectar qualquer viés na saída do TRNG. As saídas dos três canais aleatórios desejados são analisadas como uma só saída, ou seja, forma-se a seguinte sequência concatenando os 3LSBs:

OUT [0]	OUT [1]	OUT [2]	OUT [0]	OUT [1]	...	OUT [2]	...

A razão para o uso da sequência concatenada é acelerar o processo de detecção e resposta do sistema a um eventual viés na saída, uma vez que o teste necessita 512bits para calcular o coeficiente de correlação. Para determinar cada coeficiente utiliza-se então de apenas 512/3 colapsos. Como descrito na seção 3.2.2, era necessário que para cada configuração calculasse-se 3 coeficientes de correlação para evitar falsos negativos e apenas perante reprovação dos três a configuração seria alterada. O diagrama da figura 22 ilustra como o bloco foi desenvolvido.

Figura 22 - Diagrama de blocos da entidade de detecção de viés



Fonte: Leonardo Bosco Carreira (2019)

Uma combinação de registradores, acumuladores, contadores e portas lógicas é utilizada para calcular o coeficiente de correlação (XOR) entre bits adjacentes e acumulá-los por 171 colapsos, contados pelo contador de sequências de correlação.

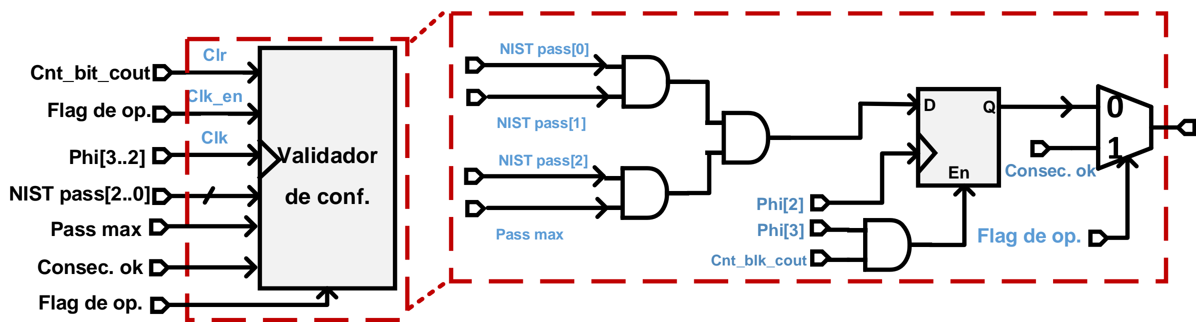
Ainda dentro da entidade Calculadora de correlação o valor obtido é comparado com o valor esperado, entre 224 e 289 como descrito em [10]. O resultado da comparação é salvo na entidade *contador de consecutivas* em um *shift register*. O processo é repetido mais 2x e o número de repetições é contabilizado por um contador dentro dessa mesma entidade. Por fim uma lógica OU recebe a saida dos 3 D-FF do *shift register* permitindo assim analisar se pelo menos uma das 3 sequências retornou *corr.ok* verdadeiro então atribuir *consec.ok = '1'.*

4.1.2.3 Validador de configuração

Trata-se do bloco que informa ao controlador de memória se a configuração deve ser salva, durante o modo de aprendizagem, ou trocada durante o modo de operação. Um multiplexador recebe o sinal *flag de operação* que informa ao sistema se está no modo de operação ou de aprendizagem. Uma porta AND verifica se os blocos foram aprovados tanto nos testes NIST (Nist pass = '1 ') quanto no teste de máximo (Pass max = '1'). A saída do circuito combinacional é salva em um Flip flop de

dados para garantir que esteja disponível ao bloco célula de memória quando os resultados dos testes forem requisitados no *clock phi[1].* Mais uma vez, como utilizado em quase todos os blocos, o *clock* phi[3] juntamente com o *cnt_blk_cout* é utilizado para sincronizar o RST do processador. A figura 23 apresenta a entidade implementada.

Figura 23 - Diagrama de blocos da entidade validador de configurações



Fonte: Leonardo Bosco Carreira (2019)

4.2 Configuração do sistema – Estabelecimento das constantes de aprendizagem

Como mencionado nas seções anteriores durante o processo de aprendizagem são utilizadas constantes que garantem que as configurações aprendidas gerem o número desejado de canais aleatórios. Por exemplo a frequência de oscilação do RO depende indiretamente do número de estágios inversores. A tabela 1 mostra as frequências medidas com o uso de um osciloscópio conectado à saído do RO. Os valores das constantes de comparação dos testes de frequência, corridas e máximo são determinantes para garantir a qualidade da saída. Por esse motivo foi dedicada uma seção para guiar o leitor na correta determinação dessas constantes e apresentar os *trade-offs* de cada escolha.

Tabela 1 - Relação do número de estágios inversores com a frequência de oscilação do RO

Número de estágios inversores (S)	**Frequência de oscilação do anel (f_{RO})**
32	12MHz
4	96MHz

Fonte: Leonardo Bosco Carreira (2019)

4.2.1 Número de estágios (S)

Para o projeto aqui proposto todos resultados foram todos obtidos para um RO de 32 estágios (S) inversores. Para justificar a escolha por um número maior de estágios deve-se recorrer as demonstrações da seção 3.1. O colapso é representado por uma função distribuição de probabilidade (pdf) de uma inversa gaussiana cujo valor médio e o parâmetro de modelagem são dados por (11) e (12) . Como pode-se notar o valor médio (μ) é obtido pelo inverso de uma distribuição meia-normal -

$|\mathcal{N}(0{,}2S\sigma^2)|$ multiplicada por uma constante $|D_2 - D_1|$. Logo a pdf do valor médio pode ser extraída derivando-se a função de distribuição cumulativa (cdf) de uma meia-normal: $F(y;\sigma) = \text{erf}(\frac{y}{\sqrt{2\sigma^2}})$ onde $y = \frac{c}{x}$.

$$F_Y(y) = F_x\left(\frac{c}{x}\right) = \text{erf}\left(\frac{c}{x\sqrt{2 \times 2S\sigma_{process}^2}}\right) \therefore \quad (14)$$

$$\frac{dF_Y}{dx} = f_Y(y) = \frac{|D_2 - D_1|}{y^2\sqrt{2S\sigma_{process}^2\pi}} \times e^{\left(-\left(\frac{|D_2 - D_1|}{2\sigma_{process} y\sqrt{S}}\right)^2\right)} \quad (15)$$

Derivando e igualando a zero (15) obtém-se o valor de y que garante a máxima probabilidade de ocorrência:

$$y|\max(f_Y(y)) = \left(\frac{|D_2 - D_1|}{2\sigma_{process}\sqrt{S}}\right) \quad (16)$$

A diferença $|D_2 - D_1|$ pode ser aproximada pela expressão $S \times Ideal_delay$, visto que expressa o tempo necessário para os sinais A e B percorrerem os S estágios do anel. Assim tem-se:

$$y|\max(f_Y(y)) = \left(\frac{|D_2 - D_1|}{2\sigma_{process}\sqrt{S}}\right) = S \times \frac{Ideal_delay}{2 \times \sigma_{process} \times \sqrt{S}} \quad (17)$$

Em estatística a razão do desvio padrão pelo valor médio é denominada coeficiente de variação $C_v = \left(\frac{\sigma}{\mu}\right)$, como Ideal_delay é o valor médio do atraso de cada estágio e $\sigma_{process}$ o desvio padrão causado pela variação no processo de fabricação [7] do circuito integrado. Tem-se:

$$y|\max(f_Y(y)) = \left(\frac{|D_2 - D_1|}{2\sigma_{process}\sqrt{S}}\right) = S \times \frac{Ideal_delay}{2 * \sigma_{process} \times \sqrt{S}} = \frac{\sqrt{S}}{2 \times C_v} \quad (18)$$

Isso demonstra que quanto maior o número de estágios do RO, maior será a probabilidade de o valor médio de colapsos ser elevado, o que é desejável para se obter maior número de canais aleatórios como demonstrado na Figura *8*.

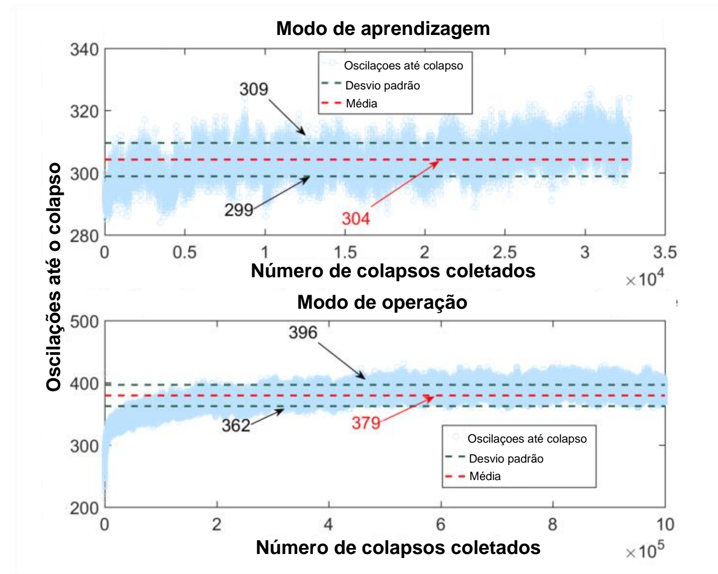
4.2.2 Máxima média de colapsos permitida

A placa utilizada possui um sinal de clock interno de 50MHz. Através do bloco gerador de *clocks* a frequência foi reduzida para $f_{start} = 6KHz$. O que nos leva a um máximo números de oscilações de:

$$Osc_{max} = \frac{f_{osc}}{f_{start}} \times \frac{1}{2} \cong 1300\ oscilações$$

Foi notado que do momento da aprendizagem até minutos após o início do modo de operação a elevação da temperatura do núcleo causava um aumento significativo da média de colapsos, em torno de 25%. Conforme mostrado na figura 24.

Figura 24 - Comportamento da saída OUT durante o modo de aprendizagem e o modo de operação



Comportamento extraído para configuração zero ao ligar a placa e após o processo de aprendizagem, mostrando uma elevação na média de colapsos

Fonte: Leonardo Bosco Carreira (2019)

Para evitar então que configurações que colapsem muito próximo ao valor máximo permitido fossem aprendidas a constante utilizada no teste de máximo foi estabelecida como 975, 75% de Osc_{max}.

4.2.3 Testes de aleatoriedade NIST

As configurações das constantes de aprendizagem dos testes de aleatoriedade foram baseadas no valor sugerido pelo órgão regulamentador para o número de blocos analisados. Uma vez que o sistema proposto realiza os testes em 128 sequências de 256 bits o órgão indica que ao menos γ das sequências analisadas devem passar aos testes, onde γ é dado por:

$$\gamma \geq (1-\alpha) - \sqrt{\left(\frac{(1-\alpha)\alpha}{M}\right)} \tag{19}$$

Substituindo $\alpha = 0.01$ e M = 128, tem-se que $\gamma \geq 123$. Esse valor foi atribuído como limite mínimo para aprovação da configuração em ambos os testes como mostrado na Figura 23.

4.2.3.1 Teste de frequência monobit

Como mostrado na seção 2 o P-valor do teste de frequência é dado pela equação $[eq]$ e deseja-se que este seja superior ao nível de significância α = 0.01 para garantir aprovação da sequência

$$P - valor = erfc\left(\frac{|S_n|}{\sqrt{2 \times N}}\right) \geq 0.01 \quad (20)$$

A implementação da função erro complementar ($erfc$) em hardware é complexa e aumentaria o a área e o tempo de computação do sistema proposto. Como o HP analisa sempre sequências de mesmo tamanho,$n = 256$, (20) foi solucionada para $|S_n|$ previamente com auxílio do *Matlab*, apêndix A. O resultado obtido foi:

$$erfc\left(\frac{|S_n|}{\sqrt{2 \times N}}\right) \geq 0.01 \rightarrow |S_n| \leq 41$$

Logo a constante limite para aprovação de uma sequência de 256bits no teste de frequência é 41, valor atribuído diretamente ao comparador como demonstrado na Figura 17.

4.2.3.2 Teste de corridas

O mesmo procedimento foi realizado para o teste de corridas (*runs test)* porém como este teste possui dois graus de liberdade, como descrito na seção 2.2, o código do *Matlab* variou o número de '1's na sequência (*nπ)* de 1 a 256 e calculou os valores de corridas (V_{obs}) que levariam a $P - valor \geq$ 0.01 para cada caso. A tabela 2 é um resumo do resultado encontrado. A tabela completa e o código utilizado para cálculo e escrita do código VHDL podem ser encontradas no apêndice B.

$$P_{value} = erfc\left(\frac{|V_{obs} - 2n\pi(1-\pi)|}{2\sqrt{2n\pi(1-\pi)}}\right), solucionada\ para\ P_{value} \geq 0.01$$

Tabela 2 - Look-up-table implementada na entidade LUT do processador hospedeiro (HP)

Número de '1's	Número mínimo de corridas	Número máximo de corridas
1	2	2
...	...	...
127	108	148
...	...	...
200	74	101
...	...	...
255	2	2

No hardware programado essa tabela foi salva em uma memória indexada pelo número de '1's, chamada de LUT (*look up table*) na figura 18.

4.4 Testes

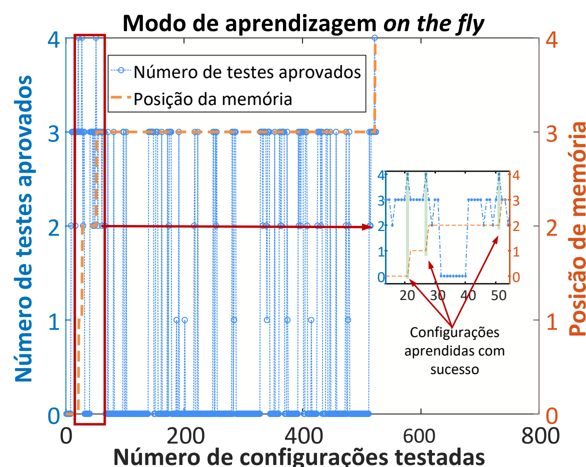
Com intuito de comprovar o funcionamento do sistema, nessa seção serão apresentados os seguintes itens: resultados do modo de aprendizagem, eficiência do método de busca guiada, saída e performance nas análises de aleatoriedade e resposta do sistema desenvolvido ao enviesamento forçado da saída.

4.4.1 Funcionamento do algoritmo de aprendizagem em tempo real

A fim de demonstrar o correto funcionamento do algoritmo proposto, com o uso do sistema de coleta de dados apresentado na seção 3.4, foram coletados os seguintes dados do HP para as primeira 500 configurações testadas: vetor posição de memória (nome),vetor de aprovação nos testes NIST (pass_NIST[2..0]) e aprovação no teste de colapso máximo (pass max).

Os resultados dos testes para cada configuração de aprendizagem foram somados, obtendo-se um número de 0 a 4, onde 0 representa reprovação em todos os testes e 4 aprovações em todos. A configuração deve ser escrita na memória apenas se todos os testes forem aprovados. A figura 25 apresenta os resultados obtidos em tempo real (*on-the-fly)* com o uso do sistema embarcado.

Figura 25 - Comportamento do HP implementado durante o modo de aprendizagem



Resultado dos testes de aprendizagem para cada configuração testada e o ponteiro posição de memória , com enfoque a direita para as configurações aprendidas.

Fonte: Leonardo Bosco Carreira (2019)

Como observado na figura 25, o sistema implementado funciona exatamente como proposto, a figura menor enfatiza os pontos onde as configurações aprovadas foram escritas na memória e o ponteiro posição de memória acrescido em uma posição.

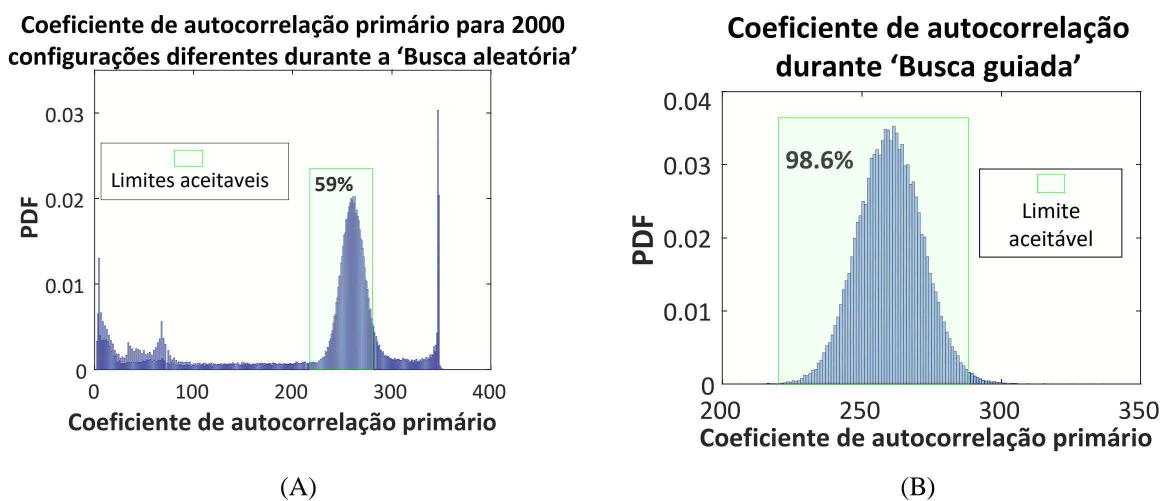
4.4.2 Eficiência do método de busca guiada

Outro ponto importante é comprovar aos leitores a eficiência do método proposto em relação ao proposto anteriormente em [7] . Para isso foram coletados os coeficientes de autocorrelação

primário calculados para todas as configurações salvas na memória, 64 configurações, e para as primeiras 2000 configurações durante o processo de aprendizagem.

As figura 26 (A) e 26 (B) comprovam a maior eficiência do sistema durante a busca guiada, provando que após a aprendizagem baseada nos testes NIST e de colapso máximo as configurações salvas apresentam 98.6% de chance de produzirem os resultados de aleatoriedade desejado, medido aqui pelo coeficiente de autocorrelação, diferentemente da busca aleatória onde as chances se mostraram 2X menores.

Figura 26 - Distribuição de probabilidade do coeficiente de autocorrelação primário medido



Distribuição coletada do sistema para os casos: (A) Busca aleatória e (B) busca guiada

Fonte: Leonardo Bosco Carreira (2019)

4.4.2 Teste de aleatoriedade

Para reforçar a eficiência do método de aprendizagem deve-se também demonstrar que as configurações aprendidas apresentam um bom desempenho em todos os testes do órgão regulamentador NIST. Para isso foi utilizado o software fornecido em [36]. Ao *software* é fornecido um arquivo contendo 110Mbits da saída do TRNG implementado, coletados através do sistema descrito na seção 3.4. O código em linguagem 'C' utilizado se encontra no apêndice C. O arquivo foi dividido em 110 sequências de 1Mbits conforme exigência para garantir confiabilidade dos resultados [9]. A tabela 3 apresenta a performance do hardware desenvolvido em todos os testes executados.

Tabela 3 - Resultados dos testes estatísticos de aleatoriedade do NIST para V_{dd}= 1.2v

Resultado dos testes estatísticos de aleatoriedade NIST †		
Testes	$\boldsymbol{P - valor_t}$ [a]	**Número de sequências aprovadas** [b]
Frequency	0.5159	110
Block Frequency	0.0073	109
Cumulative Sums Forward	0.9229	110
Cumulative Sums Reverse	0.9947	110
Runs	0.9229	109
Longest Run	0.4979	106
Rank	0.793	109
FFT	0.3654	108
Non-Overlapping Temp	0.0837	104
Overlapping Template	0.0885	105
Universal	0.5711	109
Approximate Entropy	0.7215	104
Random Excursion	0.0069	65[c]
Random Excursion Variant	0.0127	66[c]
Serial	0.2574	108
Serial	0.8421	110
Linear Complexity	0.5711	110
†Para testes com mais de um subtestes (Non-overlapping Template; Random excursion e Random excursion variant) o P-valor e o número de sequências aprovadas são os menores observados dentre todos os subtestes		
a O $\boldsymbol{P - valor_t}$ representa a uniformidade dos P-valores de todas as sequências analisadas e seu valor mínimo de aprovação é 0.00001 **[9]**		
b Valor mínimo para aprovação é de 105 sequências	c Valor mínimo para aprovação de 62 sequências	

Teste sugeridos pelo National Institute of standard and technology (NIST) aplicado as saídas do gerador de número aleatórios para condições normais de operação, 1.2v e 25oC

Fonte: Leonardo Bosco Carreira (2019)

4.5 Resposta ao enviesamento forçado da saída

Por último é necessário mostrar a eficácia do processo de detecção e recuperação do sistema proposto quando exposto a saídas enviesadas . Como o sistema desenvolvido foi implementado em uma FPGA ataques de indução de frequência como proposto em [7] se mostraram inviáveis de se aplicar pois o circuito integrado da FPGA possuía internamente filtros que bloquearam qualquer tentativa de injeção ruído na fonte de alimentação. A saída para forçar um enviesamento na saída foi então reduzir a tensão de alimentação. Como demonstrado em [28] a variância do Jitter em osciladores em anel é dada pela seguinte equação:

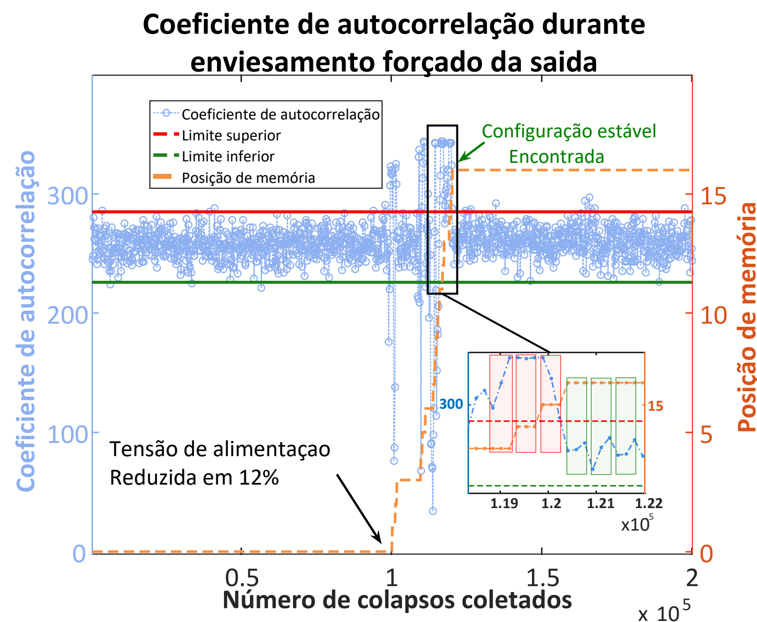
$$\sigma^2 = \frac{4kT\gamma_N t_{dN}}{I_N(V_{dd} - V_{th})} + \frac{kTC}{I_N^2} \tag{21}$$

Onde k é a constante de *boltzman*, T é a temperatura de operação, γ é o coeficiente de ruído, C é a capacitância de carga do inversor, t_{dN} é a janela de integração do ruído que pode ser representada pelo tempo carga e descarga do inversor, V_{th}é a tensão de *threshold* do transistor, I_N é a corrente de carga e descarga dos transistores e V_{dd} é a tensão de alimentação do inversor.

Como mostrado pela equação 21 caso a tensão de alimentação seja reduzida a variância do ruído tende a aumentar reduzindo então o número de oscilações que cada configuração leva até colapso tornando a saída mais enviesada uma vez que baixos valores de colapso levam a menos canais aleatórios como mostrado na *Figura 8*.

Para realizar tal medida o regulador de tensão do kit foi removido e uma fonte externa de alimentação foi utilizada para alimentar diretamente o núcleo da FPGA, o processo de aprendizagem foi realizado para a tensão típica de funcionamento, 1,2v, após alguns ciclos no modo de operação a tensão foi reduzida em 12%. O gráfico da figura 27 mostra o resultado obtido:

Figura 27 - Comportamento do sistema de detecção e correção de viés



Resposta em tempo real do sistema de detecção e correção de viés de saída aplicado as saídas do gerador de números aleatórios durante um viés forçado através da redução de tensão de alimentação V_{dd}

Fonte: Leonardo Bosco Carreira (2019)

No caso evidenciado o sistema proposto levou apenas 14 configurações até a estabilização em uma nova configuração que produzisse resultados dentro do esperado para a nova tensão de alimentação, 30% mais rápido que o proposto em [7].

A saída após a estabilização foi testada em todos os testes de aleatoriedade e assim como os resultados da sessão anterior apresentou um ótimo desempenho reprovando apenas no teste de superposição de *template**, porém passando em todos os demais testes conforme mostrado na tabela 4.

Tabela 4 - Resultados dos testes estatísticos de aleatoriedade do NIST para $V_{dd} = 1v$ após correção de viés

Resultado dos testes estatísticos de aleatoriedade NIST †		
Testes	$P - valor_t$ [a]	**Número de sequências aprovadas** [b]
Frequency	0.9435	110
Block Frequency	0.5159	110
Cumulative Sums Forward	0.793	109
Cumulative Sums Reverse	0.7757	109
Runs	0.0596	109
Longest Run	0.2695	109
Rank	0.4124	110
FFT	0.9229	109
Non-Overlapping Temp*	0.00001	101
Overlapping Template	0.0033	106
Universal	0.703	109
Approximate Entropy	0.8263	101
Random Excursion	0.0611	67[c]
Random Excursion Variant	0.087	67[c]
Serial	0.2032	109
Serial	0.48024	106
Linear Complexity	0.0215	108
†Para testes com mais de um subtestes (Non-overlapping Template; Random excursion e Random excursion variant) o P-valor e o número de sequências aprovadas são os menores observados dentre todos os subtestes		
a O $\boldsymbol{P - valor_t}$ representa a uniformidade dos P-valores de todas as sequências analisadas e seu valor mínimo de aprovação é 0.00001 **[9]**		
b Valor mínimo para aprovação de 105 sequências	c Valor mínimo para aprovação de 62 sequências	

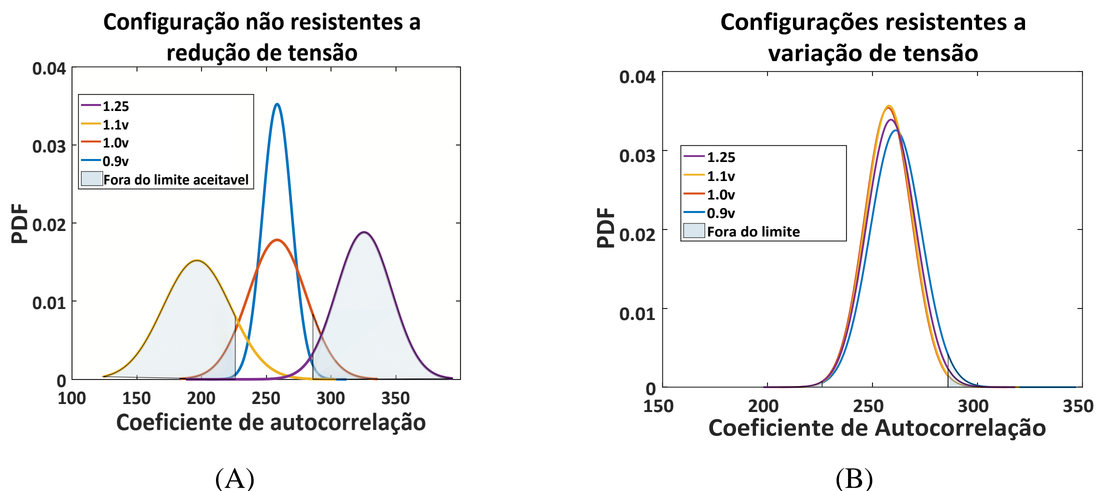
Teste sugeridos pelo National Institute of standard and technology (NIST) aplicado as saídas do gerador de número aleatórios após correção de viés para condições anormais de operação, 1.0v e 25oC

Fonte: Leonardo Bosco Carreira (2019)

Vale ressaltar que os resultados da *Figura 27* podem ainda ser otimizados se durante a aprendizagem o sistema seja exposto a diferentes condições de temperatura e tensão, e na memória sejam salvas não apenas as configurações mas também as condição atuais de tensão e temperatura a qual o TRNG está exposto, tornando o método de busca guiada mais eficiente e direcionado. Esse tema já é pesquisado pelo autor que trará resultados em breve em uma futura publicação.

Conquanto mesmo sem aplicar o procedimento descrito o sistema ainda responde de maneira eficiente, isso se deve ao fato de que ao basear a aprendizagem apenas nos resultados de aleatoriedade e não em um range de média de colapso igual feito por Yang et Al. (2016) algumas configurações aprendidas por mais tenham o valor médio de colapso reduzido ainda continuam a colapsar com valores médios acima do necessário para gerar o número desejado de canais aleatórios na saída continuando assim a produzir saídas não enviesadas. As figuras 28 (A) e 28 (B) exemplificam o caso discutido aqui:

Figura 28 - Comportamento do coeficiente de autocorrelação de duas configurações distintas



Comportamento do coeficiente de autocorrelação da saída OUT gerada por duas configurações distintas sob variação da tensão de alimentação do núcleo da FPGA em (A) Configuração não resistente redução de tensão (B) Configuração resistente a variação de tensão

Fonte: Leonardo Bosco Carreira (2019)

Como pode-se ver, uma configuração resistente a variações de tensão continua sendo aprovada no teste de autocorrelação até mesmo para redução de mais de 30% enquanto uma configuração não resistente começa a produzir maior número de bits ‘1’ ou ‘0’, o que leva ao enviesamento da saída e reprovação no teste de autocorrelação.

4.6 Performance e comparação com trabalhos anteriores

A seguir os resultados do projeto aqui implementado foram comparados quanto a área, potência, número de canais aleatórios e correção de viés com três trabalhos anteriores, um o estado da arte base para esse projeto e os demais dois TRNGs baseados em *beat frequency* implementados em FPGAs.

Tabela 5 - Tabela comparativa de performance

		Este trabalho		[7] JSSC'16	[37] CICC'14	[38] TCAS-II'17
Fonte de entropia		**RO reconfigurável**		RO reconfigurável	Beat frequency of free-running RO	Beat frequency of two free-running RO
Tecnologia		**Altera Cyclone V DE-10**	**Altera Cyclone IV**	40nm CMOS	Xilinx Virtex V	Xilinx Virtex V
Entidade		**Host processor (HP)**	**TRNG**	Reconfigurable TRNG (HP off-chip)	TRNG	TRNG
Área	Registradores	**179**	**348**	$836\ \mu m^2$	26	26
	LUTs	**339**	**885**		48	33
	Memória	**N/A**	**3072**		N/A	N/R
	Gerenciador de *clocks*	**N/A**	**N/A**		1[b]	2[c]
Potência (mW)		**180**	**80**	0.046	1384	1470
Frequência de saida (kHz)		**6**		667	N/R	223
Número de canais aleatórios		**3**		3	4	3
Número de tentativas para correção em tempo real & probabilidade de sucesso por tentativa		**522 98% (busca guiada) 59% (busca aleatória)**		6250-62500[e] 5-8%	Sem correção em tempo real	Sem correção em tempo real
Post Processor		**No**		No	Yes	Yes

Como pode ser visto o sistema aqui proposto apresenta maior consumo de recursos que os demais baseados em FPGAs, porém isso pode ser explicado pelo fato que nenhum dos dois trabalhos possui um processamento em tempo real da sequência para correção de viés.

O consumo de potência do sistema proposto é bem inferior aos demais implementados em FPGA e superior ao desenvolvido em [7] o que era esperado por se tratar de um circuito integrado dedicado aquela função. Porém a frequência de saída que pôde ser obtida foi muito inferior o que explica a diferença de potência consumida, devido ao fato que foi necessário ampliar o número de estágios para garantir a quantidade de saídas aleatórias desejadas. A arquitetura de multicaminhos também aumentou o *fanout dos inversores* aumentando a capacitância de carga e reduzindo a velocidade de oscilação.

Vale ressaltar que o objetivo do projeto era propor um sistema de detecção e correção de viés rápido, o que foi cumprido, garantindo uma resposta até 120x mais rápida que o estado da arte estudado. Porém não houve enfoque em otimização do circuito para redução do uso de recursos.

5 Conclusão

Este trabalho apresentou um processador para detecção e correção de viés na saída de geradores de números aleatórios simples com alto potencial de integração. Os resultados obtidos comprovam a eficácia do método proposto garantindo uma recuperação até 120x mais rápida que o estado da arte estudado. O algoritmo proposto, ainda pode ser aprimorado para um sistema de aprendizado mais abrangente que leve em conta as condições do ambiente no momento da aprendizagem, aumentando as chances de que o tempo de recuperação mínimo seja utilizado.

Além da possibilidade citada, o processador aqui proposto pode ser facilmente adaptado a diversas outras topologias de TRNGs uma vez o método de aprendizagem, detecção e correção é baseado em testes estatísticos e não em saídas especificas da arquitetura utilizada.

Por fim, a arquitetura proposta pode ser estendida para uso em diversas necessidades atuais como *compressive sensing,* redes autônomas de sensoriamento e autenticação de dispositivos e rede *wireless* tema da atual pesquisa do autor.

REFERÊNCIAS

[1] D. Evans, “The Internet of Things: How the Next Evolution of the Internet is Changing Everything”, *Cisco Internet Bus. Solut. Group IBSG*, vol. 1, p. 1–11, jan. 2011.

[2] “Cisco Visual Networking Index: Forecast and Trends, 2017–2022 White Paper”, *Cisco*. [Online]. Disponível em: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.html. [Acessado: 30-jun-2019].

[3] H. Bauer, M. Patel, e J. Veira, “The Internet of Things: Sizing up the opportunity | McKinsey & Company”. [Online]. Disponível em: https://www.mckinsey.com/industries/semiconductors/our-insights/the-internet-of-things-sizing-up-the-opportunity. [Acessado: 24-jun-2018].

[4] “Hacker took control of United flight and flew jet sideways, FBI affidavit says”, *UPI*. [Online]. Disponível em: https://www.upi.com/Top_News/US/2015/05/16/Hacker-took-control-of-United-flight-and-flew-jet-sideways-FBI-affidavit-says/2421431804961/. [Acessado: 24-jun-2018].

[5] “Hardware security in the IoT”. [Online]. Disponível em: http://www.embedded-computing.com/embedded-computing-design/hardware-security-in-the-iot. [Acessado: 24-jun-2018].

[6] A. Leon-Garcia e A. Leon-Garcia, *Probability, statistics, and random processes for electrical engineering*, 3rd ed. Upper Saddle River, NJ: Pearson/Prentice Hall, 2008.

[7] K. Yang, D. Blaauw, e D. Sylvester, “An All-Digital Edge Racing True Random Number Generator Robust Against PVT Variations”, *IEEE J. Solid-State Circuits*, vol. 51, nº 4, p. 1022–1031, abr. 2016.

[8] P. Kirschenmann, “Concepts of randomness”, *J. Philos. Log.*, vol. 1, nº 3, p. 395–414, ago. 1972.

[9] A. Rukhin, J. Soto, e J. Nechvatal, “A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications”, p. 131.

[10] B. Yang, V. Rožić, N. Mentens, W. Dehaene, e I. Verbauwhede, “TOTAL: TRNG on-the-fly testing for attack detection using Lightweight hardware”, in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2016, p. 127–132.

[11] Y. Dodge, “A Natural Random Number Generator”, *Int. Stat. Rev. Rev. Int. Stat.*, vol. 64, nº 3, p. 329–344, 1996.

[12] O.-O. D. Science, “Properly Setting the Random Seed in ML Experiments. Not as Simple as You Might Imagine”, *Medium*, 08-maio-2019. [Online]. Disponível em: https://medium.com/@ODSC/properly-setting-the-random-seed-in-ml-experiments-not-as-simple-as-you-might-imagine-219969c84752. [Acessado: 01-jul-2019].

[13] A. A. Rahimi, L. B. Carreira, e S. Gupta, “Synchronous multi-signal acquisition for WBSNs using gold-code based joint-compressive sensing”, in *2016 IEEE Biomedical Circuits and Systems Conference (BioCAS)*, 2016, p. 236–239.

[14] B. Chatterjee, D. Das, S. Maity, e S. Sen, “RF-PUF: Enhancing IoT Security Through Authentication of Wireless Nodes Using In-Situ Machine Learning”, *IEEE Internet Things J.*, vol. 6, nº 1, p. 388–398, fev. 2019.

[15] P. L’Ecuyer, “History of uniform random number generation”, 2017, p. 202–230.

[16] M. G. Kendall e B. B. Smith, “Randomness and Random Sampling Numbers”, *J. R. Stat. Soc.*, vol. 101, nº 1, p. 147–166, 1938.

[17] W. O. Kermack e A. G. McKendrick, "XVII.—Tests for Randomness in a Series of Numerical Observations", *Proc. R. Soc. Edinb.*, vol. 57, p. 228–240, ed 1938.

[18] W. E. Thomson, "ERNIE—A Mathematical and statistical analysis", *J.Roy.Stat.Soc.*, vol. A122, p. 301–324, jan. 1959.

[19] C. S. Petrie e J. A. Connelly, "A noise-based IC random number generator for applications in cryptography", *IEEE Trans. Circuits Syst. Fundam. Theory Appl.*, vol. 47, nº 5, p. 615–621, maio 2000.

[20] M. Matsumoto, S. Yasuda, R. Ohba, K. Ikegami, T. Tanamoto, e S. Fujita, "1200 #x003BC;m2 Physical Random-Number Generators Based on SiN MOSFET for Secure Smart-Card Application", in *2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*, 2008, p. 414–624.

[21] C. Tokunaga, D. Blaauw, e T. Mudge, "True Random Number Generator With a Metastability-Based Quality Control", *IEEE J. Solid-State Circuits*, vol. 43, nº 1, p. 78–85, jan. 2008.

[22] N. Liu, N. Pinckney, S. Hanson, D. Sylvester, e D. Blaauw, "A true random number generator using time-dependent dielectric breakdown", in *2011 Symposium on VLSI Circuits - Digest of Technical Papers*, 2011, p. 216–217.

[23] M. Bucci, L. Germani, R. Luzzi, A. Trifiletti, e M. Varanonuovo, "A high-speed oscillator-based truly random number source for cryptographic applications on a smart card IC", *IEEE Trans. Comput.*, vol. 52, nº 4, p. 403–409, abr. 2003.

[24] N. Stefanou e S. R. Sonkusale, "High speed array of oscillator-based truly binary random number generators", in *2004 IEEE International Symposium on Circuits and Systems (IEEE Cat. No.04CH37512)*, 2004, vol. 1, p. I–505.

[25] Q. Tang, B. Kim, Y. Lao, K. K. Parhi, e C. H. Kim, "True Random Number Generator circuits based on single- and multi-phase beat frequency detection", in *Proceedings of the IEEE 2014 Custom Integrated Circuits Conference*, 2014, p. 1–4.

[26] S. N. Dhanuskodi, A. Vijayakumar, e S. Kundu, "A Chaotic Ring oscillator based Random Number Generator", in *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2014, p. 160–165.

[27] K. Yang, D. Fick, M. B. Henry, Y. Lee, D. Blaauw, e D. Sylvester, "16.3 A 23Mb/s 23pJ/b fully synthesized true-random-number generator in 28nm and 65nm CMOS", in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2014, p. 280–281.

[28] A. A. Abidi, "Phase Noise and Jitter in CMOS Ring Oscillators", *IEEE J. Solid-State Circuits*, vol. 41, nº 8, p. 1803–1816, ago. 2006.

[29] B. Mesgarzadeh e A. Alvandpour, "A study of injection locking in ring oscillators", in *2005 IEEE International Symposium on Circuits and Systems*, 2005, p. 5465-5468 Vol. 6.

[30] A. T. Markettos e S. W. Moore, "The Frequency Injection Attack on Ring-Oscillator-Based True Random Number Generators", in *Cryptographic Hardware and Embedded Systems - CHES 2009*, vol. 5747, C. Clavier e K. Gaj, Orgs. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, p. 317–331.

[31] T. Technologies, "Terasic - SoC Platform - Cyclone - DE10-Nano Kit". [Online]. Disponível em: https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=1046. [Acessado: 29-jun-2019].

[32] "Quartus II Web Edition v12.1". .

[33] "Notepad++ v7.7.1 - Current Version". [Online]. Disponível em: https://notepad-plus-plus.org/download/v7.7.1.html. [Acessado: 01-jul-2019].

[34] "Sublime Text - A sophisticated text editor for code, markup and prose". [Online]. Disponível em: https://www.sublimetext.com/. [Acessado: 01-jul-2019].

[35] "MATLAB - MathWorks - MATLAB & Simulink". [Online]. Disponível em: https://www.mathworks.com/products/matlab.html. [Acessado: 01-jul-2019].

[36] I. T. L. Computer Security Division, "NIST SP 800-22: Documentation and Software - Random Bit Generation | CSRC", *CSRC | NIST*, 24-maio-2016. [Online]. Disponível em: https://csrc.nist.gov/projects/random-bit-generation/documentation-and-software. [Acessado: 01-jul-2019].

[37] Q. Tang, B. Kim, Y. Lao, K. K. Parhi, e C. H. Kim, "True Random Number Generator circuits based on single- and multi-phase beat frequency detection", in *Proceedings of the IEEE 2014 Custom Integrated Circuits Conference*, 2014, p. 1–4.

[38] A. P. Johnson, R. S. Chakraborty, e D. Mukhopadyay, "An Improved DCM-Based Tunable True Random Number Generator for Xilinx FPGA", *IEEE Trans. Circuits Syst. II Express Briefs*, vol. 64, nº 4, p. 452–456, abr. 2017.

Apêndice A – Código formato ".m" para cálculo e escrita da LUT em VHDL e LUT completa

```
clear all;
clc;

%parametros
imax = 256;

for sum =1:imax
    j=1;
    for runs =1:imax
    piv=erfc(abs(runs-(2*imax* sum/imax * (1-
sum/imax)))/(2*sqrt(2*imax)*sum/imax * (1-sum/imax)));
        if piv > 0.01
        runs_pass_temp(j)=runs;
        j=j+1;
        end
    end
    Matrix(sum,1) = ceil(min(runs_pass_temp));
    Matrix(sum,2) = floor(max(runs_pass_temp));
    runs_pass_temp = 0;
end

fid = fopen('vhdl_gen_8bits_0p01.txt','wt');
j=1;
for i=1:size(Matrix,1)-1
b(j,1)= Matrix(i,1);
b(j+1,1)=Matrix(i,2);
j=j+2;
end;

sum = [1:1:imax];
binary_line1= dec2bin(sum(1,1:size(sum,2)-1));
binary_line2_3 = dec2bin(b);

j=1;
for i=1:size(Matrix,1)-1
       line1 = strcat('when',{' '},
'"',binary_line1(i,1:size(binary_line1,2)),'"',{' '},'=>');
       line2 = strcat({'    '},'lower_run',{' '},'<=',{' '}
,'"',binary_line2_3(j,:),'"',';');
       line3 = strcat({'    '},'higher_run',{' '},'<=',{' '}
,'"',binary_line2_3(j+1,:),'"',';');
        j=j+2;
        fprintf(fid,'%s',line1{1});
        fprintf(fid,'\n');
        fprintf(fid,'%s',line2{1});
        fprintf(fid,'\n');
        fprintf(fid,'%s',line3{1});
        fprintf(fid,'\n');
end
fclose(fid);
```

Tabela 6 - LUT salva na memória do processador

# de '1's	Vobs min.	Vobs max	# de '1's	Vobs min.	Vobs max	# de '1's	Vobs min.	Vobs max
1	2	2	**86**	96	132	**171**	96	131
2	4	4	**87**	97	133	**172**	95	131
3	5	6	**88**	97	134	**173**	95	130
4	7	9	**89**	98	134	**174**	94	129
5	9	11	**90**	98	135	**175**	93	128
6	10	13	**91**	99	136	**176**	93	127
7	12	15	**92**	99	136	**177**	92	126
8	14	17	**93**	100	137	**178**	92	125
9	15	20	**94**	100	138	**179**	91	125
10	17	22	**95**	101	138	**180**	90	124
11	18	24	**96**	101	139	**181**	89	123
12	20	26	**97**	102	139	**182**	89	122
13	21	28	**98**	102	140	**183**	88	121
14	23	30	**99**	102	140	**184**	87	120
15	24	32	**100**	103	141	**185**	87	119
16	26	34	**101**	103	141	**186**	86	118
17	27	36	**102**	103	142	**187**	85	117
18	29	38	**103**	104	142	**188**	84	115
19	30	40	**104**	104	143	**189**	84	114
20	31	42	**105**	104	143	**190**	83	113
21	33	44	**106**	105	144	**191**	82	112
22	34	46	**107**	105	144	**192**	81	111
23	36	48	**108**	105	144	**193**	80	110
24	37	50	**109**	106	145	**194**	79	109
25	38	52	**110**	106	145	**195**	78	107
26	40	54	**111**	106	145	**196**	78	106
27	41	56	**112**	106	146	**197**	77	105
28	42	57	**113**	106	146	**198**	76	104
29	44	59	**114**	107	146	**199**	75	102
30	45	61	**115**	107	147	**200**	74	101
31	46	63	**116**	107	147	**201**	73	100
32	47	65	**117**	107	147	**202**	72	98
33	49	66	**118**	107	147	**203**	71	97
34	50	68	**119**	107	147	**204**	70	96
35	51	70	**120**	107	148	**205**	69	94
36	52	71	**121**	108	148	**206**	68	93
37	54	73	**122**	108	148	**207**	67	91
38	55	75	**123**	108	148	**208**	66	90
39	56	76	**124**	108	148	**209**	65	89
40	57	78	**125**	108	148	**210**	64	87
41	58	79	**126**	108	148	**211**	63	86
42	59	81	**127**	108	148	**212**	62	84
43	61	83	**128**	108	148	**213**	61	83

44	62	84	**129**	108	148	**214**	59	81
45	63	86	**130**	108	148	**215**	58	79
46	64	87	**131**	108	148	**216**	57	78
47	65	89	**132**	108	148	**217**	56	76
48	66	90	**133**	108	148	**218**	55	75
49	67	91	**134**	108	148	**219**	54	73
50	68	93	**135**	108	148	**220**	52	71
51	69	94	**136**	107	148	**221**	51	70
52	70	96	**137**	107	147	**222**	50	68
53	71	97	**138**	107	147	**223**	49	66
54	72	98	**139**	107	147	**224**	47	65
55	73	100	**140**	107	147	**225**	46	63
56	74	101	**141**	107	147	**226**	45	61
57	75	102	**142**	107	146	**227**	44	59
58	76	104	**143**	106	146	**228**	42	57
59	77	105	**144**	106	146	**229**	41	56
60	78	106	**145**	106	145	**230**	40	54
61	78	107	**146**	106	145	**231**	38	52
62	79	109	**147**	106	145	**232**	37	50
63	80	110	**148**	105	144	**233**	36	48
64	81	111	**149**	105	144	**234**	34	46
65	82	112	**150**	105	144	**235**	33	44
66	83	113	**151**	104	143	**236**	31	42
67	84	114	**152**	104	143	**237**	30	40
68	84	115	**153**	104	142	**238**	29	38
69	85	117	**154**	103	142	**239**	27	36
70	86	118	**155**	103	141	**240**	26	34
71	87	119	**156**	103	141	**241**	24	32
72	87	120	**157**	102	140	**242**	23	30
73	88	121	**158**	102	140	**243**	21	28
74	89	122	**159**	102	139	**244**	20	26
75	89	123	**160**	101	139	**245**	18	24
76	90	124	**161**	101	138	**246**	17	22
77	91	125	**162**	100	138	**247**	15	20
78	92	125	**163**	100	137	**248**	14	17
79	92	126	**164**	99	136	**249**	12	15
80	93	127	**165**	99	136	**250**	10	13
81	93	128	**166**	98	135	**251**	9	11
82	94	129	**167**	98	134	**252**	7	9
83	95	130	**168**	97	134	**253**	5	6
84	95	131	**169**	97	133	**254**	4	4
85	96	131	**170**	96	132	**255**	2	2

Apêndice B – Código em linguagem "C" da rotina de leitura de dados.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <fcntl.h>
#include <signal.h>
#include <sys/mman.h>
#include "hps_0.h"
#define REG_BASE    0xff200000  //lw h2f bridge base address
#define REG_SPAN    0x00200000  //lw h2f bridge size

void *base;
FILE *fptr;
uint32_t *data_inv, *data_count, *data_address;
int fd, now, last, i, prevAddress, currentAddress;

void handler(int signo)
{
    *data_inv=0;
    munmap(base, REG_SPAN);
    close(fd);
    exit(0);
}

int main()
{
    fd = open("/dev/mem", O_RDWR|O_SYNC);
    if(fd<0)
    {
        printf("cant open memory. \n");
        return -1;
    }
    base = mmap(NULL, REG_SPAN, PROT_READ|PROT_WRITE, MAP_SHARED, fd,
REG_BASE);

    if(base==MAP_FAILED)
    {
        printf("can't map memory. \n");
        close(fd);
        return -1;
    }

    fptr = fopen("counter3.txt", "w");

    if (fptr == NULL)
    {
    printf("Can't create file\n");
    }

    data_inv = (uint32_t*)(base+READ_INV_BASE);
    data_count = (uint32_t*)(base + READ_COUNT_BASE);
    data_address = (uint32_t*)(base + MEM_ADDRESS_BASE);

    signal(SIGINT,handler);

     last = 0;
     prevAddress = 0;
     i = 1;

    while(1)
```

```
    {   //check for falling edge for sync purpose when Fsampling >> Fhp and
for a clear edge w/ no debounce
        now = *data_inv;
        currentAddress = *data_address;

            if (now != last)
            {
                if (now < last) //means falling edge
                {   i = i+1;
                    fprintf(fptr, "%i\n", *data_count);
                    fprintf(fptr, "%i\n", *data_address);
                }
            }
            if (i == 50000000)
            {printf("Finished: collected  140Mb file \n");
            break;}
            last = now;

            if (currentAddress != prevAddress)
                    {printf("%i\n", *data_count);
                     printf("%i\n", *data_address);
                     printf("%i\n", i);
                    }
            prevAddress = currentAddress;
    }
    return 0;
}
```