UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ENGENHARIA DE SÃO CARLOS

RAPHAEL TIN CARLINI KOHN

Ambiente de Testes Estatísticos Para Geradores de Números Aleatórios Aplicado em Softwares de Análise Numérica

São Carlos
2020

RAPHAEL TIN CARLINI KOHN

Ambiente de Testes Estatísticos Para Geradores de Números Aleatórios Aplicado em Softwares de Análise Numérica

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
2020

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

Kohn, Raphael Tin Carlini
K79a Ambiente de testes estatísticos para geradores de números aleatórios aplicado em softwares de análise numérica / Raphael Tin Carlini Kohn; orientador Maximiliam Luppe. São Carlos, 2020.

Monografia (Graduação em Engenharia Elétrica com ênfase em Eletrônica) -- Escola de Engenharia de São Carlos da Universidade de São Paulo, 2020.

1. ambiente de testes estatísticos. 2. gerador de números aelatórios. 3. testes randômicos. 4. oscilador em anel. 5. caos determinístico. 6. ltspice. 7. octave. 8. NIST. I. Título.

Eduardo Graziosi Silva - CRB - 8/8907

FOLHA DE APROVAÇÃO

Nome: Raphael Tin Carlini Kohn

Título: "Ambiente de testes estatísticos para geradores de números aleatórios aplicado em softwares de análise numérica"

Trabalho de Conclusão de Curso defendido e aprovado em 26/06/2020,

com NOTA 7,0 (sete, zero), pela Comissão Julgadora:

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

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

Prof. Dr. João Paulo Pereira do Carmo - SEL/EESC/USP

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

A minha namorada Alexia, por toda paciência, apoio e companheirismo demonstrados durante não apenas esse projeto, mas nos últimos anos que me trouxeram até aqui.

A meus pais e minha irmã, que sempre me incentivaram e acreditaram em mim.

A meus amigos e amigas que me acompanharam na trajetória da graduação.

Agradecimentos

Agradeço inicialmente Alexia, minha namorada e companheira de longa data, que esteve presente nos momentos bons, ruins e necessários.

A minha família, que me forneceu o amor, apoio, conforto e paciência.

Ao professor Maximiliam Luppe, que me deu a oportunidade de participar neste projeto e as condições para terminá-lo mesmo contra todas as adversidades.

Aos amigos que me acompanharam na jornada da graduação até aqui, em especial Victor Koiti, Gustavo, Carlos, Ítalo, Victor Jácomo, Leon, Valeria, Antônio, Melissa e Rodolfo.

Aos colegas da Calina Marketing Digital, local que amadureci muito nos últimos 3 anos.

“A verdadeira medida de um homem não é sua inteligência ou quão alto ele sobe neste sistema esquisito. Não, a verdadeira medida de um homem é esta: com que rapidez ele consegue responder às necessidades dos outros e quanto de si mesmo ele consegue dar.”

Philip K. Dick

Resumo

KOHN, R. T. C. **Ambiente de Testes Estatísticos Para Geradores de Números Aleatórios Aplicado em Softwares de Análise Numérica**. 2020. Dissertação – Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2020.

Com o aumento da demanda por sistemas de criptografia/proteção de dados, as pesquisas e projetos de novos geradores de números aleatórios, essenciais para esse tipo de aplicação, se tornam cada mais mais necessários. Como o objetivo desse tipo de gerador é criar uma onda com comportamento aleatório, o projeto acaba sendo mais empírico do que teórico, visto que a modelagem de um gerador necessita de validação de suas sequências utilizando testes estatísticos. Estes, por sua vez, exigem grandes quantidades de dados, inviabilizando o desenvolvimento de tal projeto sem um sistema de coleta de dados robusto. Naturalmente, isso acaba dificultando o avanço da pesquisa na área, visto que equipamentos de aquisição de dados costumam ser caros e de difícil acesso. Dessa forma, este trabalho teve dois objetivos: primeiro, desenvolver um ambiente de testes estatísticos que precise de nenhuma ou pouca adaptação para validar dados gerados por geradores de números aleatórios de qualquer fonte (testes de bancada, simulações em SPICE, etc); segundo, simular um circuito de gerador de números aleatórios do tipo *Asynchronous Linear Feedback Shift Register*, uma topologia baseada em geradores do tipo *Linear Feedback Shift Register* e implementada em tecnologia TSMC 180nm, validando-o no ambiente de testes estatístico. No fim, conseguimos desenvolver a simulação do circuito na topologia proposta e três testes estatísticos dentro do ambiente de testes, o que foi o suficiente para validarmos os dados provenientes da simulação e obtermos indícios de que a topologia testada consegue gerar sequências aleatórias. Entretanto, ainda existe espaço para aprofundar tanto a simulação do circuito quanto o algoritmo, complementando-o com mais testes.

Palavras-chave: Ambiente de testes estatísticos, gerador de números aleatórios, testes randômicos, oscilador em anel, caos determinístico, ltspice, octave, NIST.

Abstract

KOHN, R. T. C. **Statistical test suite for Random Number Generators Applied in numerical analysis software.** 2020. – Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2020.

With the increasing demand for encryption/data protection systems, research and projects for new random number generators, essential for this type of application, becomes increasingly needed. As the objective of this type of generator is to create a wave with random behavior, which means the process is much more empirical than theoretical, given that modeling the generator requires validating its output with statistical tests. These tests require large amounts of data, thus, it is not possible to develop such a project without a robust data collection system. Naturally, this ends up hampering the advancements of research in the area, since data capture equipment is often expensive and difficult to access. Thus, this work had two objectives: first, to develop a statistical test suite that can be easy adapted for any random number generator source (bank tests, simulations in SPICE, etc.); second, to simulate a random number generator circuit with the *Asynchronous Linear Feedback Shift Register* configuration, a topology based on *Linear Feedback Shift Register* random generators and implemented in TSMC 180nm technology, using the statistical test suite of the first objective to validate the circuit. In the end, we managed to develop the simulation of the proposed topology and three statistical tests in the algorithm, which was enough to obtain indications that the tested topology can generate random sequences. However, there is still space to deepen the simulation of the circuit and to complement the algorithm with more statistical tests.

Keywords: Statistical test suite, random number generator, randomness tests, ring oscillator, deterministic chaos, ltspice, octave, NIST.

Lista de Ilustrações

Lista de Abreviaturas e siglas

ALFSR	Asynchronous Linear Feedback Shift-Register.
CMOS	Complementary Metal Oxide Semiconductor
FPGA	Field Programmable Gate Array
LFSR	Linear Feedback Shift-Register.
PRNG	Pseudo Random Number Generator.
RNG	Random Number Generator.
SPICE	Simulation Program with Integrated Circuits Emphasis.
TRNG	True Random Number Generator.
TERO	Transient Effect Ring Oscillator
TSMC	Taiwan Semiconductor Manufacturing Company

Lista de Tabelas

Sumário

1 Introdução

1.1 Motivação

O desenvolvimento e aplicação de geradores de números aleatórios tem crescido nas últimas décadas, motivado principalmente pela preocupação de se proteger o grande volume de dados gerados e trocados hoje em dia por equipamentos tecnológicos [1].

Entretanto, o processo de desenvolvimento desses geradores tem um foco maior nos dados empíricos do que na teoria, muito por conta da natureza dos sinais gerados e de seu objetivo de torná-los imprevisíveis.

Quem tiver o interesse de desenvolver este tipo de gerador necessitará validar seus dados antes de poder validar/melhorar sua modelagem, o que pode se mostrar um desafio visto que os testes de validação exigem um grande número de dados, necessitando de um sistema de aquisição robusto, o que não é acessível a todos.

Esta dificuldade de validação empírica motivou este trabalho para viabilizar uma maneira de pesquisadores desenvolverem projetos sobre o tema utilizando softwares gratuitos, com uma interface amigável e facilmente adaptável para diversas fontes de dados.

Também surgiu a necessidade de se validar uma nova topologia de geradores de números aleatórios que será aplicado em um outro trabalho. Desta forma, este projeto irá desenvolver tanto uma interface de testes estatísticos para números aleatórios quanto uma simulação de um gerador de números aleatórios, além de avaliar essa simulação utilizando a interface de testes.

1.2 Objetivos

Com a motivação de facilitar o acesso a testes estatísticos para geradores aleatórios e validar a topologia que será detalhada mais adiante, foram traçados dois objetivos principais para o projeto:

1. Criar um programa de fácil utilização que consiga realizar os testes de aleatoriedade da interface de testes do Institucional Nacional de Padrões e Tecnologias dos EUA (NIST).
2. Simular um circuito de gerador de números aleatórios e validar seus dados utilizando o programa do primeiro objetivo.

Para explicar o planejamento e execução do projeto para alcançar esses dois objetivos, dividimos a monografia em outros quatro capítulos: Revisão Bibliográfica, onde explicaremos a parte teórica do trabalho e introduziremos conceitos importantes para entender o funcionamento dos produtos finais do projeto; Materiais e Métodos, capítulo no qual entraremos mais a fundo sobre quais foram as ideias e ferramentas utilizadas para atingir cada um dos objetivos propostos; Resultados e Discussões, momento em que apresentaremos os resultados finais tanto o programa do primeiro objetivo quanto a simulação do circuito do segundo objetivo, além de descrever validações e análises feitas em cima de ambos; e, por fim, a Conclusão, onde comparamos os resultados obtidos com os objetivos que estabelecemos aqui na Introdução, além de propor próximos passos para o projeto.

2 Revisão Bibliográfica

Nesta seção apresentamos conceitos sobre sequências de números aleatórios que serão cruciais para o entendimento dos testes feitos pelo ambiente de testes estatísticos.

2.1 Conceitos de Aleatoriedade

Uma sequência de eventos é chamada aleatória se cada um de seus elementos independe dos outros (passados ou futuros).

Um ótimo exemplo prático é o lançamento de uma moeda honesta (50% de chance de cara ou coroa), onde cada lançamento independe de outros e, dado um longo tempo, a sequência sempre tenderá a ter uma proporção igual de caras e coroas. Entretanto, é curioso perceber como dependendo do tamanho das amostras até mesmo uma sequência considerada ideal pode ser confundida como não aleatória: a moeda pode gerar tanto a sequência 1 = “1001111010” quanto 2 = “1111111111”.

Essa confusão só ocorre quando não diferenciamos uma “geração aleatória” de um “arranjo aleatório” [2]. Se olharmos apenas para a capacidade de “geração aleatória” do lançamento da moeda (o gerador que estamos avaliando no caso), estaremos avaliando apenas a chance individual de cada lançamento, o que faz com que concluamos que as duas sequências têm a mesma chance de ocorrer. Entretanto, se olharmos para a capacidade do gerador de criar “arranjos aleatórios”, estaremos olhando para as chances das sequências ocorrerem, com a sequência 1 (“1001111010”) tendo uma chance de ocorrer maior por se aproximar mais da distribuição de 50% entre 1s e 0s característica do gerador avaliado.

Essa percepção é extremamente importante para os testes de aleatoriedade que nos propomos a implementar, visto que a quantidade de dados das sequências simuladas é o fator mais limitante em todo o projeto e, ao mesmo tempo, o que garante que os testes sejam válidos.

2.2 Imprevisibilidade

Outra característica essencial para uma sequência de elementos ser considerada aleatória é que nenhum evento consiga ser previsto com outros passados. Essa característica é chamada de “imprevisibilidade futura” [1].

Aqui, também é interessante trazer outro exemplo de aparente contradição dos conceitos com o paradoxo de pi [2]: a sequência do número pi: 314159265358979323846… não só aparenta ser aleatória como também passa nos testes de aleatoriedade (como veremos em sua forma binária na seção 5.2). Entretanto, existem fórmulas matemáticas que conseguem prever o valor número de pi - ou seja, sua sequência pode ser classificada como aleatória, mas também existem métodos matemáticos que prevêem sua forma.

Isso salienta novamente a importância de considerar o ponto de vista que se está tendo nas análises: uma conclusão parte da visão computacional, enquanto a outra, da estatística. Essas “previsões” são, na verdade, aproximações de expansões decimais obtidas empiricamente e que convergem para pi, onde cada número de casas decimais diferentes necessita de uma fórmula diferente. Logo, do ponto de vista estatístico, ele não é previsível, visto que não existe nenhuma fórmula para calcular individualmente cada um de seus elementos.

2.3 Testes de aleatoriedade

Um teste de aleatoriedade é, no fundo, um teste estatístico que busca validar uma “hipótese nula” (*H0*). No caso desta aplicação, tal hipótese trata se a sequência testada é aleatória. Acompanhada dela, temos a “hipótese alternativa” (*Ha*), uma hipótese complementar por assim dizer, que diz que a sequência testada não é aleatória.

Com isso, temos duas possibilidades de resultados no teste: ou *H0* é verdadeiro (e a onda é aleatória) ou *Ha* é verdadeiro (e a onda não é aleatória) [1]. Entretanto, existe a possibilidade do teste falhar e chegarmos a um falso negativo ou positivo.

Tabela 1. Tabela verdade sobre os testes.

Resultado	Conclusão	
	Aceitar *H0*	Rejeitar *H0* (Aceitar *Ha*)
H0 Verdadeiro	Sem erro	Falso Negativo
Ha Verdadeiro	Falso Positivo	Sem erro

Fonte: Traduzido da página 13 de [1].

Seja α a probabilidade de um Falso Negativo ocorrer, β o de um Falso Positivo e *n* o número de bits na sequência.

α é conhecido como o nível de significância do teste por ser um número fixo, enquanto β é variável, já que uma sequência não aleatória pode se apresentar aleatória de diversas maneiras.

A relação entre α, β e *n* nos possibilita que, com apenas o valor de dois dos três conseguimos encontrar o terceiro valor. Com isso, fixa-se os valores de *n* e α de tal maneira que β seja o menor possível, visto que não conseguimos controlá-lo [1].

Na prática, isso estabelece uma proporção entre *n* e α e dita o mínimo de dados necessários para o teste em questão dado determinado nível de significância. Cada teste estatístico fornece no fim o *valor de P*, um número de zero a um que indica quão forte é a evidência de que a aquela sequência é aleatória (sendo zero não aleatória e um máxima aleatoriedade) [1].

O *valor de P* é comparado com o de α e, caso seja maior, a hipótese *H0* não é rejeitada com uma confiabilidade de $(1 - \alpha)$ e o próximo teste é feito. É este o motivo de α ser chamado de nível de significância, pois ele determina a precisão do processo. Um valor comum a ser adotado é o de "$\alpha = 0.01$", que significa que a valor de confiança para que a sequência seja aleatória é de 99% (apenas um a cada cem testes vão resultar em um falso negativo), deixando os testes precisos o suficiente. Em cada teste chega-se ao *valor de P* ao comparar seus resultados à uma distribuição de referência, normalizando o valor entre zero e um - ou seja, para esse valor comum de $\alpha = 0.01$, *valor de P* precisa ser menor que 0.01 para *H0* ser rejeitada[1].

As especificidades dos testes implementados no trabalho serão detalhadas na secção 3.2.1.2.

2.4 Caos Determinístico

Sistemas determinísticos permitem prever qualquer ponto no tempo, dado que se saiba suas condições iniciais [8]. Entretanto, alguns desses sistemas apresentam uma grande sensibilidade para variações nos dados iniciais, dando a impressão de que sejam aleatórios, mesmo que isso seja causado apenas por uma falta de precisão na medição dos parâmetros. Esses casos são definidos como Sistemas Caóticos Determinísticos e, apesar de não serem aleatórios, ainda assim são considerados imprevisíveis já que uma pequena variação nas condições iniciais acarreta em resultados finais ao longo do tempo muito distintos.

2.5 Geradores de Números Aleatórios

Todos os conceitos explicados até agora são aplicados na geração de números aleatórios: sistemas que, como o nome diz, servem para gerar uma sequência de números que seja aleatória. Sua demanda vem principalmente de aplicações envolvendo criptografia e segurança de dados, exigindo nesses casos que os geradores sejam robustos o suficiente para aguentar tentativas de invasões aos dados que estiverem auxiliando na codificação.

2.5.1 Contextualização

Generalizando, existem dois tipos de geradores: os aleatórios (TRNG - *True Random Number Generator*) e os pseudo-aleatórios (PRNG - *Pseudo Random Number Generator*). Enquanto os primeiros são obtidos primariamente de fontes de entropia, como ruído térmico de um transistor ou efeito fotoelétrico, os últimos são compostos de fontes determinísticas, como algoritmos [3][4][5].

Entretanto, analisando as topologias desenvolvidas nos últimos anos, percebe-se que nenhum desses dois tipos de geradores atendem às necessidades: os TRNGs por não serem robustos [3] e os PRNGs por não serem aleatórios o suficiente. A solução foi juntar os dois tipos em um só: PRNGs que utilizam fontes de entropia como sementes para a geração de seus números, resultando em sistemas caóticos, imprevisíveis por definição e robustos o suficiente para ataques e/ou variações de temperatura, tensão, etc.

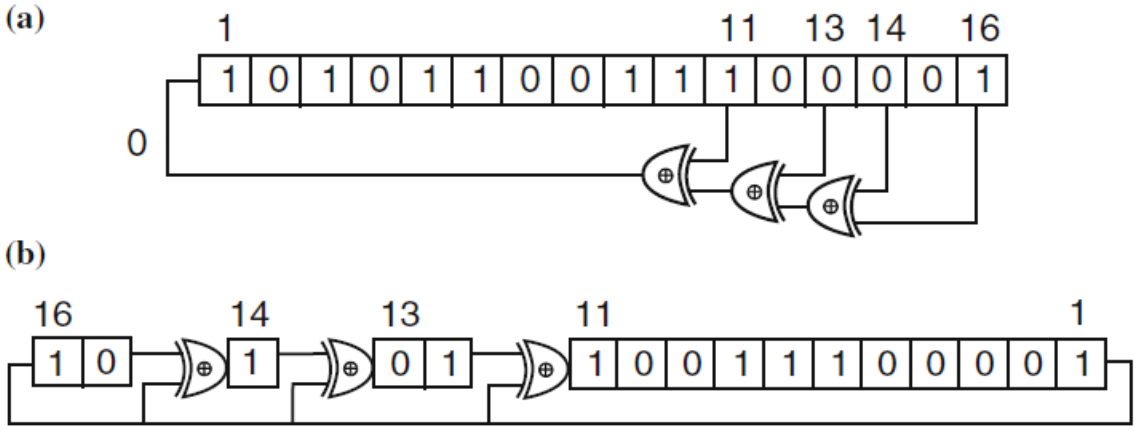
2.5.2 Topologias

Nesta secção, serão dados exemplos de topologias dos sistemas caóticos descritos anteriormente. Os exemplos apresentam uma característica em comum: todos se utilizam do fato de serem implementados com semicondutores (seja dentro de FPGAs - *Field Programmable Gate Array* ou diretamente com transistores CMOS - *Complementary Metal Oxide Semiconductor*), obtendo sua fonte de entropia do ruído interno de seus componentes.

Um exemplo de topologia PRNG são os LFSR (*Linear Feedback Shift-Register*), circuitos que se baseiam em osciladores em anel compostos por flip-flops para gerar sua pseudo aleatoriedade. Na figura 1 podemos encontrar duas variações: a variação (a), chamada de Fibonacci; e a variação (b), chamada Galois [15]. A Característica que diferencia um do outro é a posição das portas XOR, pois na Fibonacci um das entradas das portas vem de posições centrais do circuito enquanto na Galois todas as XOR recebe a

saída do circuito. Entretanto, o funcionamento de ambos é praticamente o mesmo: a cada iteração, a saída das XOR são atualizadas, “empurrando” a sequência de bits e atualizando as próprias entradas das XORs - por exemplo, na variação (a) da figura 1, a próxima iteração fará com que as posições 1, 11, 13, 14 e 16 fiquem com os valores 0, 1, 0, 0 e 0 respectivamente. No caso, ambas variações tem 16 bits, o que significa que existem $2^{16}-1$ combinações entre as posições de bits possíveis, fazendo com o que o gerador repita a sequência das combinações eventualmente, algo que pode ser explorado para invasões. Além disso, quando se implementa LFSRs, costuma-se utilizar flip-flops para armazenar e propagar os bits de cada posição, o que significa que é necessário um clock controlando as iterações do circuito.

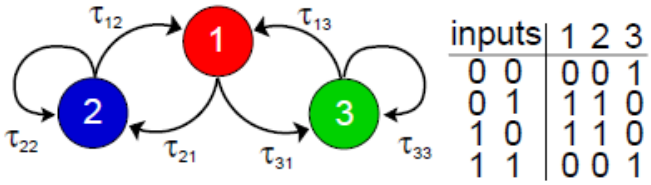
Figura 1. Exemplos de circuitos LFSR



Fonte: Página 60 de [15].

Em 2009, Zhang propôs um circuito CMOS composto por duas portas lógicas XOR e uma XNOR, resultando em uma máquina de estados com comportamento descrita na Figura 2 [4]. Essa configuração apresenta características de um sistema caótico e funciona assincronamente, não dependendo de um clock e, consequentemente, tornando o circuito menos suscetível a ataques. Segundo [11], as redes propostas até então ou eram síncronas ou lentas demais para aplicações, destacando o circuito de Zhang.

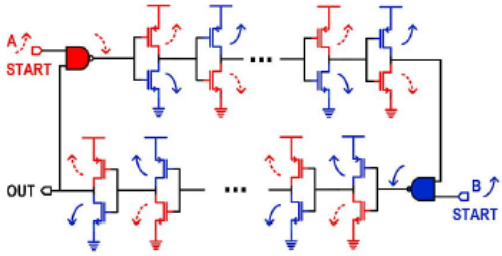
Figura 2. Máquina de estados resultante



Fonte: Imagem retirada da pág. 4 de [4].

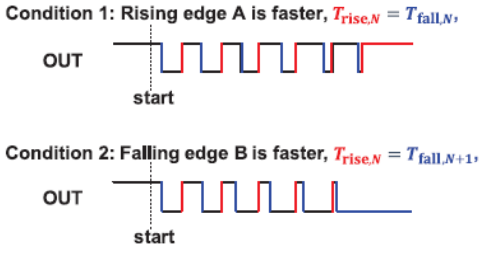
Outro exemplo de topologia é o criado por Yang [9], onde se utilizou duas linhas de atraso composta por inversores (diferente dos LFSRs convencionais que utilizam flip-flops), ligadas por portas NAND, conforme Figura 3. Essas duas linhas formam um oscilador em anel, sendo controladas por meio de trem de pulsos nas entradas A e B das NANDs. A diferença de fase entre essas duas entradas determina a saída em Out, onde A “mais rápido” que B faz com que a saída comece a tender para “1”, enquanto caso B seja “mais rápido” do que A, a saída tende para “0” (Figura 4). Esse circuito costuma ser conhecido como TERO - *Transient Effect Ring Oscilator.*

Figura 3. TERO TRNG



Fonte: Imagem retirada da pág. 2 de [9].

Figura 4. “Corrida” entre A e B



Fonte: Imagem retirada da pág. 2 de [9].

Esses foram apenas alguns exemplos de topologias diferentes para geradores de números aleatórios, existindo diversas outras que não foram exploradas aqui. Entretanto, as topologias descritas, somadas aos outros conceitos introduzidos no restante do capítulo, são o suficiente para o entendimento do projeto final, seus métodos e resultados.

Nos próximos capítulos, detalharemos mais sobre como esses tópicos explicados aqui se encaixam com o projeto, além de explicar o que exatamente foi feito, quais foram os resultados obtidos e como isso se compara com as expectativas iniciais do projeto.

3 Materiais e Métodos

O projeto se dividiu em duas partes: a simulação do circuito de gerador de número aleatório em LTSpice e a criação do algoritmo de checagem de aleatoriedade conforme padrão NIST.

3.1 Simulação

Para a simulação do circuito, foi necessário definir tanto qual ferramenta seria utilizada além de quais componentes e topologia comporiam o circuito final.

3.1.1 LTSpice

Foi escolhido simular o circuito em Spice, que traduzindo para o português significa *Programa de Simulação com Ênfase em Circuitos Integrados,* utilizando o programa LTSpice. Ele foi escolhido tanto por ser um software gratuito quanto pela familiaridade pelo seu uso em disciplinas da graduação.

Esse tipo de programa costuma ter quatro tipos de simulação:

1. Análise DC
2. Análise AC
3. Análise da polarização
4. Análise Transiente

No caso, estamos interessados apenas na análise transiente, visto que queremos analisar o circuito no domínio do tempo. Nela, o programa basicamente computa qual o comportamento do circuito assim que ele é ligado (tendo a possibilidade de descrever as condições de contorno, caso necessário). Para circuitos não-lineares (como o caso deste projeto), ele utilizar o Método de Newton–Raphson [14] para calcular as iterações de cada nódulo do sistema.

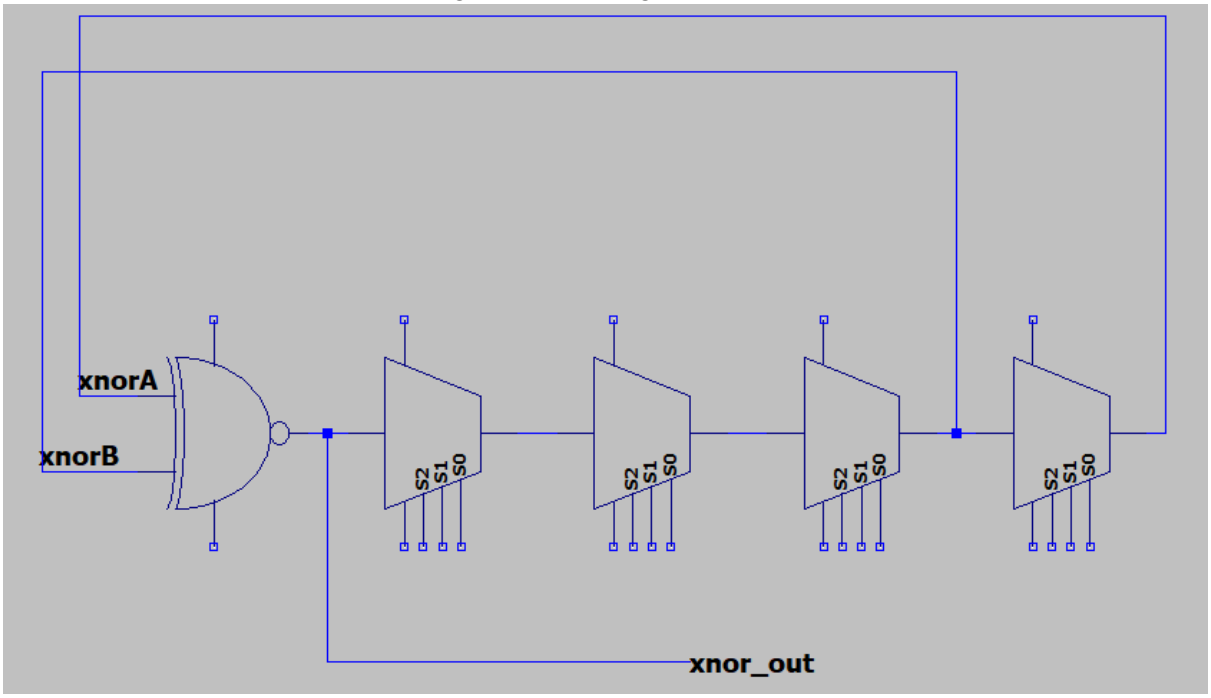
Um ponto importante na simulação é o passo máximo que ela pode dar (max step) - essa variável é uma das que mais impacta tanto na precisão da simulação, quanto no tempo dela. Isso ocorre pois o método LTSpice é otimizado para, caso os dados gerados em instantes sucessivos tenham pequenas variações, a distância entre os instantes analisados seja gradativamente incrementada, permitindo que a simulação seja realizada num menor tempo - entretanto, isso pode acarretar erros grosseiros de computação, principalmente em circuitos com instabilidades. Por conta disso, configurar um max step

condizente com o período dos dados garante que a simulação computará todos os pontos necessários, mesmo que isso aumente o tempo total.

3.1.2 Topologia ALFSR

A topologia escolhida para o projeto é um LFSR Assíncrono (ALFSR - *Asynchronous Linear Feedback Shift Register*), implementado em tecnologia CMOS TSMC 180nm (figura 5). Nela, os flip-flops originais dos LFSR convencionais exemplificados na seção 2.5.2 são substituídos por “células de atraso” com três bits de seleção (S0, S1 e S2) que permitem escolher o tempo de atraso - isso é possível pois dentro de cada uma dessas células existem sete buffers que resultam em oito tempos de atrasos diferentes (sem buffer, um buffer, dois buffers, …, sete buffers), onde três colunas de MUX controladas pelos bits de seleção tornam possível essa escolha do tempo de atraso. Os Buffers são compostos por dois inversores em série - os subcircuitos das células de atraso serão detalhados mais adiante.

Figura 5. Topologia ALFSR



Fonte: Própria

As células de atraso foram baseadas no trabalho de Yang [09], o circuito do tipo TERO apresentado na secção 2.5.2. A novidade introduzida pelo ALFSR é a presença de seletores para escolher por quantas linhas de atraso o sinal passará dentro da célula, resultando em $2^8 - 1$ configurações para o circuito total de quatro células. Isso possibilita o

desenvolvimento em projetos futuros de um circuito de proteção de ataques que consegue mudar a configuração escolhida assim que percebe que a saída não é mais aleatória.

3.2 Ferramentas para Classificação Dados

Para verificar quais configurações do circuito são aleatórias, desenvolveu-se um código na ferramenta Octave para avaliação da aleatoriedade de suas saídas, utilizando o algoritmo do NIST. A escolha do Octave foi feita pois ele permite preparar os dados para a simulação de uma forma mais amigável e prática do que o Test Suite do NIST em C, o qual exige que seus dados entrem diretamente como binários - desta forma, os testes estatísticos continuam sendo funções independentes do resto do algoritmo como no Test Suite do NIST, mas com a diferença que o código criado no Octave permite adaptar os dados recebidos independente da fonte e com inputs e adaptações necessárias feitas do usuário sejam mínimas.

Para garantir a confiabilidade do algoritmo, foram utilizados os métodos de validação oferecidos pelo próprio NIST e que serão apresentados mais adiante.

O algoritmo foi desenvolvido em duas partes distintas:

1. Preparação dos dados em formato de texto do LTSpice
2. Testes Estatísticos validados pelo NIST

Dessa forma os testes estatísticos não ficam presos a preparação de dados específica para LTSpice, permitindo que eles sejam reutilizados independente da fonte que será testada. O único trabalho, neste caso, seria montar um código que organize os dados no padrão dos testes.

3.2.1 Métodos de Preparação dos Dados

O objetivo desta parte do código é receber os dados da fonte e prepará-los para o padrão que os testes foram programados. No caso deste trabalho, foi desenvolvido apenas um programa para preparação de dados gerados pelo LTSpice, mas o código pode ser facilmente adaptado em trabalhos futuros para qualquer fonte, como outros simuladores SPICE ou dados retirados de circuitos em bancada.

Na figura 6 é possível encontrar as primeiras linhas de um arquivo de texto que é gerado pelo circuito final simulado no LTSpice (no apêndice C existe uma versão mais completa). Neste caso, foram incluídas oito variáveis do circuito, totalizando nove colunas visto que a primeira coluna é o tempo de simulação de cada um dos pontos salvos. Vale destacar que as três colunas essenciais para o programa funcionar são V(clk_coleta),

V(rstn) e V(rnd0_clk), que representam o clock do circuito de coleta, o reset de troca de configuração e a saída final do circuito pós etapa de coleta, respectivamente.

Figura 6. Exemplo arquivo final gerado pela simulação de LTSpice

time	V(bit1)	V(bit2)	V(bit3)	V(clk_coleta)	V(rnd0)	V(rnd0_clk)	V(rstn)	V(xnor_out)			
0.000000000000000e+000	0.000000e+000		0.000000e+000		0.000000e+000	0.000000e+000	0.000000e+000	5.794203e-009	0.000000e+000	0.000000e+000	6.081074e-009
5.851428592654594e-013	0.000000e+000		0.000000e+000		0.000000e+000	0.000000e+000	4.681143e-003	3.833081e-005	0.000000e+000	2.106514e-002	4.403265e-004
1.170285718530919e-012	0.000000e+000		0.000000e+000		0.000000e+000	0.000000e+000	9.362286e-003	6.723530e-005	0.000000e+000	4.213029e-002	8.366953e-004
1.755428577796378e-012	0.000000e+000		0.000000e+000		0.000000e+000	0.000000e+000	1.404343e-002	8.671926e-005	0.000000e+000	6.319543e-002	1.189112e-003
2.340571437061837e-012	0.000000e+000		0.000000e+000		0.000000e+000	0.000000e+000	1.872457e-002	9.678270e-005	0.000000e+000	8.426057e-002	1.497578e-003

Fonte: Própria.

A preparação em si conta com as seguintes etapas:

1. Inputs Iniciais
    a. Aqui o usuário indica quais as colunas que incluem os dados a serem avaliados, os dados de clock, os dados de tempo, quantas configurações do circuito estão inclusas (representada pelo número de bits a disposição), a tolerância dos testes estatísticos e por fim, quantos testes serão rodados.
    b. O código também prepara parte dos vetores que serão necessários desde o começo do código.
2. Vetor Clock
    a. Nesta parte o código detecta em quais linhas houve subida de borda do vetor de clock, preenchendo um vetor que será utilizado na próxima etapa para filtrar quais linhas dos dados serão de fato utilizados na análise.
    b. Esta etapa é fundamental, visto que os dados do LTSpice não tem uma cadência temporal estabelecida, podendo variar de 0 até o valor de “max time step” da simulação. Logo, mesmo uma onda que já tenha passado por um processo de coleta dentro da simulação precisa passar por este processo para que o algoritmo consiga avaliar corretamente a onda.
    c. O código faz uma checagem para que o vetor de clock não tenha nenhum valor preenchido durante a troca de configurações (reset diferente de 1.8V). Desta forma, isso não precisa ser repetido nas próximas etapas, visto que as únicas linhas válidas serão aquelas nas quais clock = 1.
3. Preencher Matrizes com os dados a serem avaliados.
    a. Aqui, o código preenche matrizes de três dimensões com todos os dados que serão utilizados dentro dos testes estatísticos.

i. Linhas = Dados
ii. Coluna = Configuração
iii. Matriz = Onda/Fonte

b. Essa organização permite que se avalie num mesmo arquivo TXT de LTSpice múltiplas configurações de múltiplas saídas/pontos do circuito simulado.
c. OBS: Um ponto de melhora detectado para o código é conseguir avaliar simulações com ".step param" sem precisar editar previamente os arquivos. Atualmente ele não consegue ler totalmente os TXTs gerados nesse tipo de simulação pois o LTSpice inclui uma linha de texto entre as "runs".

3.2.2 Testes NIST

Até o momento da entrega do projeto, foram criados os três primeiros testes do NIST. Como explicado na seção 2.1.1, o erro de Tipo 2 é minimizado São eles:

1. Teste Frequência (Monobit).

O objetivo é avaliar se a proporção entre 1s e 0s é aproximadamente a mesma que uma sequência binária realmente aleatória: ou seja, metade "1" e metade "0". No caso, isso é feito utilizando como referência estatística a distribuição meia normal. Na prática, isso se traduz em realizar a soma *Sn* dos *n* elementos da sequência binária $\varepsilon=\varepsilon_1,\varepsilon_2,\varepsilon_3\ldots\varepsilon_n$ analisada no teste:

$$Sn = \sum_{i=1}^{n} 2 * \varepsilon_i - 1$$

O segundo passo é dividir *Sn* por $\sqrt{n}$ e então calcular a *erfc* (função complementar do erro) do resultado obtido dividido pela raiz de 2. O resultado final é o *valor de P* e todo esse processo pode ser compactado na seguinte equação:

$$valor\ de\ P\ =\ erfc\left(\frac{|Sn|}{\sqrt{2n}}\right)$$

O *valor de P* então é comparado com *a*, sendo a hipótese nula não rejeitada caso ele seja maior. O algoritmo também armazena em uma matriz de Resultados o valor “1” caso H0 não tenha sido rejeitada e “0” caso tenha.

O número de bits mínimo recomendado pelo NIST é n= 100.

2. Teste Frequência em Bloco.

O objetivo deste teste é descobrir se a proporção entre 1s e 0s próximo de ½ encontrada no primeiro teste também é válida em repartições menores da onda e não apenas no total. Um exemplo da necessidade deste teste é a sequência “11110000” que passaria no primeiro, mas seria recusada no segundo, conforme será explicado adiante. Ele usa como referência a distribuição Qui-Quadrado (χ^2).

Sejam então os *n* bits da sequência ε divididos em *N* blocos de dados com *M* bits cada. O primeiro passo é computar a proporção π_i de 1s dentro de cada um dos *N* blocos, seguindo a fórmula:

$$\pi_i = \frac{\sum_{j=1}^{M} \varepsilon_{(i-1)*M+j}}{M}$$, onde 1<= *i* <= *N*

Aqui, *i* representa qual dos *N* blocos estamos calculando a proporção. Essa equação basicamente faz a soma de todos os elementos do bloco *N*, que resulta no número total de 1s do bloco, e a divide pela quantidade *M* de bits dentro do bloco. A posição do elemento a ser calculado é dada por $"(i-1)*M+j"$, ou seja, para o primeiro bloco (*i*=1) a equação somará os elementos entre as posições $"\varepsilon_1"$ e $"\varepsilon_M"$, para o segundo bloco (*i*=2), a soma será com os elementos entre as posições $"\varepsilon_{(M+1)}"$ e $"\varepsilon_{2*M}"$ e assim por diante - desta forma, a fórmula consegue valer para qualquer bloco *N* da sequência ε ,, garantindo que não haverá sobreposição de elementos entre os blocos.

Em seguida calcula-se a função de Qui-Quadrado:

$$\chi^2 = 4 * M \sum_{i=1}^{N} (\pi_i - \tfrac{1}{2})^2$$

E então computa-se o *Valor de P* (*Pvalue*) utilizando a função gamma incompleta (*igamc*) da seguinte forma:

$$Pvalue = igamc(\tfrac{N}{2}, \tfrac{\chi^2}{2})$$

Caso *Pvalue*> α, a hipótese nula não é rejeitada.

Os valores mínimos recomendados são $n \geq 100 \geq MN$; $M \geq 20$; $N < 100$

É por isso que a sequência citada anteriormente, “11110000”, não passa neste segundo teste: se considerarmos *N*=2 apenas para demonstrarmos este exemplo, teríamos os blocos de dados “1111” e “0000”. Ambos claramente não tem uma proporção de “1s” próxima de ½, o que faz a sequência original falhar no teste.

3. Teste das Corridas.

A função deste teste é entender se existe alguma sequência longa demais de algum dos bits, indicando um comportamento “constante” da onda, sem variação para gerar imprevisibilidade suficiente para ser considerada aleatória. O resultado é o mesmo para corridas de 1s ou de 0s, então escolhemos fazer corridas de 1s.

O primeiro passo é calcular a proporção π de 1s da sequência ε inteira, parecido com o que foi feito no teste anterior:

$$\pi = \frac{\sum_{j=1}^{n} \varepsilon_j}{n}$$

O segundo é computar o teste estatístico Vn:

$$V_n = \sum_{k=1}^{n-1} r(k) + 1 \text{ , sendo}$$

$\varepsilon_{(k)} = \varepsilon_{(k+1)} \Rightarrow r(k)$ =0, caso contrário, $r(k)$ =1

O *Pvalue* é calculado então desta forma:

$$Pvalue = erfc\left(\frac{|V_n - 2n\pi(1-\pi)|}{2\sqrt{2n}\pi(1-\pi)}\right)$$

Sendo *Pvalue*> α, então a hipótese nula não é rejeitada.

O valor mínimo recomendado é $n \geq 100$.

Ainda existem outros testes nos documentos de referência do NIST, mas este trabalho tratará apenas desses três que foram apresentados. Independente, o ambiente de testes foi desenvolvido de tal forma que a adição de novos testes é simples, logo não será um problema complementar essa parte do projeto no futuro.

Neste capítulo apresentamos as duas principais ferramentas em que desenvolvemos o projeto, LTSpice e Octave. Ambos os programas são gratuitos e de fácil acesso/utilização, servindo para a proposta do trabalho de facilitar a validação de geradores de números aleatórios para outras pesquisas. Além disso, também falamos mais a fundo sobre a topologia ALFSR do circuito que simulamos, mostrando as novas propostas que ele traz em relação a topologias existentes.

Na próxima parte da monografia, vamos explorar de uma forma mais prática como ocorreu a implementação da simulação e os resultados obtidos ao analisar os dados obtidos da simulação dentro do ambiente de testes estatísticos. Também trazemos a validação do ambiente de testes em si, utilizando valores referência fornecidos pelo NIST.

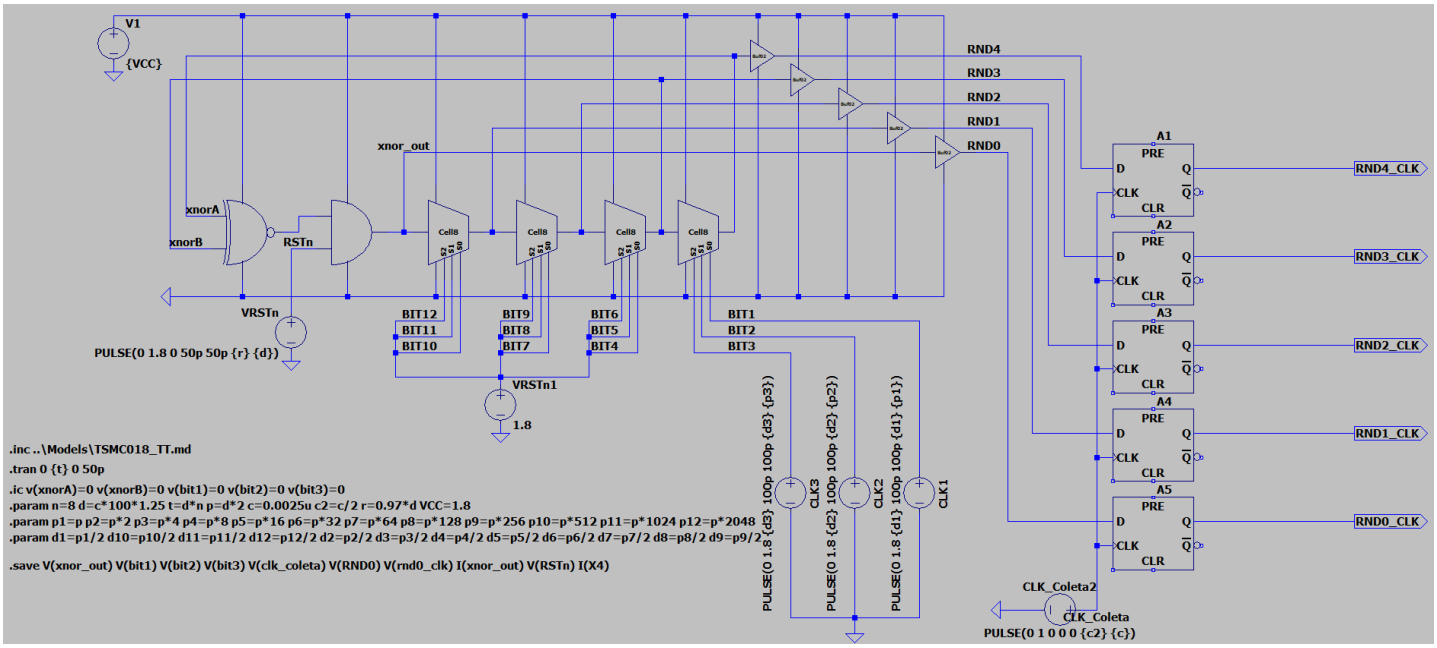
Resultados e Discussões

Nesta seção serão detalhadas tanto a simulação do ALFSR dentro do LTSpice, quanto o algoritmo criado dentro do Octave para avaliar os resultados da simulação.

4.1 Simulação ALFSR LTSpice

O circuito (Figura 7) da simulação pode ser dividido em três partes principais: o seletor de configuração; o circuito de coleta e o gerador de números aleatórios ALFSR. Além disso, existe uma parte na simulação que inclui os Inputs e diretrizes de Spice utilizados na simulação. Começaremos explicando os inputs primeiramente e depois descreveremos o comportamento de cada uma das partes principais do circuito.

Figura 7. Circuito completo do gerador de números aleatórios



Fonte: própria

4.1.1 Inputs e diretrizes de Spice

Os parâmetros da simulação foram escolhidos da seguinte forma:

- *c* = 0,0025 µs - período do clock de coleta de dados.
- *n* = 8 - número de configurações testadas

- *d* = c*125 - o tempo de onda ativa das ondas do seletor de configuração. Calculado de tal forma que garanta pelo menos 125 pontos para cada configuração.
- *p* = d*2 - o período das ondas de seleção;
- *c2* = $\frac{c}{2}$ - tempo ativo (duty cicle) de "c"
- *r* = 0,97*d variável para garantir que o tempo de reset seja apenas 3% de d, garantindo que o reset ocorre com o mínimo de perda possível
- *t* = *d*n* - o tempo que a simulação vai rodar. Garante que toda simulação vai coletar a mesma quantidade de dados de cada configuração.
- Para configurar as ondas de seleção de bit foram criados os parâmetros p1...p12 e *d1*...*d12*, todos dependentes de *p*, seguindo a fórmula "2^n*100**n*". Na prática, cada bit tem a metade da frequência do anterior. Juntos, formam um contador binário de 12 bits.
- O comando ".ic" garante que partes sensíveis do circuito não tenham um valor inicial que possam alterar a resposta do circuito sem aviso, como as entradas da porta XNOR e os bits de seleção, já que caso as duas portas da XNOR fiquem "1", o circuito todo "trava" em "1" e para de oscilar.
- O comando ".param" foi utilizado para declarar as variáveis, incluindo qualquer dependência que tenham com outras.
- O comando "save" foi utilizado para acelerar a simulação, salvando apenas os valores indicados e que serão utilizados.

Todos esses valores foram implementados no código de simulação nas diretrizes do spice, conforme a figura 8.

Figura 8. Inputs e Diretrizes SPICE.

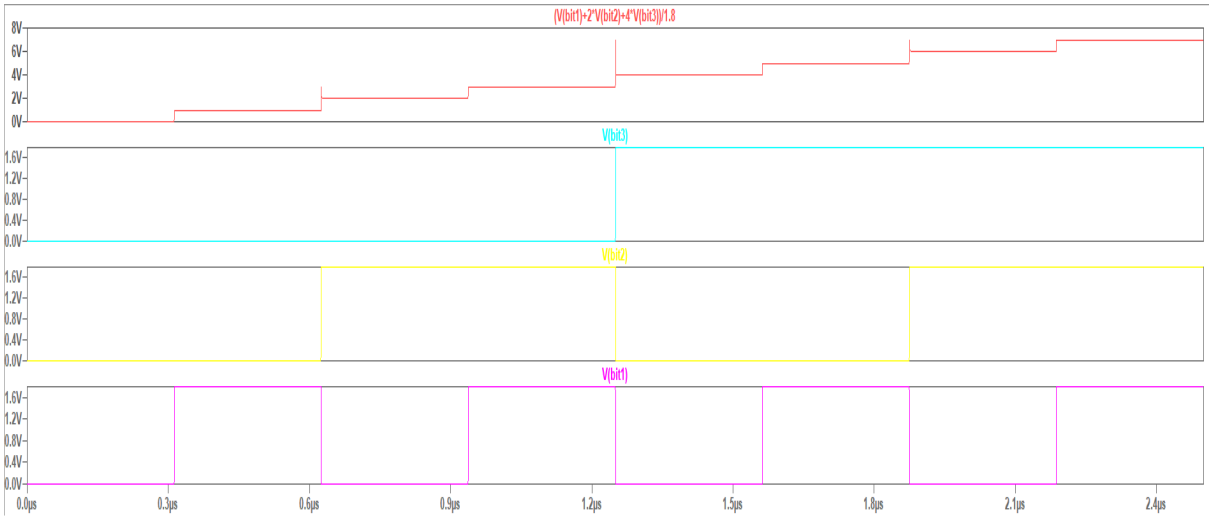
```
.inc ..\Models\TSMC018_TT.md
.tran 0 {t} 0 50p
.ic v(xnorA)=0 v(xnorB)=0 v(bit1)=0 v(bit2)=0 v(bit3)=0

.param n=8 d=c*100*1.25 t=d*n p=d*2 c=0.0025u c2=c/2 r=0.97*d VCC=1.8
.param p1=p p2=p*2 p3=p*4 p4=p*8 p5=p*16 p6=p*32 p7=p*64 p8=p*128 p9=p*256 p10=p*512 p11=p*1024 p12=p*2048
.param d1=p1/2 d10=p10/2 d11=p11/2 d12=p12/2 d2=p2/2 d3=p3/2 d4=p4/2 d5=p5/2 d6=p6/2 d7=p7/2 d8=p8/2 d9=p9/2

.save V(xnor_out) V(bit1) V(bit2) V(bit3) V(clk_coleta) V(RND0) V(rnd0_clk) I(xnor_out) V(RSTn)
```

Fonte: própria

4.1.2 Seletor de configuração

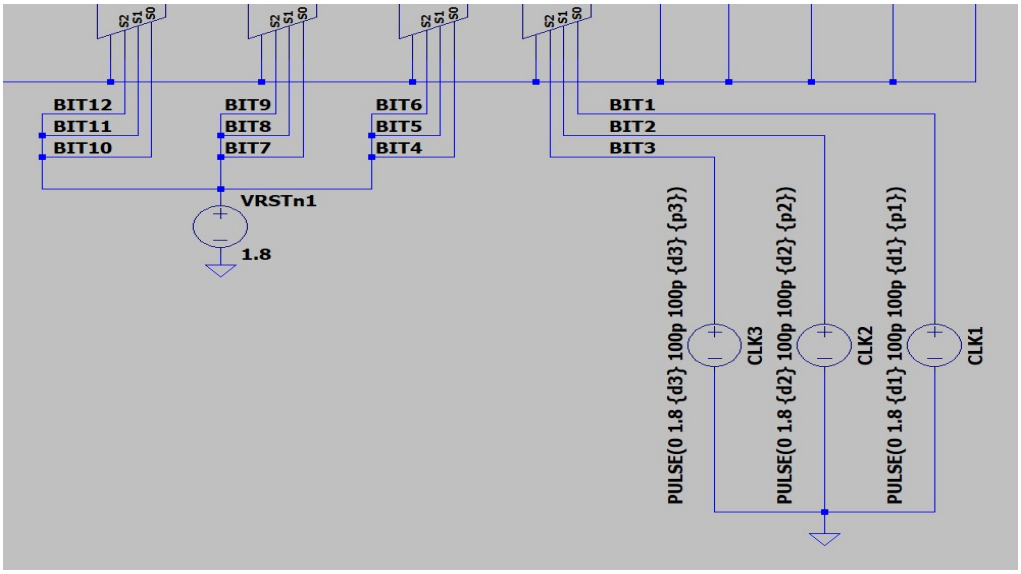
Responsável por selecionar qual configuração será utilizada, seu funcionamento é de um contador binário de doze bits, conforme explicado na seção 5.1.2. Entretanto, para a simulação tratada aqui, foram variados apenas os três bits da últimas célula de atraso - todos os outros foram travados em nível “1”, conforme visto na figura 10. Os resultados das ondas dos bits 1, 2 e 3 podem ser vistos na figura 9.

Figura 9. Formas de ondas dos Bits 1, 2 e 3.



Fonte: própria

Figura 10. Seletor de configuração

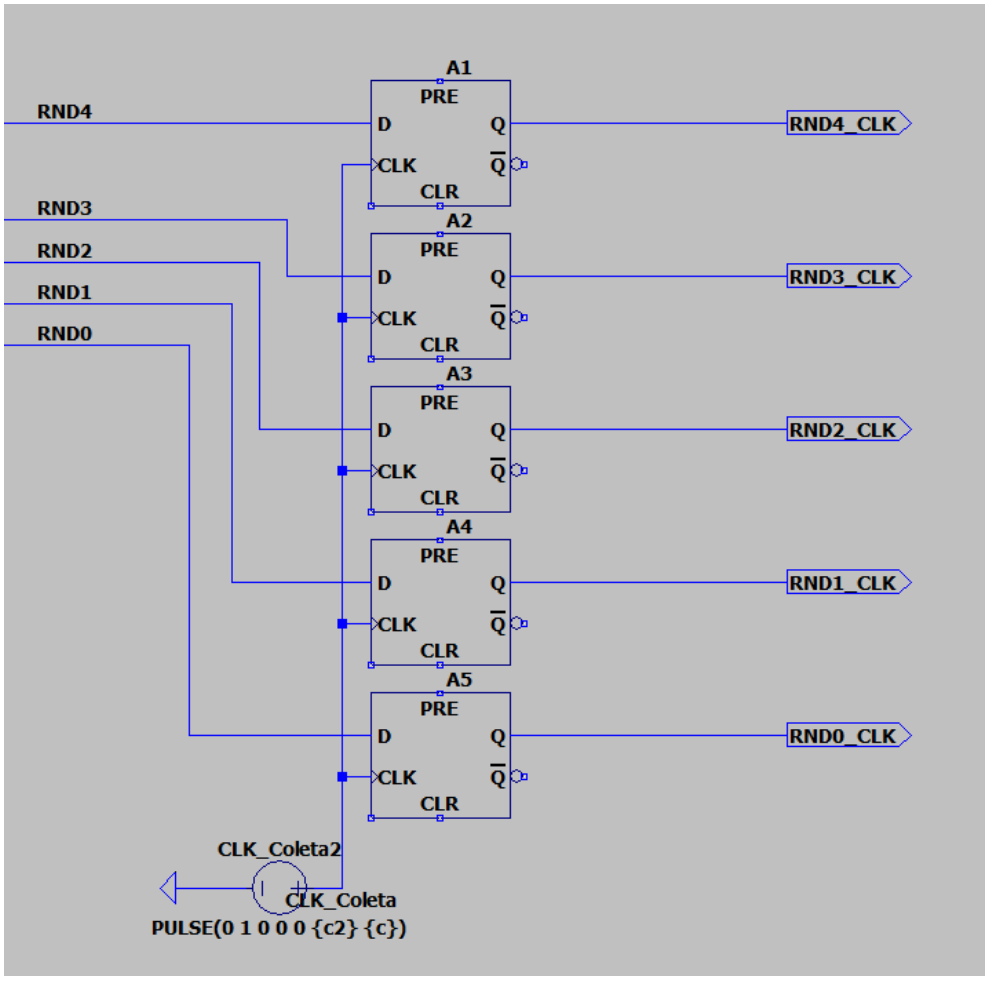


Fonte: própria

4.1.3 Circuito de coleta

O circuito de coleta da figura 11 garante que serão coletados dados binários das saídas do ALFSR, de acordo com o período *c* configurado na parte de inputs - no caso da simulação, foi escolhido o valor de 0,0025 µs.

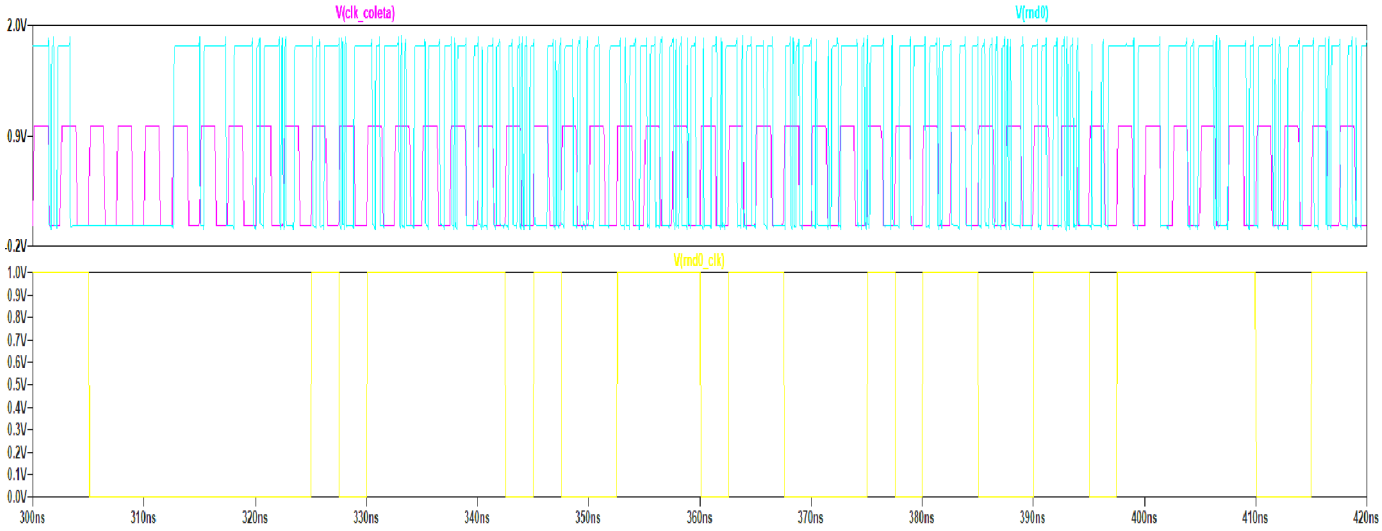
Figura 11. Circuito de coleta



Fonte: própria

A Figura 12 contém um exemplo do funcionamento da coleta. É possível perceber também no começo da onda que *RND0* se mantém como 0 por um tempo: é nesse intervalo que o reset está “ativo”. Importante notar que, por ser um circuito específico da simulação para coletar dados binários de saída prontos para o ambiente de testes, foram usados flip-flops padrões do LTSpice, que trabalham com a tensão de 1V. Por esse motivo a tensão do clock de coleta, V(clk_coleta), aparece como 1V na figura 11.

Figura 12. Exemplo coleta dados

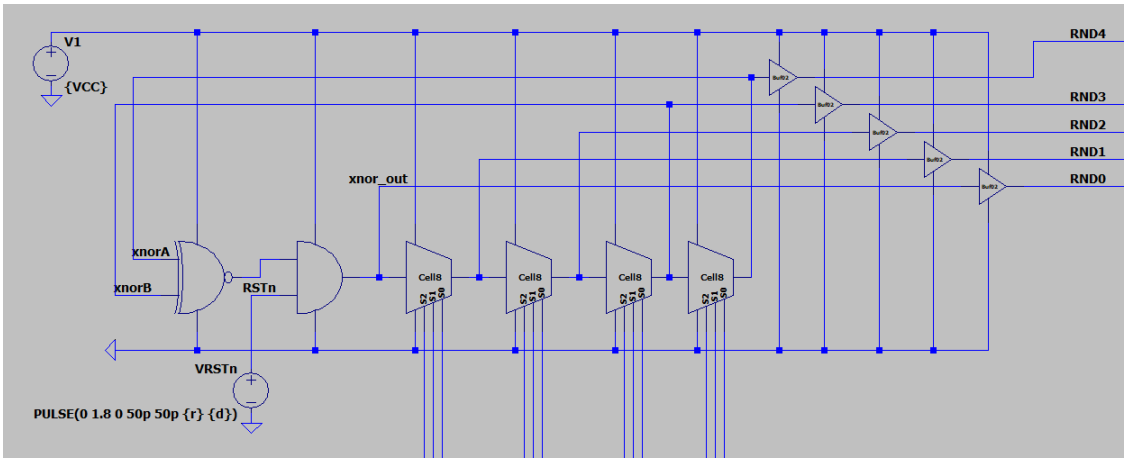


Fonte: própria

4.1.4 ALFSR

Essa é a parte do circuito que gera os dados aleatórios, sendo composta basicamente de células de atraso e uma porta *XNOR* (figura 13). O esquemático dos componentes utilizados estão no apêndice D - Esquemático dos componentes utilizados no projeto.

Figura 13. ALFSR



Fonte: própria

As células de atraso foram colocadas em série, utilizando-se as saídas das duas últimas como entradas para uma porta *XNOR*, que alimenta o sinal para a primeira célula, retroalimentando o circuito.

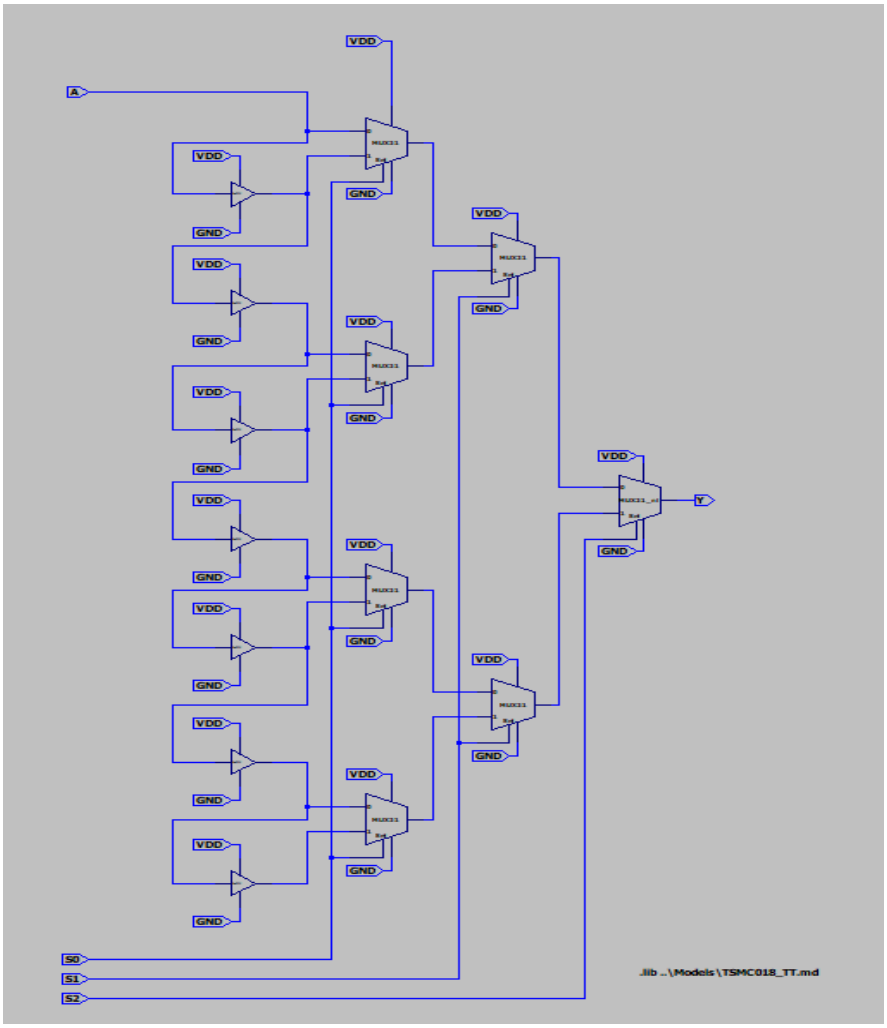
Entretanto, existe a possibilidade das duas entradas da porta *XNOR* terem nível lógico “1” ao mesmo tempo, o que faz com que a saída da porta também seja “1” e o circuito

“trave” em nível lógico alto, independente da escolha de configuração. Por isso, foi implementado uma porta *AND* entre a *XNOR* e a entrada da primeira célula a fim de resetar o circuito em toda alteração de configuração. Tal processo é garantido pela entrada “RSTn” da porta *AND* representada na Figura 7, cujo período é *d* segundos (conforme a seção 5.1.2) e *duty cycle*, 97%. Na Figura 15 é possível conferir a forma de onda na prática.

A saída de cada um dos elementos do ALFSR foi desviada para o circuito de coleta a fim de analisar seus comportamentos, passando apenas por mais um Buffer para estabilizar o sinal.

Cada uma das células de atraso (Figura 14) é composta por sete Buffers, que por sua vez são construídos com dois inversores em série. Três “colunas” de MUXes de duas entradas e uma saída, controladas pelos Bits de seleção, determinam quantos buffers o sinal terá que cruzar até sair da célula. Como as três primeiras células estão como “1”, então o delay é máximo.

Figura 14. Circuito interno da célula de atraso

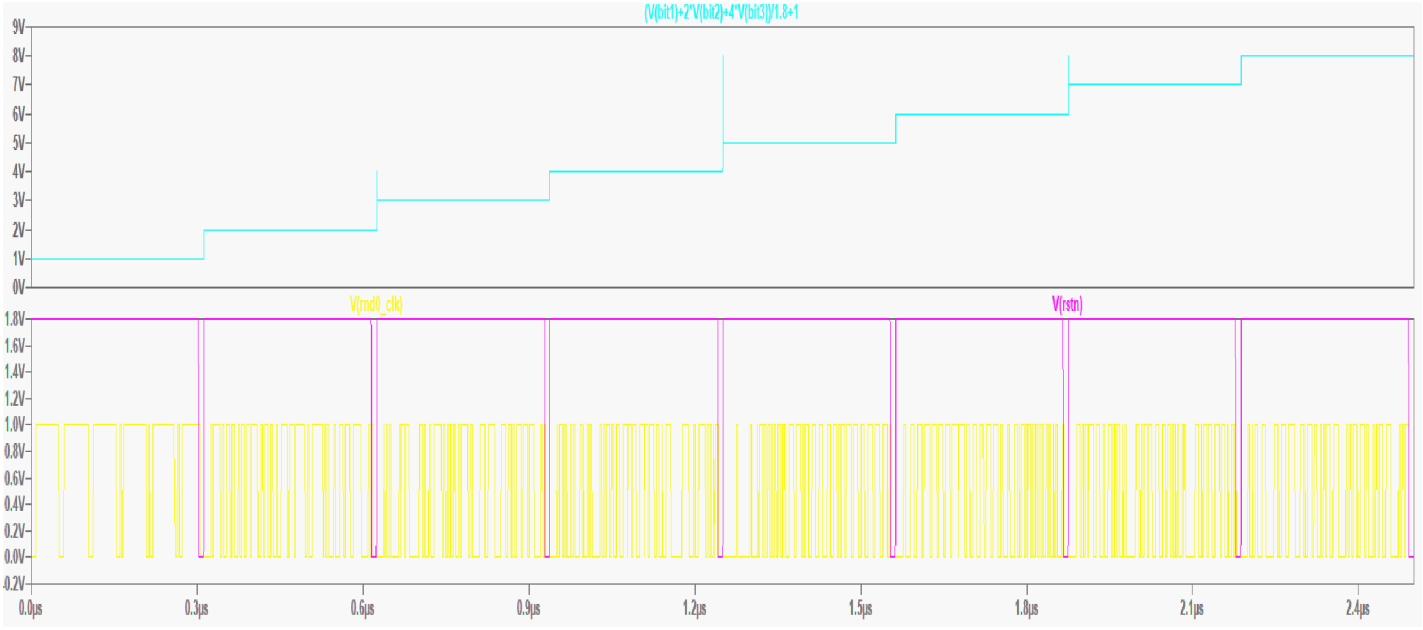


Fonte: Própria

4.1.5 Resultado

A Figura 15 mostra as oito configurações simuladas, tendo a onda de reset e um gráfico indicando qual o número da configuração em relação ao tempo para deixar mais claro qual onda pertence a qual combinação de bits.

Figura 15. Forma de onda da saída que será analisada



Fonte: Própria

Avaliando o consumo de cada configuração, podemos montar a tabela 2. Nela, o *I* da segunda coluna é a corrente RMS do circuito e a terceira coluna contém a tensão *V* de alimentação do circuito. Considerando que essa é a corrente aproximada que atravessa todo o circuito, conseguimos multiplicá-la pela tensão e obter a potência média total *I*V* de cada configuração na quarta coluna.

Já a quinta coluna contém os dados de *Bit Rate* de cada configuração. Esse valor é calculado dividindo o número de bits finais gerados pelo tempo que a configuração ficou ligada (considerando não apenas as entradas seletoras das células de atraso, mas também se a entrada do controle de reset está ligada, permitindo o funcionamento do circuito).

Na última coluna é calculada a energia por bit aleatório, utilizando a seguinte equação [3]:

$$EB = \frac{I*V}{Bit\ Rate}$$

Tabela 2: Potência x Configuração

Configuração 4ª Célula	I (A)	V (V)	I*V (W)	*Bit Rate* (bit/s)	*EB* (J/bit)
000	4,67E-05	1,80E+00	8,41E-05	4,88E+07	1,72E-12
001	6,27E-05	1,80E+00	1,13E-04	4,88E+07	2,31E-12
010	5,09E-05	1,80E+00	9,17E-05	4,88E+07	1,88E-12
011	4,87E-05	1,80E+00	8,76E-05	4,88E+07	1,79E-12
100	5,26E-05	1,80E+00	9,47E-05	4,88E+07	1,94E-12
101	4,13E-05	1,80E+00	7,43E-05	4,88E+07	1,52E-12
110	5,48E-05	1,80E+00	9,86E-05	4,88E+07	2,02E-12
111	4,45E-05	1,80E+00	8,01E-05	4,88E+07	1,64E-12
Média	**5,03E-05**	**1,80E+00**	**9,05E-05**	**4,88E+07**	**1,85E-12**

Fonte: Própria

Onde *EB* é a Energia gasta pela geração de bit aleatório (*I*, *V* e *Bit Rate* já foram apresentados). Com esses cálculos podemos ver que o custo energético fica na casa de pJ, bem abaixo do próprio circuito resultante que propõe esse mesmo tipo de estudo de outro circuito gerador de números aleatórios [3], onde ele fica na casa de nJ.

4.2 Algoritmo Octave NIST

Foram desenvolvidos cinco algoritmos, podendo ser encontrados no apêndice:

- Um para cada um dos três testes NIST descritos na seção 3.2.1.
- Um para coleta de dados da simulação do LTSpice.
- Um para coleta de dados dos arquivos de validação do NIST.

Como dito anteriormente, um dos objetivos do projeto era desenvolver um algoritmo de testes NIST que fosse independente do algoritmo que trata a fonte de dados. Como visto, conseguimos concluir essa parte, já que o código de avaliação estatística é o mesmo utilizado tanto para os arquivos de validação quanto para a simulação do LTSPice.

Inicialmente a intenção era desenvolver todos os quinze testes disponíveis na interface do NIST, mas por limitações foi possível implementar apenas os três primeiros, descritos anteriormente.

4.2.1 Validação Testes NIST

No material guia do ambiente de testes estatísticos do NIST [1] é disponibilizado uma tabela com os *Pvalues* de ondas disponíveis juntas do programa em C desenvolvido pelo instituto.

Foi rodado o algoritmo de testes do Octave utilizando o algoritmo de preparo de dados para testar quatro sequências disponibilizadas pelo NIST, sendo elas:

1. Expansão binária de π (pi)
2. Expansão binária de ε (neperiano)
3. Expansão binária de $\sqrt{2}$
4. Expansão binária de $\sqrt{3}$

Os resultados obtidos estão na figura 16.

Figura 16. Resultado teste de validação do algoritmo NIST no Octave

```
Inputs Iniciais Feitos. Tempo: 15.190613
Preencher os dados do txt em vetores para manipulao. Tempo: 15.20
teste =  1
Teste Monobit Finalizado. Tempo: 95.914124
teste =  2
Teste Frequencia em Bloco Finalizado. Tempo: 97.190331
teste =  3
Teste das Corridas Finalizado. Tempo: 179.896301
Pvalue =

   0.57821   0.95375   0.81188   0.61005
   0.38062   0.21107   0.83322   0.47396
   0.41927   0.56192   0.31343   0.26112
   0.00000   0.00000   0.00000   0.00000
   0.00000   0.00000   0.00000   0.00000
   0.00000   0.00000   0.00000   0.00000
   0.00000   0.00000   0.00000   0.00000
   0.00000   0.00000   0.00000   0.00000
   0.00000   0.00000   0.00000   0.00000
   0.00000   0.00000   0.00000   0.00000
   0.00000   0.00000   0.00000   0.00000
   0.00000   0.00000   0.00000   0.00000

Resultado =

   1   1   1   1
   1   1   1   1
   1   1   1   1
   0   0   0   0
   0   0   0   0
   0   0   0   0
   0   0   0   0
   0   0   0   0
   0   0   0   0
   0   0   0   0
```

Fonte: Própria

Comparando os resultados obtidos com a tabela disponibilizada pelo NIST, temos os dados da tabela 3. Com isso, podemos concluir que os três testes implementados funcionam com os mesmos resultados que o ambiente de testes do NIST.

Tabela 3. Comparação entre os *Pvalues* do algoritmo de Octave e a tabela referência do NIST.

Testes	Pi OCTAVE	Pi NIST	Erro Pi	Neperiano OCTAVE	Neperiano NIST	Erro Neperiano	$\sqrt{2}$ OCTAVE	$\sqrt{2}$ NIST	Erro $\sqrt{2}$	$\sqrt{3}$ OCTAVE	$\sqrt{3}$ NIST	Erro $\sqrt{3}$
Teste frequência monobit	0,578211	0,578211	0,00%	0,953749	0,953749	0,00%	0,811881	0,811881	0,00%	0,6100514618	0,610051	0,00%
Teste frequência em bloco	0,380615	0,380615	0,00%	0,211072	0,211072	0,00%	0,833222	0,833222	0,00%	0,4739612657	0,473961	0,00%
Teste das corridas	0,419268	0,419268	0,00%	0,561917	0,561917	0,00%	0,313427	0,313427	0,00%	0,2611232603	0,261123	0,00%

Fonte: Própria

4.2.2 Teste Aleatoriedade ALFSR

Rodando os dados da simulação de LTspice no algoritmo validado gera o resultado da figura 17.

Figura 17. Resultado testes estatísticos das configurações entre os bits 111.111.111.000...111.111.111.111 do ALFSR

```
Inputs Iniciais Feitos. Tempo: 0.020218
Vetor Clock Finalizado. Tempo: 2.385864
Preencher os dados do txt em vetores para manipulação. Tempo: 4.534225
teste =  1
Teste Monobit Finalizado. Tempo: 4.566452
teste =  2
Teste Frequencia em Bloco Finalizado. Tempo: 4.607254
teste =  3
Teste das Corridas Finalizado. Tempo: 4.628571
Pvalue =

 Columns 1 through 6:

   5.9552e-12   2.9298e-04   7.1725e-01   7.1725e-01   1.0000e+00   2.0498e-01
   3.2448e-08   8.5756e-03   3.7990e-01   4.0116e-01   1.6888e-03   1.7364e-02
   5.1007e-12   8.5026e-04   3.7108e-01   5.7843e-01   3.6528e-01   9.7121e-01

 Columns 7 and 8:

   1.0000e+00   2.9791e-02
   9.0042e-01   4.6945e-01
   3.6528e-01   9.4584e-01

Resultado =

   0   0   1   1   1   1   1   1
   0   0   1   1   0   1   1   1
   0   0   1   1   1   1   1   1
```

Fonte: Própria

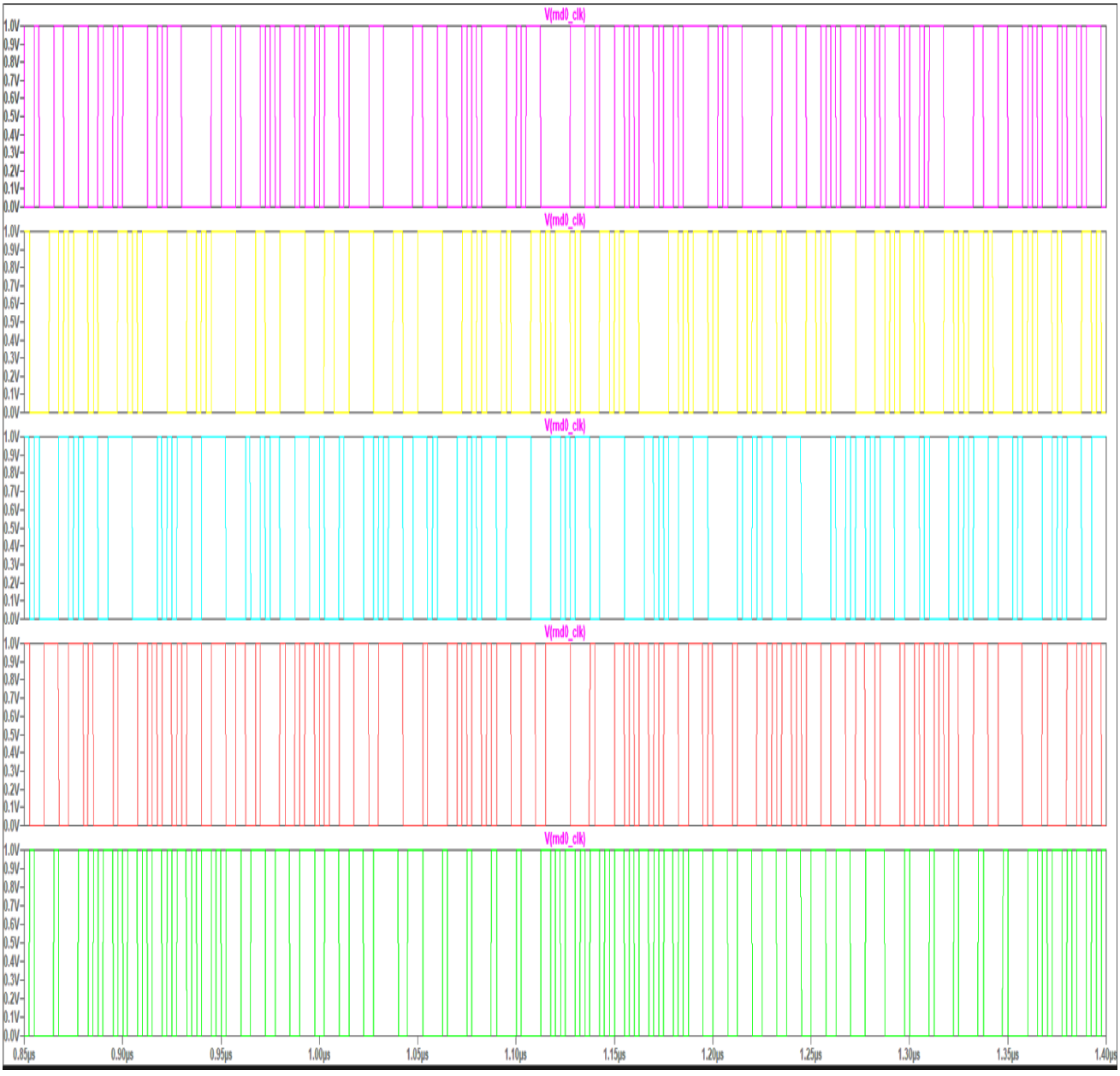
Como é possível avaliar na matriz “Resultado”, as configurações 000 (1ª coluna), 001(2ª coluna) e 100 (5ª coluna) se provaram não aleatórias por falharem em pelo menos um dos testes (o que os faz falhar automaticamente em todos os seguintes).

Entretanto, as demais configurações passaram em todos os testes, indicando que elas podem ser de fato aleatórias e que o circuito tem a capacidade de gerar números aleatórios.

4.3 Variações de temperatura

Foi feito também uma simulação final variando a temperatura do circuito para observar se isso afetaria os testes de aleatoriedade de uma configuração. Baseado nos teste explicados anteriormente, a configuração 111 foi simulada novamente com mais tempo para cinco valores de temperatura: 0ºC, 15ºC, 30ºC, 45ºC e 60ºC. Parte das formas de onda resultantes do sistema de coleta estão na figura 12 - as temperaturas estão arranjadas de cima (mais frio, começando de 0º) para baixo (mais quente, terminando em 60ºC).

Figura 18. Formas de ondas da simulação de temperatura.



Fonte: Própria

A tabela 4 são apresentados os valores de P e na tabela 5, seus respectivos resultados de aleatoriedade testando cada uma das temperaturas no mesmo algoritmo de teste NIST do Octave apresentado anteriormente para $\alpha = 0.01$. Na tabela 5, um resultado "1" significa que a temperatura passou naquele teste estatístico e que a hipótese de aleatoriedade foi aprovada, enquanto "0" significa o contrário, que a hipótese foi rejeitada.

Tabela 4. Valores P por temperatura.

PValue	0ºC	15ºC	30ºC	45ºC	60ºC
Teste frequência monobit	0,0065	0,1290	0,6580	0,8003	0,6580
Teste frequência em bloco	0,3450	1,0000	0,9998	0,9989	0,0000
Teste das corridas	0,2989	0,0000	0,1305	0,5706	0,5230

Fonte: própria

Tabela 5. Resultados de aleatoriedade por temperatura

Resultado (Alpha=0.01)	0ºC	15ºC	30ºC	45ºC	60ºC
Teste frequência monobit	0	1	1	1	1
Teste frequência em bloco	1	1	1	1	0
Teste das corridas	1	0	1	1	1

Fonte: própria

É possível perceber que as únicas duas temperaturas que passaram em todos os testes foram as de 30ºC e 45ºC, indicando que o circuito talvez não funcione tão bem fora dessa faixa. São necessários mais testes com outras configurações para entender se isso é uma característica da configuração ou do circuito como um todo, além de explorar mais a faixa entre 15ºC e 30ºC e entender qual seria a menor temperatura para o qual a configuração não é recusada em nenhum teste.

Concluídas as apresentações e análises, vemos que tanto a simulação do circuito quanto o ambiente de testes foram entregues e com resultados satisfatórios para este trabalho. A simulação mostrou indícios de que consegue gerar sequências aleatórias em mais de uma configuração e dentro da faixa de temperatura entre 30º e 45º, que, apesar de apenas duas temperaturas terem passado em todos os testes, ainda são necessários mais testes para determinar exatamente quais as temperaturas mínimas e máximas de operação do circuito para cada configuração diferente. Já o ambiente de testes foi validado e mostrou-se confiável para avaliar sequências aleatórias, necessitando agora ser aprimorado com a ampliação dos testes estatísticos que ele engloba.

5 Conclusão

Como dito anteriormente, o trabalho atingiu seus objetivos, sendo possível mostrar que a ideia de separar os algoritmos de tratamento de dados e de testes estatísticos no Octave tornou esta análise mais versátil, principalmente considerando que a intenção é a de preparar uma biblioteca com esses testes estatísticos e disponibilizá-los como funções próprias de pacotes do Octave. Além disso, obtivemos indícios de que o circuito ALFSR simulado tem a capacidade de gerar números aleatórios em diferentes configurações e temperaturas.

É de interesse também desenvolver um pacote de processamento dos sinais, pois mesmo que o algoritmo de processamento tenha sido feito pensando nos dados de LTspice, o usuário ainda precisa tomar alguns cuidados para que ele funcione sem necessidade de adaptações além dos dados de entrada. O principal problema que não foi resolvido foi a incompatibilidade com dados coletados de simulações que utilizem o comando .step param, já que o LTspice dessa forma inclui uma linha de *string* no começo de cada corrida diferente.

Outro ponto que é necessário ressaltar: qualquer algoritmo de processamento de dados precisa declarar algumas variáveis que o algoritmo de NIST vai usar, sendo assim, os testes estatísticos são dependentes de uma forma que não fica atualmente muito clara no código - a solução de torná-los um pacote de funções resolverá o problema, visto que essas variáveis seriam inputs.

Como trabalhos futuros, propõe-se terminar de desenvolver todos os testes NIST, a fim de concluir a avaliação da topologia de ALFSR, além de continuar testando as configurações restantes em diferentes temperaturas, avaliando para encontrar aquelas que se mostrarem mais estáveis e com um custo/benefício energético melhor.

Referências

[1] I.T.L Computer Security Division, “NIST SP 800-22, A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications” [Online]. Disponível em https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-22r1a.pdf [Acessado: 01-jun-2020].

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

[3] PARK, M., RODGERS, J., LATHROP, D. (2015). True random number generation using CMOS Boolean chaotic oscillator. Microelectronics Journal. 46. 10.1016/j.mejo.2015.09.015.

[4] ZHANG, R., CAVALCANTE, H., GAO, Z., GAUTHIER, D., SOCOLAR, J., ADAMS, M., LATHROP, D., (2009). Boolean Chaos. Physical review. E, Statistical, nonlinear, and soft matter physics. 80. 045202. 10.1103/PhysRevE.80.045202.

[5] PARK, M., RODGERS, J., LATHROP, D., (2012). Modeling chaos in on-chip ultra-wideband chaotic oscillator. IEEE MTT-S International Microwave Symposium digest. IEEE MTT-S International Microwave Symposium. 1-3. 10.1109/MWSYM.2012.6259678.

[6] ADDABBO, T., FORT, A., MORETTI, R., MUGNAINI, M., VIGNOLI, V., GARCIA-BOSQUE, M., "Lightweight True Random Bit Generators in PLDs: Figures of Merit and Performance Comparison," 2019 IEEE International Symposium on Circuits and Systems (ISCAS), Sapporo, Japan, 2019, pp. 1-5, doi: 10.1109/ISCAS.2019.8702791.

[7] 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.

[8] CATTANI, M., Deterministic Chaos Theory: Basic Concepts. Rev. Bras. Ensino Fís., São Paulo , v. 39, n. 1, e1309, 2017 . Disponível em <http://www.scielo.br/scielo.php?script=sci_arttext&pid=S1806-11172017000100409&lng=pt&nrm=iso>. acessos em 23 jun. 2020. Epub 17-Out-2016. https://doi.org/10.1590/1806-9126-rbef-2016-0185.

[9] YANG, K., BLAAUW, D., SYLVESTER, D., "An All-Digital Edge Racing True Random Number Generator Robust Against PVT Variations," in IEEE Journal of Solid-State Circuits, vol. 51, no. 4, pp. 1022-1031, April 2016, doi: 10.1109/JSSC.2016.2519383.

[10] ANTOGNETTI, P., MASSOBRIO, G., 1993. Semiconductor Device Modeling with Spice (2nd. ed.). McGraw-Hill, Inc., USA.

[11] Rosin, David. (2015). Dynamics of Complex Autonomous Boolean Networks. 10.1007/978-3-319-13578-6.

[12] LTSpice, Versão 17, Analog Devices Inc., https://www.analog.com/en/design-center/design-tools-and-calculators/ltspice-simulator.html

[13] EATON, J. W., BATEMAN, D., HAUBERG, S., WEHBRING, R., (2014). GNU Octave version 5.2.0 manual: a high-level interactive language for numerical computations. CreateSpace Independent Publishing Platform. ISBN 1441413006, URL http://www.gnu.org/software/octave/doc/interpreter/

[14] SPICE Differentiation. Disponível em: <https://www.analog.com/ru/technical-articles/spice-differentiation.html>. Acesso em 25/06/2020.

[15] ROSIN, D. P., Dynamics of Complex Autonomous Boolean Networks. Springer International Publishing, 2015. ISBN 3319135775, 9783319135779.

Apêndice A - Códigos de avaliação de teste estatístico do NIST em formato “.m”

```
clear
clc
close
pkg load control
pkg load statistics
##fid = fopen('data.pi.pi');
##data = textscan(fid,'%f', -1);
###header = textscan(fid,'%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s', 1);
##fid = fclose(fid);


t0 = clock();
#Caracteristicas do arquivo
#Coluna de tempo
#tc = 1;
#Coluna Clock
#cc = 2;
#Coluna modo
#bc = 9;
#Coluna Reset
#r = 8;
#Começo coluna Dados
cd = 1;
#número colunas de dados
nd = 1;

#bits a disposição?
b = 2;
#Tolerancia testes
a = 0.01;
#constantes
sqrt2 = 1.41421356237309504880;

#matrix dados
n = 1000000;
V(:,4) = load('data.sqrt3');
V(:,1) = load('data.pi');
V(:,2) = load('data.e');
V(:,3) = load('data.sqrt2');

#Inicio Matrizes
Resultado = zeros(12,(2^b),nd);
Pvalue = zeros(12,(2^b),nd);
bit1 = zeros(1,(2^b),nd);
bit0 = zeros(1,(2^b),nd);
Sn = zeros(1,(2^b),nd);

et=1;
elapsed_time(et) = etime (clock (), t0);
printf("Inputs Iniciais Feitos. Tempo: %f\n",elapsed_time(et))

#n = número de cada corrida
n = 1000000;

et++;
elapsed_time(et) = etime (clock (), t0);
printf("Preencher os dados do txt em vetores para manipulação. Tempo:
%f\n",elapsed_time(et))

#Teste Frequencia (Monobit)
p=1;
teste=1
while p<=nd
  c=1;
  while c<=size(V,2)
    nv=1;
    while nv<=n
      Sn(1,c,p)=Sn(1,c,p)+(2*(V(nv,c))-1);
      nv++;
    end
    Sobs(1,c,p)=abs(Sn(1,c,p))/sqrt(n);
    Erfc(1,c,p) = Sobs(1,c,p)/sqrt2;
    Pvalue(1,c,p) = erfc(Erfc(1,c,p));
      if (Pvalue(teste,c,p)<a)
        Resultado(teste,c,p) = 0; #Não Passou
```

```
75          elseif Pvalue(teste,c,p)>=a;
76            Resultado(teste,c,p) = 1; #Passou
77          else
78            Resultado(teste,c,p) = -1; #Erro
79          endif
80        c++;
81      end
82      p++;
83    end
84
85    et++;
86    elapsed_time(et) = etime (clock (), t0);
87    printf("Teste Monobit Finalizado. Tempo: %f\n",elapsed_time(et))
88
89    #Teste Frequencia em Bloco
90    c=1;
91    p=1;
92    M=128;
93    N=floor(n/M);
94    teste++;teste
95    while p<=nd
96      c=1;
97      while c<=size(V,2)
98        if (Resultado(teste-1,c,p)==1)
99          l=1;
100         while l<=N
101           pi(l,c,p) = sum(V([M*(l-1)+1:M*l],c,p))/M;
102           l++;
103         end
104         X(1,c,p)=4*M*sum(((pi(:,c,p).-(1/2)).^2));
105         Pvalue(teste,c,p)=1-gammainc(X(1,c,p)/2,N/2);
106         if Pvalue(teste,c,p)<a
107           Resultado(teste,c,p) = 0; #Não Passou
108           elseif Pvalue(teste,c,p)>=a
109            Resultado(teste,c,p) = 1; #Passou
110           else
111            Resultado(teste,c,p) = -1; #Erro
112          endif
113       else
114       endif
115       c++;
116     end
117     p++;
118   end
119
120   et++;
121   elapsed_time(et) = etime (clock (), t0);
122   printf("Teste Frequencia em Bloco Finalizado. Tempo: %f\n",elapsed_time(et))
123
124
125   #Teste das Corridas
126   c=1;
127   teste++;teste
128   p=1;
129   Vn = zeros(1,(2^b),nd);
130   while p<=nd
131     c=1;
132       while c<=size(V,2)
133         if (Resultado(teste-1,c,p)==1)
134           l=2;
135           while l<=n
136             if (V(l,c,p)!=V(l-1,c,p))
137               Vn(1,c,p)++;
138             endif
139            l++;
140           end
141           pimono(1,c,p)=sum(V([1:n],c,p))/n;
142           Erfc_arg =
              abs(Vn(1,c,p)+1-2*n*pimono(1,c,p)*(1-pimono(1,c,p)))/(2*sqrt(2*n)*pimono(1,c,p)*(1
              -pimono(1,c,p)));
143           Pvalue(teste,c,p)=erfc(Erfc_arg);
144           if Pvalue(teste,c,p)<a
145               Resultado(teste,c,p) = 0; #Não Passou
146               elseif Pvalue(teste,c,p)>=a
147                Resultado(teste,c,p) = 1 ;#Passou
```

```
148                else
149                 Resultado(teste,c,p) = -1 ;#Erro
150              endif
151            else
152            endif
153            c++;
154          end
155        p++;
156      end
157
158      et++;
159      elapsed_time(et) = etime (clock (), t0);
160      printf("Teste das Corridas Finalizado. Tempo: %f\n",elapsed_time(et))
161
162
163      Pvalue
164      Resultado
```

Apêndice B - Códigos de avaliação de circuito ALFSR em formato “.m”

```
clear
clc
close
pkg load control
fid = fopen('ALFSR8_RSTn_V8.txt');
data = textscan(fid,'%f%f%f%f%f%f%f%f', -1,'headerLines',1);
fid = fclose(fid);

#Caracteristicas do arquivo
#Coluna de tempo
tc = 1;
#Coluna Clock
cc = 5;
#Coluna modo
#bc = 9;
#Coluna Reset
r = 8;
#Começo coluna Dados
cd = 7;
#número colunas de dados
nd = 1;
#horário começo simulação
t0 = clock;

#bits a disposição?
b = 3;
#Tolerancia testes
a = 0.01;
#numero de testes a serem rodados
nt = 3;

#matrix dados
f = cell2mat(data);
#Vetor tempo
t = f(:,tc);
#Inicio Matrizes
Resultado = zeros(nt,(2^b),nd);
Pvalue = zeros(nt,(2^b),nd);
bit1 = zeros(1,(2^b),nd);
bit0 = zeros(1,(2^b),nd);
Sn = zeros(1,(2^b),nd);

et=1;
elapsed_time(et) = etime (clock (), t0);
printf("Inputs Iniciais Feitos. Tempo: %f\n",elapsed_time(et))

#Vetor Clock - ele serve para puxar apenas os dados que vão ser utilizados pelo teste
l=2;
clk(:,1) = f(:,cc)*0;
while l<=size(f,1)
  if (f(l,r)==1.8)
        if (f(l-1,cc)!=1 && f(l,cc)==1)
          clk(l,1)=1;
        else
          clk(l,1)=0;
        endif
  else
   clk(l,1) = 0;
  endif
l++;
end

et++;
elapsed_time(et) = etime (clock (), t0);
printf("Vetor Clock Finalizado. Tempo: %f\n",elapsed_time(et))


#Preencher os dados do txt em vetores para manipulação
c=1;
l=2;
lv=1;
while l<size(f,1)
      if (f(l,r)==1.8 && f(l+1,r)!=1.8)
        c++;
        lv=1;
```

```
76          else
77            if (clk(l,1)==1)
78               p = 1;
79                 while p<=nd
80                  V(lv,c,p) = f(l,cd+p-1);
81                  p++;
82                 end
83               lv++;
84            else
85            endif
86          endif
87          l++;
88    end
89
90    #n = número de cada corrida
91    n = size(V,1);
92
93    et++;
94    elapsed_time(et) = etime (clock (), t0);
95    printf("Preencher os dados do txt em vetores para manipulação. Tempo:
      %f\n",elapsed_time(et))
96
97    #Teste Frequencia (Monobit)
98    p=1;
99    teste=1
100   while p<=nd
101     c=1;
102     while c<=size(V,2)
103       nv=1;
104       while nv<=n
105         Sn(1,c,p)=Sn(1,c,p)+(2*(V(nv,c,p))-1);
106         nv++;
107       end
108       Sobs(1,c,p) = abs(Sn(1,c,p))/sqrt(n);
109       Erfc_1(1,c,p) = Sobs(1,c,p)/sqrt(2);
110       Pvalue(1,c,p) = erfc(Erfc_1(1,c,p));
111         if (Pvalue(teste,c,p)<a)
112           Resultado(teste,c,p) = 0; #Não Passou
113         elseif Pvalue(teste,c,p)>=a;
114           Resultado(teste,c,p) = 1; #Passou
115         else
116           Resultado(teste,c,p) = -1; #Erro
117         endif
118       c++;
119     end
120     p++;
121   end
122
123   et++;
124   elapsed_time(et) = etime (clock (), t0);
125   printf("Teste Monobit Finalizado. Tempo: %f\n",elapsed_time(et))
126
127   #Teste Frequencia em Bloco
128   c=1;
129   p=1;
130   M=20;
131   N=floor(n/M);
132   teste++;teste
133   while p<=nd
134     c=1;
135     while c<=size(V,2)
136         l=1;
137         while l<=N
138           pi(l,c,p) = sum(V([M*(l-1)+1:M*l],c,p))/M;
139           l++;
140         end
141         X(1,c,p)=4*M*sum(((pi(:,c,p).-(1/2)).^2));
142         Pvalue(teste,c,p)=1-gammainc(X(1,c,p)/2,N/2);
143         if Pvalue(teste,c,p)<a
144           Resultado(teste,c,p) = 0; #Não Passou
145           elseif Pvalue(teste,c,p)>=a
146            Resultado(teste,c,p) = 1; #Passou
147           else
148            Resultado(teste,c,p) = -1; #Erro
149          endif
```

```
150       c++;
151     end
152     p++;
153   end
154
155   et++;
156   elapsed_time(et) = etime (clock (), t0);
157   printf("Teste Frequencia em Bloco Finalizado. Tempo: %f\n",elapsed_time(et))
158
159
160   #Teste das Corridas
161   c=1;
162   teste++;teste
163   p=1;
164   Vn = zeros(1,(2^b),nd);
165   while p<=nd
166     c=1;
167       while c<=size(V,2)
168           l=2;
169           while l<=n
170             if (V(l,c,p)!=V(l-1,c,p))
171               Vn(1,c,p)++;
172             endif
173            l++;
174           end
175           pimono(1,c,p)=sum(V([1:n],c,p))/n;
176           Erfc_3(1,c,p) =
          abs(Vn(1,c,p)+1-2*n*pimono(1,c,p)*(1-pimono(1,c,p)))/(2*sqrt(2*n)*pimono(1,c,p)*(1
          -pimono(1,c,p)));
177           Pvalue(teste,c,p)=erfc(Erfc_3(1,c,p));
178           if Pvalue(teste,c,p)<a
179               Resultado(teste,c,p) = 0; #Não Passou
180               elseif Pvalue(teste,c,p)>=a
181                Resultado(teste,c,p) = 1 ;#Passou
182               else
183                Resultado(teste,c,p) = -1 ;#Erro
184           endif
185         c++;
186       end
187     p++;
188   end
189
190   et++;
191   elapsed_time(et) = etime (clock (), t0);
192   printf("Teste das Corridas Finalizado. Tempo: %f\n",elapsed_time(et))
193
194
195   Pvalue
196   Resultado
```

Apêndice C - 35 primeiras linhas do arquivo em formato “.txt” gerado pelo LTSpice

time	V(bit1)	V(bit2)	V(bit3)	V(clk_coleta)	V(rnd0)	V(rnd0_clk)	V(rstn)	V(xnor_out)			
0.000000000000000e+000	0.000000e+000	0.000000e+000	0.000000e+000	0.000000e+000	0.000000e+000	5.794203e-009	0.000000e+000	0.000000e+000	6.081074e-009
5.851428592654594e-013	0.000000e+000	0.000000e+000	0.000000e+000	0.000000e+000	4.681143e-003	3.833081e-005	0.000000e+000	2.106514e-002	4.403265e-004
1.170285718530919e-012	0.000000e+000	0.000000e+000	0.000000e+000	0.000000e+000	9.362286e-003	6.723530e-005	0.000000e+000	4.213029e-002	8.366953e-004
1.755428577796378e-012	0.000000e+000	0.000000e+000	0.000000e+000	0.000000e+000	1.404343e-002	8.671926e-005	0.000000e+000	6.319543e-002	1.189112e-003
2.340571437061837e-012	0.000000e+000	0.000000e+000	0.000000e+000	0.000000e+000	1.872457e-002	9.678270e-005	0.000000e+000	8.426057e-002	1.497578e-003
2.925714296327297e-012	0.000000e+000	0.000000e+000	0.000000e+000	0.000000e+000	2.340571e-002	9.742561e-005	0.000000e+000	1.053257e-001	1.762092e-003
3.510857155592756e-012	0.000000e+000	0.000000e+000	0.000000e+000	0.000000e+000	2.808686e-002	8.864800e-005	0.000000e+000	1.263909e-001	1.982654e-003
4.096000014858214e-012	0.000000e+000	0.000000e+000	0.000000e+000	0.000000e+000	3.276800e-002	7.044987e-005	0.000000e+000	1.474560e-001	2.159265e-003
8.192000029716429e-012	0.000000e+000	0.000000e+000	0.000000e+000	0.000000e+000	6.553600e-002	1.399844e-005	0.000000e+000	2.949120e-001	2.994932e-003
1.638400005943286e-011	0.000000e+000	0.000000e+000	0.000000e+000	0.000000e+000	1.310720e-001	-2.725829e-005	0.000000e+000	5.898240e-001	2.405233e-003
3.276800011886571e-011	0.000000e+000	0.000000e+000	0.000000e+000	0.000000e+000	2.621440e-001	-8.772958e-005	0.000000e+000	1.179648e+000	-8.876598e-003
5.000000000000000e-011	0.000000e+000	0.000000e+000	0.000000e+000	0.000000e+000	4.000000e-001	-2.833841e-004	0.000000e+000	1.800000e+000	-3.433415e-002
5.172319998811343e-011	0.000000e+000	0.000000e+000	0.000000e+000	0.000000e+000	4.137856e-001	1.955277e-005	0.000000e+000	1.800000e+000	-3.219894e-002
5.516959996434029e-011	0.000000e+000	0.000000e+000	0.000000e+000	0.000000e+000	4.413568e-001	6.554342e-004	0.000000e+000	1.800000e+000	-1.435945e-002
6.206239991679400e-011	0.000000e+000	0.000000e+000	0.000000e+000	0.000000e+000	4.964992e-001	2.237386e-003	0.000000e+000	1.800000e+000	6.660601e-002
7.584799982170144e-011	0.000000e+000	0.000000e+000	0.000000e+000	0.000000e+000	6.067840e-001	1.941918e-003	0.000000e+000	1.800000e+000	2.966211e-001
8.835151382892105e-011	0.000000e+000	0.000000e+000	0.000000e+000	0.000000e+000	7.068121e-001	9.517808e-004	0.000000e+000	1.800000e+000	5.224717e-001
1.009574962943709e-010	0.000000e+000	0.000000e+000	0.000000e+000	0.000000e+000	8.076600e-001	-2.673545e-003	0.000000e+000	1.800000e+000	7.299587e-001
1.250000000000000e-010	0.000000e+000	0.000000e+000	0.000000e+000	0.000000e+000	1.000000e+000	-1.503868e-002	0.000000e+000	1.800000e+000	1.078251e+000
1.274042503705629e-010	0.000000e+000	0.000000e+000	0.000000e+000	0.000000e+000	1.000000e+000	-1.992721e-002	0.000000e+000	1.800000e+000	1.107980e+000
1.322127511116887e-010	0.000000e+000	0.000000e+000	0.000000e+000	0.000000e+000	1.000000e+000	-2.743278e-002	0.000000e+000	1.800000e+000	1.164449e+000
1.418297525939403e-010	0.000000e+000	0.000000e+000	0.000000e+000	0.000000e+000	1.000000e+000	-3.077911e-002	0.000000e+000	1.800000e+000	1.266724e+000
1.610637555584436e-010	0.000000e+000	0.000000e+000	0.000000e+000	0.000000e+000	1.000000e+000	1.091497e-001	0.000000e+000	1.800000e+000	1.440025e+000
1.817109800443523e-010	0.000000e+000	0.000000e+000	0.000000e+000	0.000000e+000	1.000000e+000	1.514595e+000	0.000000e+000	1.800000e+000	1.572770e+000
2.050822012181207e-010	0.000000e+000	0.000000e+000	0.000000e+000	0.000000e+000	1.000000e+000	1.753473e+000	0.000000e+000	1.800000e+000	1.651812e+000
2.244086094464725e-010	0.000000e+000	0.000000e+000	0.000000e+000	0.000000e+000	1.000000e+000	1.788906e+000	0.000000e+000	1.800000e+000	1.660107e+000
2.457652510869126e-010	0.000000e+000	0.000000e+000	0.000000e+000	0.000000e+000	1.000000e+000	1.800288e+000	0.000000e+000	1.800000e+000	1.636556e+000
2.651875560927120e-010	0.000000e+000	0.000000e+000	0.000000e+000	0.000000e+000	1.000000e+000	1.800365e+000	0.000000e+000	1.800000e+000	1.641687e+000
3.106793187090768e-010	0.000000e+000	0.000000e+000	0.000000e+000	0.000000e+000	1.000000e+000	1.799662e+000	0.000000e+000	1.800000e+000	1.738728e+000
3.310232576542547e-010	0.000000e+000	0.000000e+000	0.000000e+000	0.000000e+000	1.000000e+000	1.799882e+000	0.000000e+000	1.800000e+000	1.766290e+000
3.719362060173625e-010	0.000000e+000	0.000000e+000	0.000000e+000	0.000000e+000	1.000000e+000	1.799961e+000	0.000000e+000	1.800000e+000	1.789829e+000
3.983781482878791e-010	0.000000e+000	0.000000e+000	0.000000e+000	0.000000e+000	1.000000e+000	1.799974e+000	0.000000e+000	1.800000e+000	1.795165e+000
4.248200905583958e-010	0.000000e+000	0.000000e+000	0.000000e+000	0.000000e+000	1.000000e+000	1.799984e+000	0.000000e+000	1.800000e+000	1.799050e+000
4.512620328289125e-010	0.000000e+000	0.000000e+000	0.000000e+000	0.000000e+000	1.000000e+000	1.799991e+000	0.000000e+000	1.800000e+000	1.801486e+000

Apêndice D - Esquemático dos componentes utilizados no projeto.

Figura D-1. Porta AND (and02)

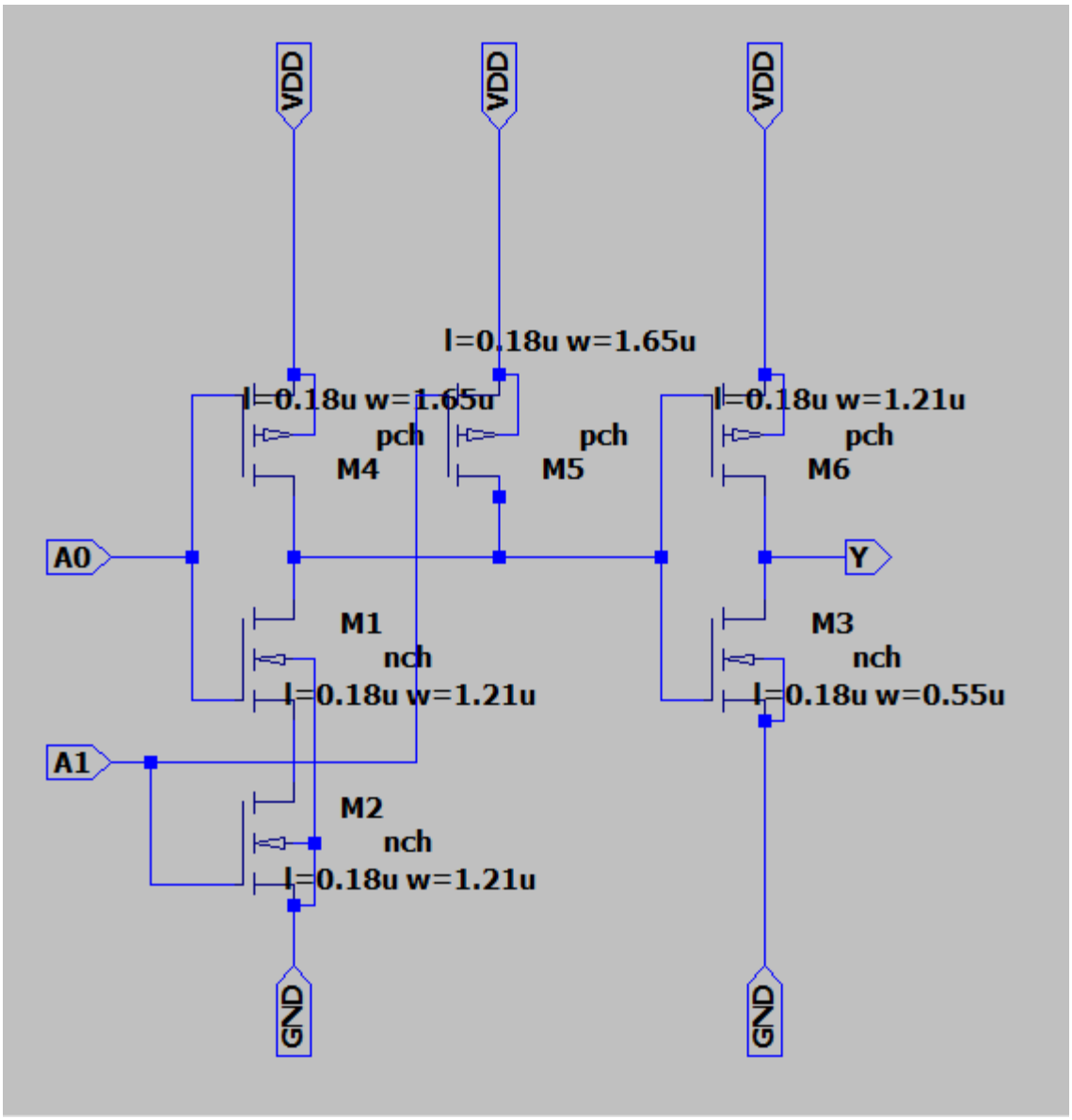


Figura D-2. Buffer (buf02)

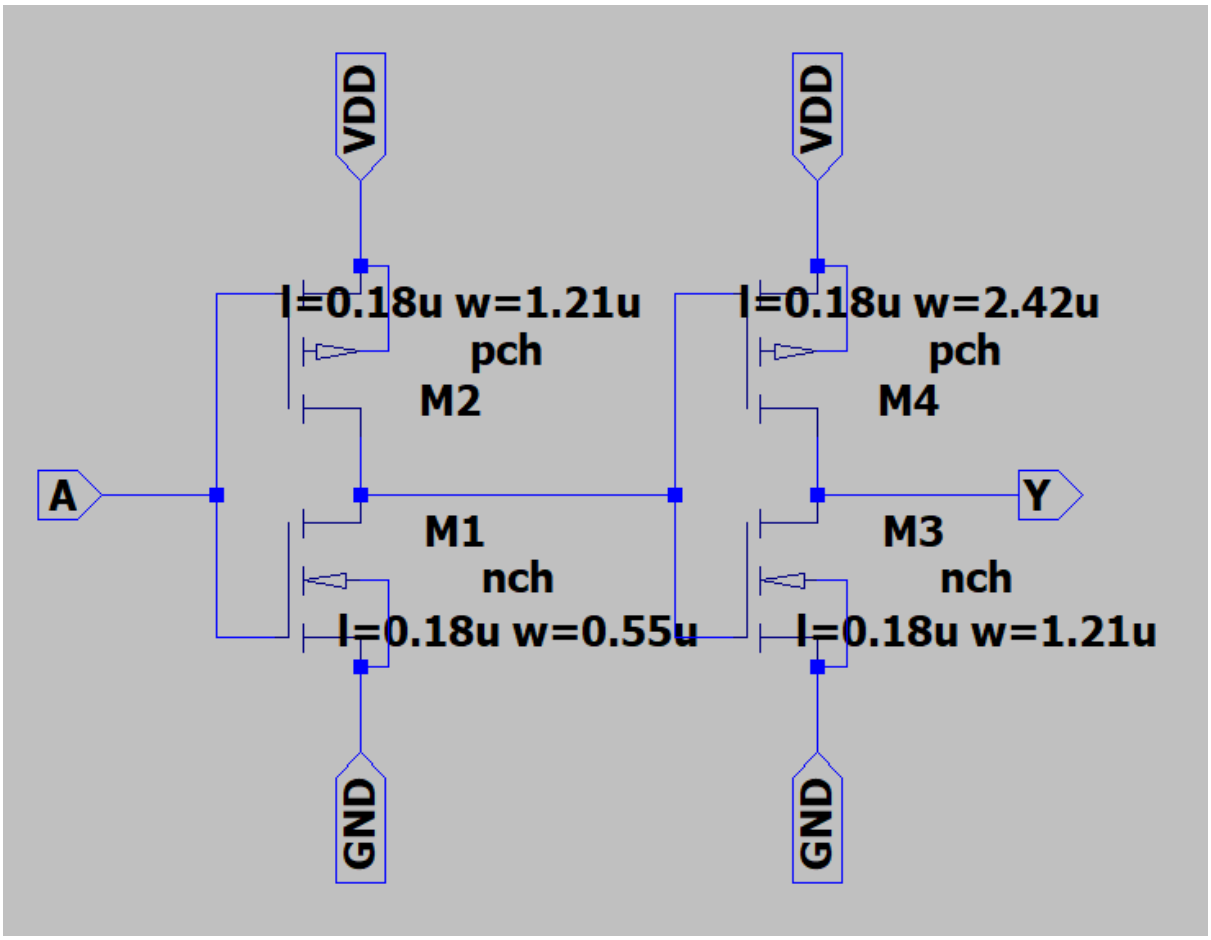


Figura D-3. Mux (mux21)

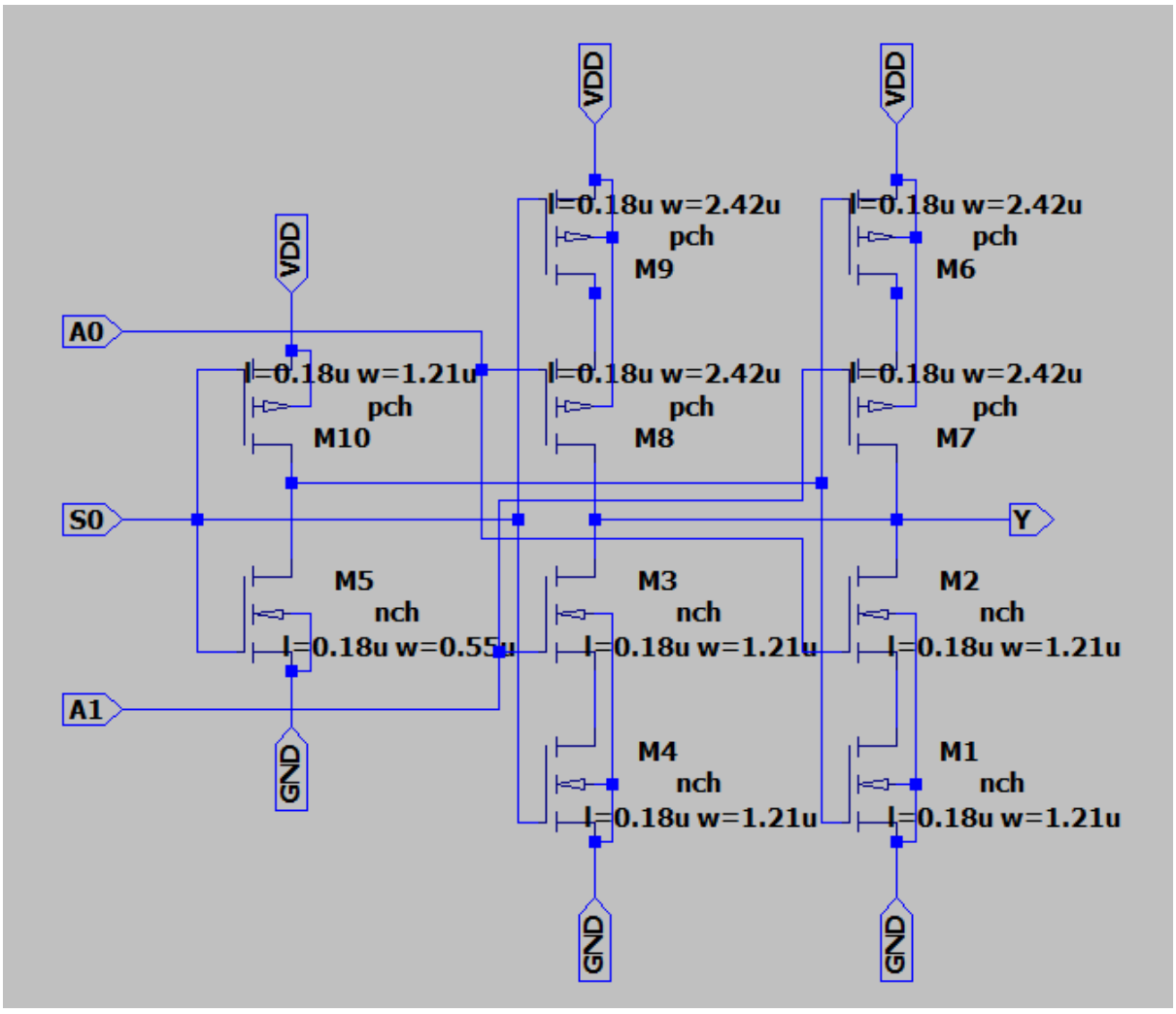


Figura D-4. Mux Não Inversor (mux21_ni)

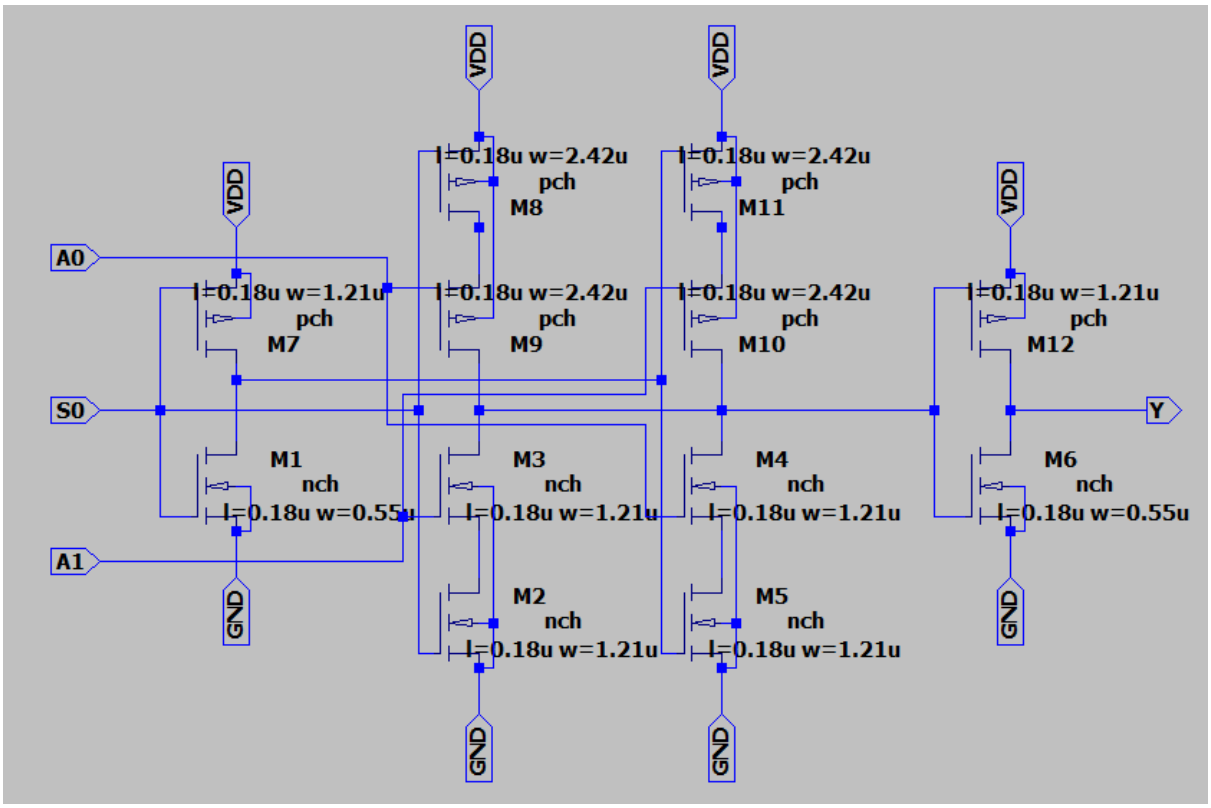


Figura D-5. Porta XNOR (xnor2)