Kaio Nikelisson de Lima Fernandes

Implementação de um Gerador de Números Aleatórios Gaussianos em hardware

Campina Grande, PB

2019

Kaio Nikelisson de Lima Fernandes

Implementação de um Gerador de Números Aleatórios Gaussianos em hardware

Trabalho de Conclusão de Curso (TCC) submetido à Coordenação de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande, *Campus* Campina Grande, como parte dos requisitos necessários para obtenção do título de Graduado em Engenharia Elétrica.

Universidade Federal de Campina Grande – UFCG

Centro de Engenharia Elétrica e Informática

Departamento de Engenharia Elétrica

Orientador: Marcos Ricardo Alcântara Morais, D. Sc.

Campina Grande, PB

2019

Kaio Nikelisson de Lima Fernandes

Implementação de um Gerador de Números Aleatórios Gaussianos em hardware

Trabalho de Conclusão de Curso (TCC) submetido à Coordenação de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande, *Campus* Campina Grande, como parte dos requisitos necessários para obtenção do título de Graduado em Engenharia Elétrica.

Trabalho aprovado. Campina Grande, PB, 11 de dezembro de 2019:

Marcos Ricardo Alcântara Morais, D. Sc.
Orientador

Gutemberg Gonçalves Júnior, D. Sc
Convidado

Campina Grande, PB
2019

Dedico esse trabalho à resiliência.
É provável que esse e muitos outros projetos
nunca teriam existido sem ela.

Agradecimentos

Primeiramente, agradeço a minha mãe. Obrigado por tudo. Sou só o eco da orquestra que você toca. Você é minha vida e o meu caminho.

Agradeço também à minha família, á minha segunda mãe - que a vida resolveu presentear na forma de tia: sou eternamente grato, Beniza; ao meu irmão, Romeu; e ao meu padrasto, Lindomar.

Sou grato ao amor que tem nome de Mariana. Pela casa compartilhada por 5 anos, pelos aconchegos e pelas conversas. Pelo nosso futuro.

Aos Manolos (Felipe, Ítalo, João, Lucas, Luiz e Rômulo), muito obrigado pela amizade. De verdade. Em especial, sou grato à João, que embarcou no mesmo navio que eu e não me deixou lutar as batalhas do curso sozinho.

Ao Armário, que me acolheu e compartilhou comigo todos os momentos bons e ruins da engenharia. Não sei como o destino conseguiu reunir pessoas tão incríveis.

Por meio de Marcos Morais e Gutemberg Júnior, os mestres que me guiaram pelos caminhos da microeletrônica, agradeço à todos os professores que colaboraram para a minha formação.

Ao Laboratório X-MEN e todos que fizeram parte dessa história, muito obrigado. Essa organização é parte integral do meu sucesso e tudo o que eu tenho é carinho por tudo e todos.

Gostaria de deixar os meus agradecimentos à NXP Semicondutores. Em especial, à César Dueñas que gentilmente me concedeu permissão para utilizar as ferramentas e os kits de desenvolvimento (TDKs) disponível na empresa para o desenvolvimento dessa pesquisa.

Por fim, um homem deve reconhecer seus santuários e, por isso, finalizo com um ode ao café e aos jogos. Às séries e aos animes. Foram essas coisas simples que me fizeram suportar momentos difíceis e clarear a mente fadigada pelos estudos.

"42."
(Guia do Mochileiro das Galáxias)

Lista de ilustrações

Lista de tabelas

Lista de Códigos

Lista de abreviaturas e siglas

GRNG	Gaussian Random Number Generator
URNG	Uniform Random Number Generator
ICDF	Inverse Cumulative Distribution Function
LFSR	Linear Feedback Shift Register
LUT	Look-up Table
MCM	Multiplierless Constant Multiplier
RTL	Register Transfer Level
fdp	Função Distribuição de Probabilidade
pdf	Probability Density Function
RNG	Random Number Generator
LCG	Linear Congruential Generator
HSM	Hierarchical Segmentation Method
SoC	System on Chip

Resumo

Distribuições aleatórias tem se tornado elementos fundamentais para o desenvolvimento da ciência em diversas área. Entretanto, a geração de números aleatórios em sistemas digitais foi e continua sendo um desafio na medida em que não é possível converter um sistema determinístico em uma máquina capaz de gerar eventos estocástico. Para vencer essas limitações, foram desenvolvidos diversos métodos que implementam pseudo-geradores de números aleatórios. Esse trabalho tem como objetivo implementar, em *hardware*, um gerador de números aleatórios gaussianos baseado no método de inversão e adotando duas arquiteturas: LUT e MCM. Para tal, foram desenvolvidos modelos matemáticos em Python para validar o método de geração e verificar a funcionalidade das implementações executadas em RTL. Em consequência, as duas arquiteturas foram implementadas em Verilog e verificadas por meio de simulações, obtendo valores de média e desvio padrão dentro do esperado. Além disso, as propostas foram sintetizadas utilizando tecnologias de 28nm e 45nm, o que mostrou considerável superioridade da arquitetura utilizando LUT em termos de frequência, área e potência. Não obstante os resultados obtidos, pode-se reconhecer o potencial de aplicação de estruturas de MCM em sistemas com multiplicações fixas que possuam menor complexidade.

Palavras-chave: GRNG. MCM. ICDF. Síntese.

Abstract

Random distributions have become fundamental elements to the science development in many areas. However, the random number generation in digital systems was and will continue to be big challenge considering that it's not possible to convert a deterministic system into a machine capable of generating stochastic events. To overcome these limitations, many methods were developed to implement pseudo-random number generators. This project aims to implement, in hardware, a Gaussian random number generator based in the inversion's method and adopting two main architectures: LUT and MCM. To accomplish that, mathematical models were developed in Python to validate the generation method and to functionally verify the RTL implementations. Consequently, the two architectures were build in Verilog and verified by digital simulations, achieving mean and standard deviation values as expected. Besides that, the proposals were synthesized using a 28nm and 45nm technology that ended presenting considerable LUT's implementation superiority regarding timing, area and power. Nonetheless the results obtained, it was possible to recognize MCM's structures application potential in system needing fixed multiplications with less complexity than the ones addressed in this project.

Keywords: GRNG. MCM. ICDF. Synthesis.

Sumário

1 Introdução

Ao se abordar um número aleatório por si só, pode-se defini-lo como um elemento amostrado de um conjunto de valores possíveis o qual possui uma determinada probabilidade de ser selecionado. (HAAHR, 2016) Isolados, os números aleatórios não parecem ter demasiada importância prática, no entanto são essenciais para o desenvolvimento da ciência quando interpretados como um conjunto e visualizados como uma distribuição probabilística.

As distribuições de probabilidade são funções matemáticas capazes de descrever fenômenos aleatórios por meio de atribuições probabilísticas. Em outras palavras, as distribuições probabilísticas materializam o comportamento de uma grupo de amostras atribuindo-as uma frequência de ocorrência em um determinado ensaio que acaba por se traduzir em sua probabilidade. Por assim ser, tais elementos matemáticos são capazes de modelar fenômenos de natureza randômica (sejam eles: biológicos, naturais, físicos, químico, entre outros) e, portanto, são amplamente utilizados em simulações que objetivam reproduzir os fenômenos supracitados.

Embora as distribuições probabilísticas sejam matematicamente tangíveis, a geração de números aleatórios em sistemas digitais foi e continua sendo um desafio na medida em que não é possível converter um sistema determinístico em uma máquina capaz de gerar eventos estocásticos. Atualmente, os geradores de números (pseudo) aleatórios adotam um amplo arsenal de técnicas para se obter uma população de amostras que reproduzam a distribuição desejada. (SCHRYVER et. al., 2012) Normalmente, distribuições uniformes são alcançadas por meio de transformações binárias (LCG, MRG, LFSR, etc.) e as distribuições não uniformes são obtidas a partir das distribuições uniformes utilizando-se uma das técnicas a seguir:

1. Transformação;
2. Rejeição de amostra;
3. Inversão;
4. Recursão.

Apesar das particularidades de cada um dos métodos, a técnica de inversão (ou ICDF) tem se tornado cada vez mais popular devido a sua capacidade de transformar uma distribuição uniforme em diversas outras a depender somente da função utilizada. O método é assim tão poderoso pois utiliza a função Q (*quantile function*) de forma que

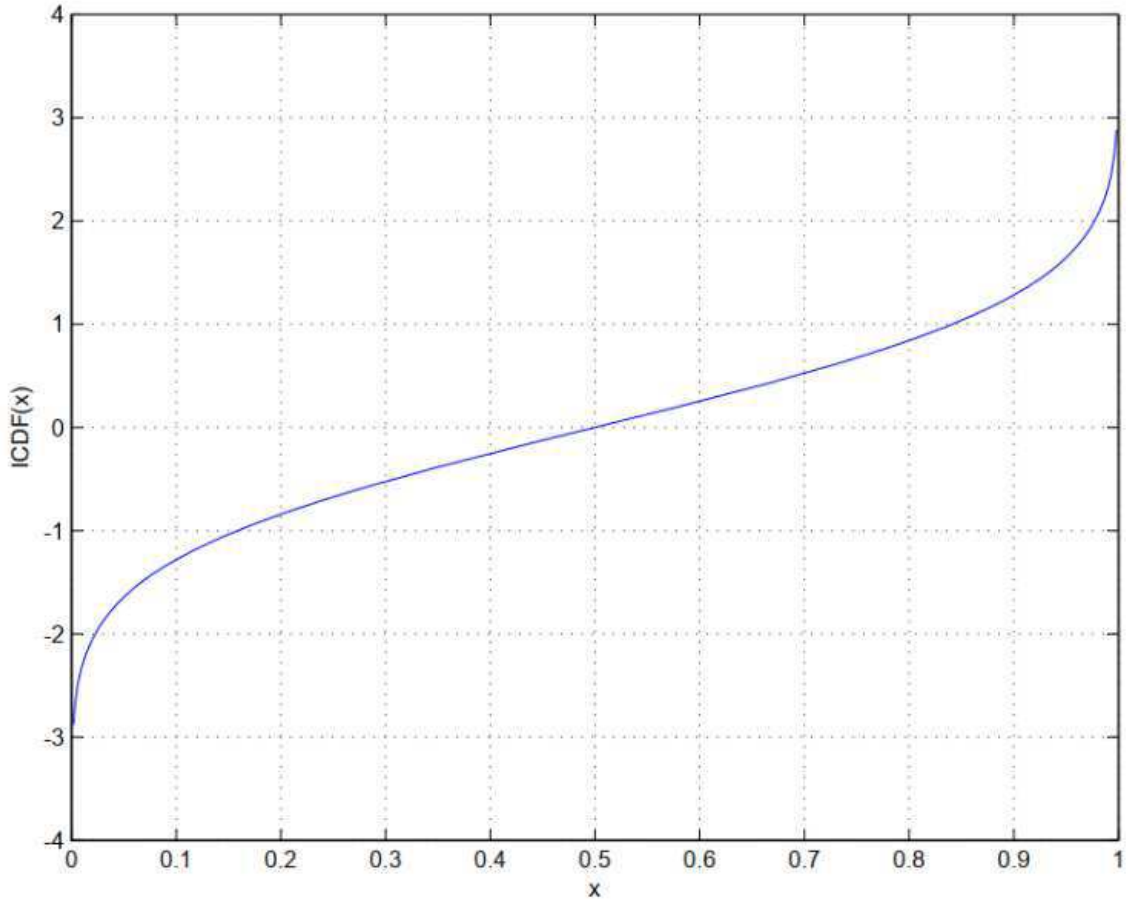


Figura 1 – Função Q Uniforme-Gaussiana (*probit function*).

seja possível mapear uma determinada variável X, de distribuição uniforme por exemplo, em uma nova função densidade de probabilidade, de natureza gaussiana ou exponencial. (CHEUNG et. al., 2007) A Figura 1 ilustra a função Q - mais conhecida como função *probit* - que converte uma distribuição uniforme em uma distribuição normal.

Atualmente, embora a maior parte desses métodos sejam aplicados em simulações utilizando CPUs de alto desempenho ou até mesmo em placas gráficas (GPUs), o cenário tecnológico tem mudado. Segundo Parhami (2010), o mercado de portáteis e o amplo direcionamento para o desenvolvimento de aplicações embarcadas com circuitos dedicados têm modificado as necessidades e tornado o baixo consumo cada vez mais importante no desenvolvimento de sistemas digitais.

Como consequência da especialização dos sistemas digitais (SoC), é cada vez mais recorrente a necessidade de simular fenômenos por meio de modelos randômicos de forma que seja possível validar a funcionalidade do produto pós-silício. Ademais, considerando a íntima relação da geração de números aleatórios com os circuitos dedicados de criptografia e segurança (cada vez mais relevantes no âmbito da tecnologia da informação e da indústria de semicondutores), é possível inferir o intrínseco papel dos *hardwares* de geração de

números aleatórios no atual mercado de circuitos integrados.

Por assim ser, o método do ICDF tem se tornado o principal meio pelo qual distribuições aleatórias são alcançadas tanto no âmbito de *software*, como no de *hardware*. Não obstante o seu potencial, computar a função de distribuição acumulada inversa (*Inverse Cumulative Distribution Function*, ICDF) exige cálculos extremamente complexos que inviabilizam sua implementação direta em sistemas digitais devido à exigência elevada de potência e área e a baixa latência total do sistema final.

Para tentar driblar as barreiras supracitadas, métodos de aproximação por partes e interpolação são utilizados para simplificar o cálculo da função inversa. (LEE et. al., 2009) Tal metodologia exige a utilização de *look-up tables* (LUT) e decodificadores que, apesar de inteligentes, ainda são bastante custosos para o sistema como um todo. Na tentativa de descobrir viabilizar novas arquiteturas para a geração de números aleatórios em *hardware*, esse trabalho tem como objetivo desenvolver um hardware dedicado de um gerador gaussiano de números aleatórios utilizando o método de inversão (ICDF) baseado em blocos de MCM (*Multiplierless Constant Multiplier*) como alternativa para as LUTs.

2 Objetivos

2.1 Objetivo Geral

O projeto tem como objetivo implementar, em *hardware*, um gerador gaussiano de números aleatórios baseado no método de inversão (ICDF) que se utiliza de técnicas de MCM (*Multiplierless Constant Multiplier*) como alternativa para as LUTs.

2.2 Objetivos Específicos

Figuram entre os objetivos específicos:

- Estudar e compreender o método de ICDF na geração de números aleatórios;
- Implementar um modelo ótimo de aproximação por partes da função *probit* - mais conhecida como a inversa cumulativa da função normal;
- Desenvolver um modelo matemático em linguagem de alto nível que melhor adapte os conceitos do ICDF aos atributos da lógica digital;
- Desenvolver uma arquitetura que substituísse as LUTs das abordagens tradicionais por estruturas de MCM;
- Descrever o modelo em RTL por meio de Verilog/SystemVerilog;
- Testar (verificando e validando) a funcionalidade do projeto final;
- Comparar o resultado final da arquitetura proposta com a arquitetura tradicional por meio de síntese lógica.

3 Referencial Teórico

3.1 Distribuição de Probabilidade

Segundo Feller (1968), a teoria matemática da probabilidade encontra propósito e forma quando aplicada à situações e experiências reais. É fácil explicar os conceitos que alicerçam a probabilidade quando tomamos como exemplo um simples dado justo de 6 faces. Nesse caso, se levarmos em consideração o arremesso único desse dado, podemos definir o espaço amostral desse experimento como $U = \{1, 2, 3, 4, 5, 6\}$, ou seja, todos os possíveis valores (amostras) que podem resultar da execução do experimento. Realizado um arremesso, entende-se como ponto amostral o resultado observado após a execução do experimento. Além disso, a teoria da probabilidade define o evento como outro elemento fundamental da sua existência. Ainda considerando o exemplo do dado, pode-se entender a obtenção da face 5 como um evento.

Tomando o dado como um objeto justo, é intuitivo dizer que a chance de se obter qualquer um dos números em suas faces é igual. Para constatar essa afirmação, jogar o dado 600 vezes para cima e documentar cada um dos resultados na forma de um histograma parece ser uma solução. Esse procedimento é usual e gera como produto uma função discreta que sintetiza o comportamento do experimento.

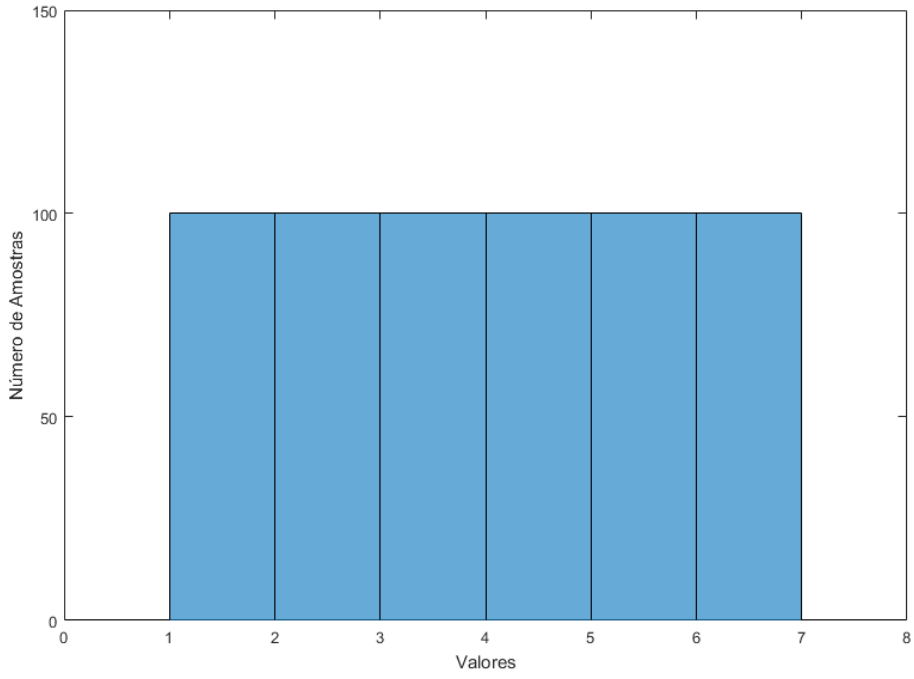


Figura 2 – Resultado teórico do experimento do dado arremessado 600 vezes.

É possível notar a partir da Figura 2 que todos os valores ocorrem o mesmo número de vezes e, portanto, o comportamento do arremesso de um dado é uniforme e a probabilidade de cada amostra é igual à $P(X = x) = 100/600 = 1/6$. A função da Figura 2 é conhecida como função massa de probabilidade.

Para espaços amostrais contínuos nos quais o número de amostras é infinito e não pode ser determinado, esse tipo de visualização se torna cada vez mais difícil. No entanto, o mesmo experimento pode ser reproduzido para esse tipo de variável aleatória e uma mesma curva pode ser traçada. O exemplo claro é a velocidade do vento em uma determinada direção. Nesses casos, a curva do comportamento da variável é conhecida como função densidade de probabilidade (fdp ou pdf, *probability density function*) e definida matematicamente pela Equação 3.1. (ORLOFF & BLOOM, 2014)

$$P(c \leq X \leq d) = \int_c^d f(x)dx \tag{3.1}$$

Ross (2010) ressalta que, diferente da função massa de probabilidade, o valor da função no ponto amostral não reflete sua probabilidade, visto que a chance de um exato número ocorrer em um espaço amostral infinito é nula. A Figura 3 apresenta uma determinada fdp (Distribuição de Rayleigh).

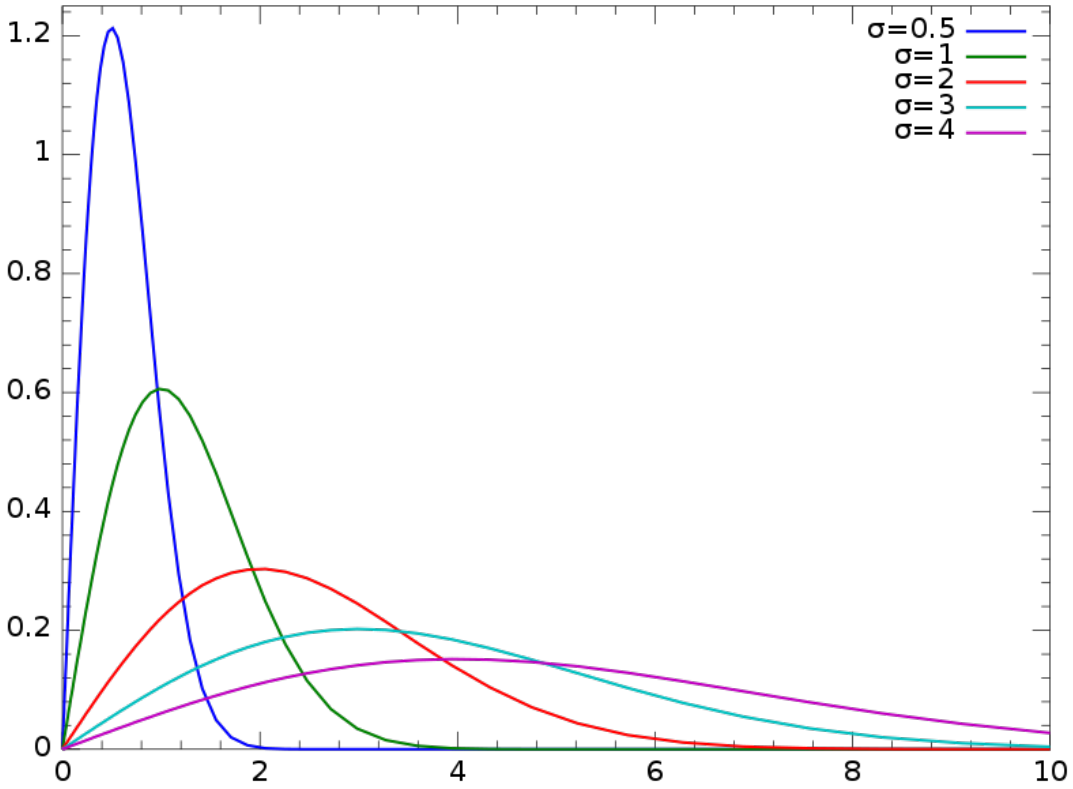


Figura 3 – Distribuição contínua de Rayleigh.

3.1.1 Distribuição Uniforme

A distribuição uniforme trata-se de uma das funções de probabilidade mais simples e populares. Uma variável X uniformemente distribuída em um intervalo (a,b) possui seus valores igualmente prováveis em todos os pontos entre a e b e valor constante da função densidade de probabilidade. (ROSS, 2010) A Equação 3.2 descreve sua função densidade de probabilidade que materializa-se na Figura 4.

$$f(x) = \begin{cases} \frac{1}{b-a} , & if\ a < x < b \\ 0 , & c.c. \end{cases} \tag{3.2}$$

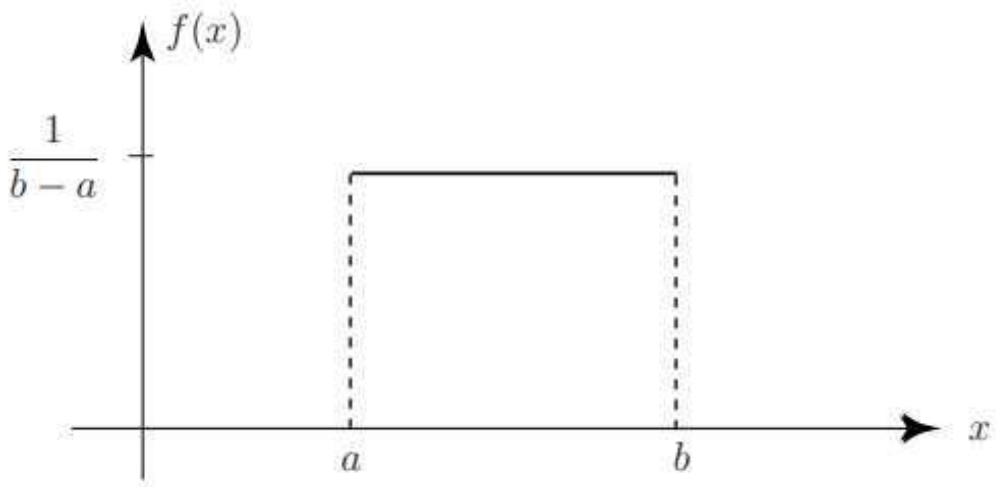


Figura 4 – Função genérica de uma distribuição uniforme.

O intervalo uniforme mais habitual trata-se do seu formato normalizado em que (a, b) = (0, 1). Nessa representação, a distribuição tem esperança igual a 0.5 e desvio padrão igual à 1/12.

3.1.2 Distribuição Normal (Gaussiana)

A distribuição normal ou Gaussiana foi inicialmente apresentada em 1733 pelo matemático francês Abraham DeMoivre que chegou em sua fórmula ao tentar descrever propriedades de distribuições binomiais, como: (a) qual a chance de se obter 60 coroas ao se arremessar uma moeda justa 100 vezes. Quanto maior o número de arremessos, mais e mais próximo o histograma do ensaio se delineava em uma distribuição normal. (LANE, 2013)

Mais tarde, Laplace (assim como Aleksandr Lyapunov) compreendeu a importância da distribuição normal quando expandiu as descobertas de DeMoivre mostrando que praticamente todos os eventos naturais tendiam de alguma forma para uma distribuição

normal - dando origem ao Teorema do Limite Central que rege a Teoria da Probabilidade. (ROSS, 2010)

Assim, por aparecer tão naturalmente num gigantesco acervo de fenômenos reais, a distribuição adotou coerentemente a alcunho de "normal". Em consequência dos fatos apresentados, a distribuição gaussiana é amplamente utilizada para modelar sistemas reais e/ou simular fenômenos, seja para validar teorias ou tecnologias.

Matematicamente, a função que descreve o comportamento de uma variável normal tem o formato de um sino centralizado no valor de sua esperança - que também determina o eixo vertical ao qual o sino é simétrico. Os atributos gráficos e probabilísticos dessa distribuição são integralmente definidos por dois parâmetros: sua média (μ) e seu desvio padrão (σ^2). A Figura 5 ilustra o formato de sino mencionado anteriormente e o impacto dos parâmetros no comportamento da função.

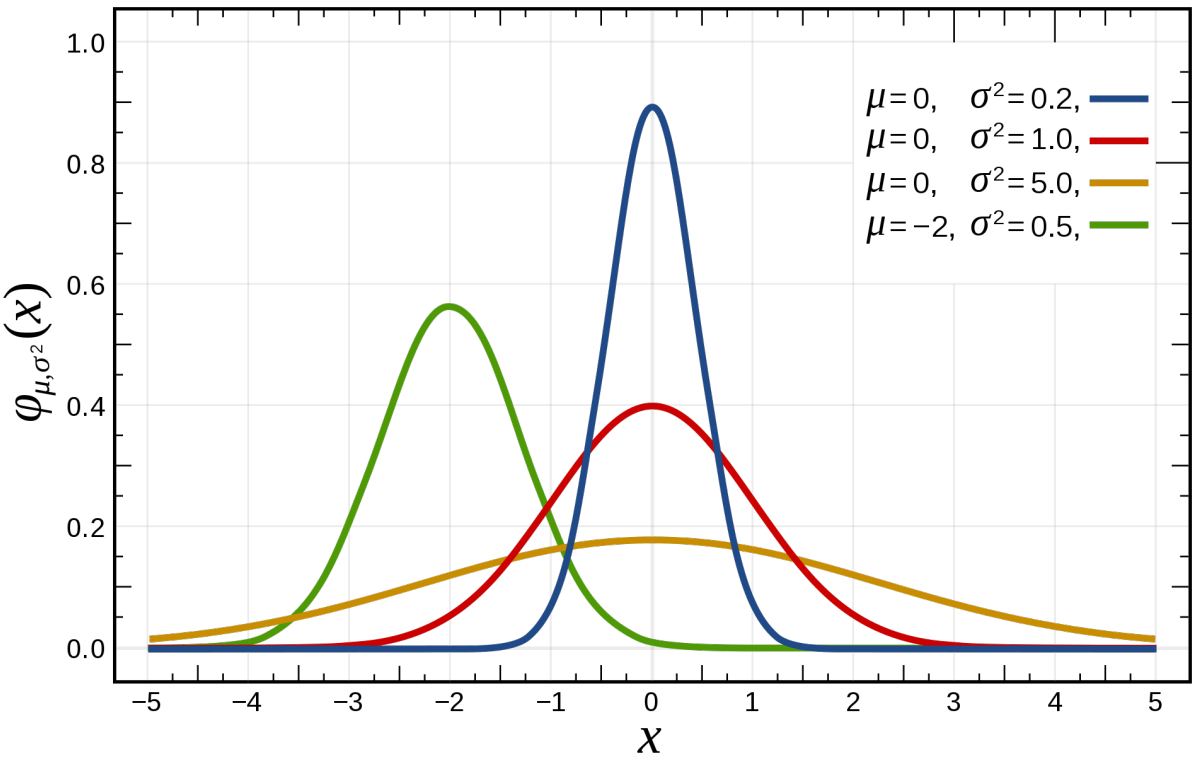


Figura 5 – Distribuição normal com diferentes valores de média e desvio padrão.

Por fim, a famigerada função normal pode ser calculada utilizando a Equação 3.3.

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} exp\left[-\frac{(x-\mu)^2}{2\sigma^2} \right] \tag{3.3}$$

3.2 Gerador de Números Aleatórios Uniformes (URNG): Tausworthe

Os geradores de números aleatórios uniformes foram as primeiras estruturas desenvolvidas pela ciência com a capacidade de gerar distribuições pseudo-aleatórias, visto que, apesar de possuírem atributos estatísticos de uma distribuição igualmente provável, eram computadas por sistemas determinísticos e possuíam previsibilidade (periodicidade).

Em geral, as técnicas para geração de números aleatórios uniformemente distribuídos se baseiam no simples processo da transformação de um estado para um outro estado, podendo ser representado matematicamente da seguinte forma: `estado` $\leftarrow$ `f(estado)`. Os detalhes da transformação utilizada são responsáveis pelos atributos de independência e distribuição uniforme da população concebida. (OWEN, 2013)

Como define L'Ecuyer (2005), os geradores aleatórios (RNG) podem ser sintetizados em uma estrutura matemática do formato `(S,` μ`, f, U, g)`, em que:

S: o conjunto finito de possíveis estados.

μ**:** distribuição aleatória usada para determinar o estado inicial (s_0), ou semente.

f: a função de transição ou transformação.

U: o espaço representativo de saída.

g: a transformação de saída (`S` $\rightarrow$ `U`).

Além disso, L'Ecuyer (2005) ainda destaca o fato das distribuições uniformes que partem desse modelo possuírem periodicidade intrínseca a qual influencia diretamente na qualidade do gerador - geradores com periodicidade (p) muito pequena, repetem seu espaço amostral com frequência e, portanto, são altamente previsíveis.

Os métodos de transformação do espaço S amplamente utilizados em aplicações e difundidos no ambiente acadêmico utilizam fórmulas de recursão para gerar novos estados a partir de estados passados. Entre eles, os mais célebres são o LCG (*Linear Congruential Generators*) e a versão digital do LCG, LFSR (*Linear Feedback Shift Register*). (WUERTZ & DUTANG, 2009)

Pelo fato dos computadores e demais processadores utilizarem álgebra binária, o LFSR (também conhecido como gerador de Tausworthe) tem se destacado como o principal gerador de números aleatórios pelas suas propriedades digitais e pela elevada qualidade estatística de sua saída. Como a própria denominação do método já entrega, a transformação se trata de uma soma (linear) entre os valores passados (realimentação) de

modo a gerar os valores futuros por meio do descarte paulatino dos elementos passados (deslocamento registrado). A Equação 3.4 formaliza a transição de estado.

$$x_n = (x_{n-q} + x_{n-k}) \quad mod\ 2 \tag{3.4}$$

O valores **q** e **k** revelados na Equação 3.4 determinam o grau do trinômio característico utilizado como modelo para a criação do gerador, ou seja, indicam quais elementos devem ser utilizados na realimentação - em que o **k** representa o maior grau. A Figura 6, composta de equações e diagrama, ilustra um exemplo de como utilizar a fórmula de recorrência para obter os valores de x_n.

$$x^7 + x^3 + 1 = 0$$
$$b_{n+7} \oplus b_{n+3} \oplus b_n = 0$$
$$b_{n+7} = b_{n+3} \oplus b_n$$
$$b_n = b_{n-4} \oplus b_{n-7}$$

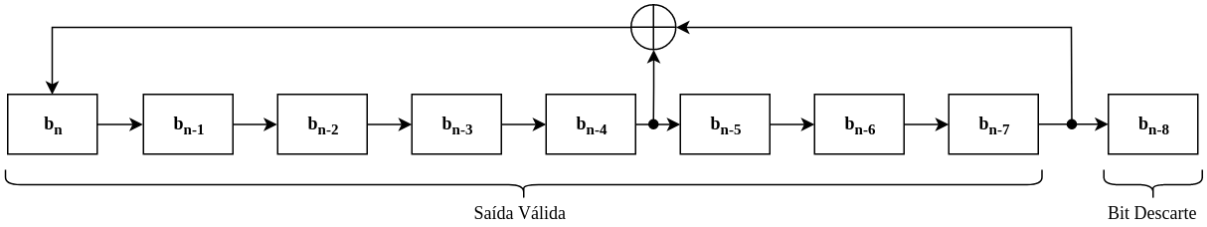


Figura 6 – Exemplo de LSFR com {L = 8, q = 7, k = 3, s = 1}.

Além disso, como é possível notar pela fórmula de recorrência, os valores do LFSR pertencem ao GF(2). Foi para que fosse possível converter um conjunto de elementos binários que L'Ecuyer (1994) definiu a transformação **g** como estrutura fundamental de um gerador aleatório. Tomando L como o tamanho máximo da coleção de registros do LFSR, a transformação **g** costuma normalizar a saída uniforme (**U**(0,1)) e adota a solução formal ilustrada pela Equação 3.5. (GILLE-GENEST, 2012)

$$u_n = \sum_{j=1}^{L} x_{ns+j-1} * 2^{-j} \tag{3.5}$$

A variável **s** introduzida na Equação 3.5 é conhecida como "passo"e indica a quantidade de vezes que a recorrência deve ser aplicada para que o próximo valor aleatório do gerador seja considerado válido.

Não obstante a eficiência e a consistente periodicidade dos geradores de Tausworthe, modelos e simulações que exigem elevada granularidade e um número exorbitante de

iterações não tem suas necessidades supridas, pois a periodicidade de tais arquiteturas continua não sendo suficiente. Na tentativa de reverter essa limitação, Tezuka (1989) e Wang & Compagner (1993) propuseram o emprego de geradores combinados de forma que fosse possível maximizar o período total das amostras obtidas.

Segundo L'Ecuyer (1996), se propriamente configurado, o período de um conjunto de LFSR combinados pode alcançar um período de repetição próximo ao produto do período máximo de cada LFSR separadamente - que é definido como $p = 2^{l-1}$, sendo **l** o comprimento da saída. Para que o máximo período seja alcançado, os trinômios que regem cada um dos geradores individuais devem ser primitivos e primos entre si. Dessa forma, considerando três Tausworthe (P_0, P_1, P_2) com parâmetros $\{k_0, q_0, s_0, k_1, q_1, s_1, k_2, q_2, s_2\}$, o resultado da soma (ou operação bit a bit de uma ou-exclusivo) das três estruturas, se propriamente configuradas, alcança período máximo de $(2^{k_0} - 1)x(2^{k_1} - 1)x(2^{k_2} - 1)$.

Além de realizar vários apontamentos teóricos sobre o geradores de distribuição uniforme, L'Ecuyer (1996) também foi responsável por desenvolver um algoritmo mais simples (*QuickTaus*) para a implementação das estruturas de LFSR baseadas em trinômios. Considerando A, B e C vetores de tamanho **l**, com A inicializado por s_{n-1}; $r = k - q$; e que C é umas máscara composta por K bits iguais à 1 nos bits mais significativos (MSB) e $(l - k)$ bits iguais à 0 nos bits menos significativos (LSB), o algoritmo consiste nos seguintes passos:

1. B ← q-bit de A deslocado pra esquerda;
2. B ← $A \oplus B$;
3. B ← (k - s)-bit de B deslocado pra direita;
4. A ← A & C;
5. A ← s-bit de A deslocado pra esquerda;
6. A ← $A \oplus B$.

Porém L'Ecuyer (1996) estabelece algumas condições para que o algoritmo entregue o resultado desejado. Tais condições estão sintetizadas na Equação 3.6.

$$\begin{cases} P(z) = z^k - z^q - 1\ ,\ \textit{trinômio com}\ 0 < 2q < k \\ 0 < s < k - q < k \leq L \\ MDC(s, 2^k - 1) = 1 \end{cases} \tag{3.6}$$

As descobertas de L'Ecuyer facilitaram não somente a implementação de geradores aleatórios baseados em LFSR por meio de *software*, como otimizaram os *hardwares* que

dependiam integralmente de modelos digitais de recorrência para funcionar. A Figura 7 representa o algoritmo de L'Ecuyer implementado em RTL por Cheung *et. al.* (2007).

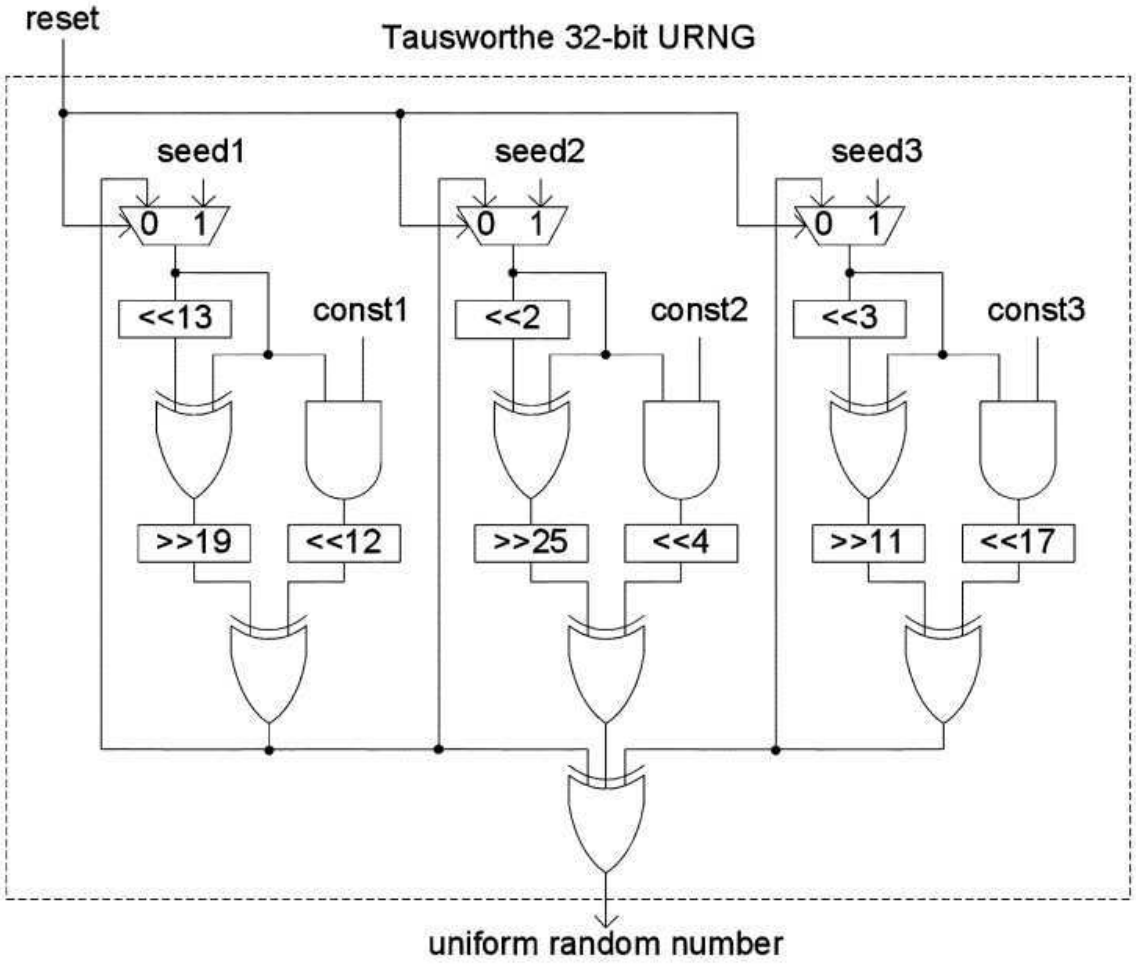


Figura 7 – Tausworthe combinado com parâmetros $(k_0 = 31, q_0 = 13, s_0 = 12)$, $(k_1 = 29, q_1 = 2, s_1 = 4)$ e $(k_2 = 28, q_2 = 3, s_2 = 17)$ implementado em RTL por Cheung *et. al.* (2007).

3.3 Gerador de Números Aleatórios Gaussianos (GRNG)

Na literatura, poucas são as formas de se obter amostras de uma sequência de distribuição normal que não por meio de métodos indiretos. Como já fora especificado anteriormente, a maior parte dos sistemas de recorrência eficientes geram apenas distribuições uniformes, fazendo com que as demais distribuições de probabilidade (incluindo a distribuição de Gauss) sejam obtidas utilizando como base uma população de valores uniformemente distribuídos. (SCHRYVER et. al., 2012) O estado da arte da geração de números aleatórios não uniformes faz uso frequente de quatro técnicas de conversão:

1. Transformação;

2. Rejeição de amostras;
3. Inversão;
4. Recursão.

As estruturas de GRNG baseadas nas técnicas de transformação utilizam funções matemáticas e definições da probabilidade de modo que seja possível converter variáveis uniformemente distribuídas em uma distribuição normal. O algoritmo de Box-Muller (mais célebre do seu tipo) utiliza funções transcendentais (senos, cossenos, logaritmos e raízes) e relações fundamentais da probabilidade para converter, simultaneamente, duas variáveis uniformes em duas variáveis normalmente distribuídas.

Os métodos de recursão se baseiam em uma característica bastante particular das distribuições aleatórias. Por exemplo, a generalização de Fibonacci afirma que se x_1 e x_2 são variáveis uniformemente distribuídas (U(0,1)), a operação $(x_1 + x_2)\ mod\ 1$ resultará na mesma distribuição (U(0,1)). Wallace (1996) baseou seu trabalho no fato de que, se $x_1, x_2, ..., x_k$ são i.i.d. em N(0,1), a transformação $1/K^{1/2} \sum_{k=1}^{K} x_k$ resulta em uma mesma distribuição N(0,1). Logo, os métodos recursivos utilizam um conjunto de amostras pertencentes à uma determinada distribuição e às aplicam em uma transformação para gerar um novo grupo de amostras de mesmo tamanho e referentes à mesma curva densidade de probabilidade.

Existem uma infinidade de métodos fundamentados na técnica de rejeição de amostra. Não obstante essa variedade, Thomas *et. al.* (2007) sintetiza matematicamente o método quando considera que: sendo $y = f(x)$ um função com integral finita, C o grupo de pontos (x, y) sob a função, e Z uma subseção na qual C está contigo, o método de rejeição sorteia pontos (x, y) uniformemente de do subgrupo de elementos pertencentes à Z até que (x, y) façam parte de C. O método é considerado de rejeição pois até que a exigência seja cumprida, os demais valores sorteados são sumariamente descartados.

O método da Inversa da Função de Distribuição Cumulativa (*Inverse Cumulative Distribution Function*, ICDF) utiliza a função Q (*quantile function*) de forma que seja possível mapear uma determinada variável X em uma nova função densidade de probabilidade. Não obstante essa seja uma de suas utilidades, a sua real função é descrever um valor de x para o qual todas as variáveis menores ou iguais terão probabilidade p de serem encontradas, ou seja, produz a inversa da cdf (*Cumulative Distribution Function*). A Equação 3.7 a seguir formaliza o conceito.

$$F_X(x) := Pr(X \leq x) = p \tag{3.7}$$

Assim sendo, os métodos de inversão com ICDF são generalistas e podem ser aplicados no desenvolvimento de RNG para qualquer distribuição de probabilidade que

seja monótona. Para tal, deve-se determinar qual a distribuição desejada e determinar a sua ICDF para mapeamento a partir de variáveis uniformemente distribuídas.

Apesar de todos os métodos apresentarem pontos positivos e negativos, o método de inversão da função de densidade cumulativa (ICDF) tem se estabelecido nos últimos anos como o padrão para aplicações dedicadas à *hardware* não só pela qualidade superior da distribuição gerada, mas também pelos resultados de performance que vem evoluindo paulatinamente por meio de adaptações nas arquiteturas propostas. Por assim ser, o tópico a seguir entrará em mais detalhes sobre o ICDF e as arquiteturas mais clássicas da literatura.

3.3.1 ICDF Gaussiano

O método da inversa da função densidade cumulativa (ICDF) orientada para a produção de uma distribuição normal utiliza a função *probit* para mapear um conjunto de amostras uniformes ($\mathbf{U}(0,1)$) em variáveis com comportamento normal ($\mathbf{N}(0,1)$). A Figura 8 ilustra a função supracitada para uma distribuição de desvio padrão máximo igual 4σ e a Equação 3.8 formaliza sua estrutura matemática.

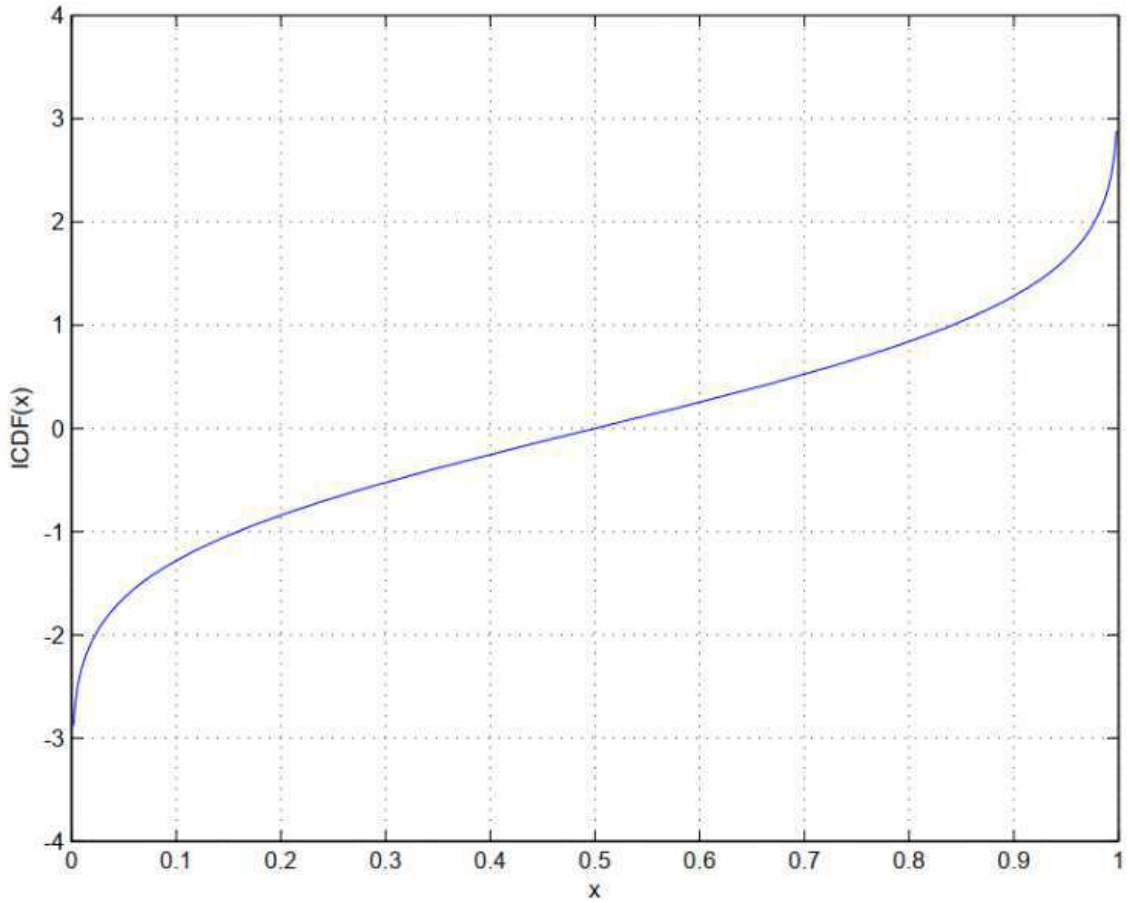


Figura 8 – Função *probit*.

$$probit(p) = \sqrt{2}erf^{-1}(2p - 1) \tag{3.8}$$

Em implementações dedicadas à *hardware*, a principal barreira diz respeito à complexidade envolvida no cálculo da função *probit*, dado que a função erro ($f(x) = erf(x)$) é obtida por meio da determinação de uma integral e caracteriza uma função transcendental - classe de funções não primordiais que são difíceis de implementar e são altamente dispendiosas, seja em área, frequência ou potência.

No entanto, a avaliação direta de função não é a única maneira de se estimar o valor de uma determinada curva matemática. Cheung *et. al.* (2007) apresentou como alternativa para esse problema a segmentação da função *probit* e a interpolação dos segmentos por meio de polinômios. Tais polinômios seriam armazenados em uma *look-up table* (LUT) e devidamente endereçados quando seus segmentos fossem necessários para o cálculo.

Assim, ao invés de avaliar uma função transcendental, essa abordagem possibilita que o RTL avalie apenas polinômios utilizando somadores e multiplicadores. A desvantagem mais óbvia desse modelo se trata dos erros de aproximação do processo de interpolação dos segmentos da função *probit*, mas considerando que o sistema digital é limitado pela precisão do comprimento da palavra de saída, se propriamente computadas, os erros podem ser considerados irrelevantes no resultado final. A Figura 9 foi retirada do trabalho de Cheung *et. al.* (2007) e esboça a ideia por meio de diagramas generalistas.

Em geral, os gerador de números aleatórios uniformes escolhidos para essas aplicações são os geradores Tausworthe combinados. Além disso, o modelo de segmentação da função gaussiana segue os princípios da teoria de segmentação hierárquica (HSM, *Hierarquical Segmentation Method*) a qual é mais profundamente discutida na Seção 3.4. Os polinômios são avaliados com os coeficientes extraídos das LUTs utilizando o método de Horner. Entretanto, para manter o modelo o mais coerente possível com os cálculos do ICDF, o sistema de endereçamento da LUTs proposto por Cheung *et. al.* exige a implementação de estruturas de *barrel shifter* que acabam por ser extremamente onerosas no que diz respeito aos atributos de síntese do RTL (área, potência e frequência).

Na buscar de melhorar a performance dos geradores gaussianos baseados no método do ICDF com segmentação da função *probit*, Gutierrez *et. al.* (2012) apresentou, como alternativa ao endereçamento utilizando *barrel shifters*, o uso de um endereço fixo da saída do URNG para o cálculo realizado por meio dos polinômios do segmento. Isso só é possível pois o modelo foi normalizado e, embora a sequência utilizada na avaliação do polinômio não tenha sido exatamente aquela que foi sorteada no URNG, por ser normalizada ela continua fazendo parte do espectro do segmento selecionado e integralmente aleatória. Ademais, como os bits de cálculo eram também utilizados para o endereçamento em determinados segmentos, Gutierrez *et. al.* (2012) inclui uma máscara para reduzir a

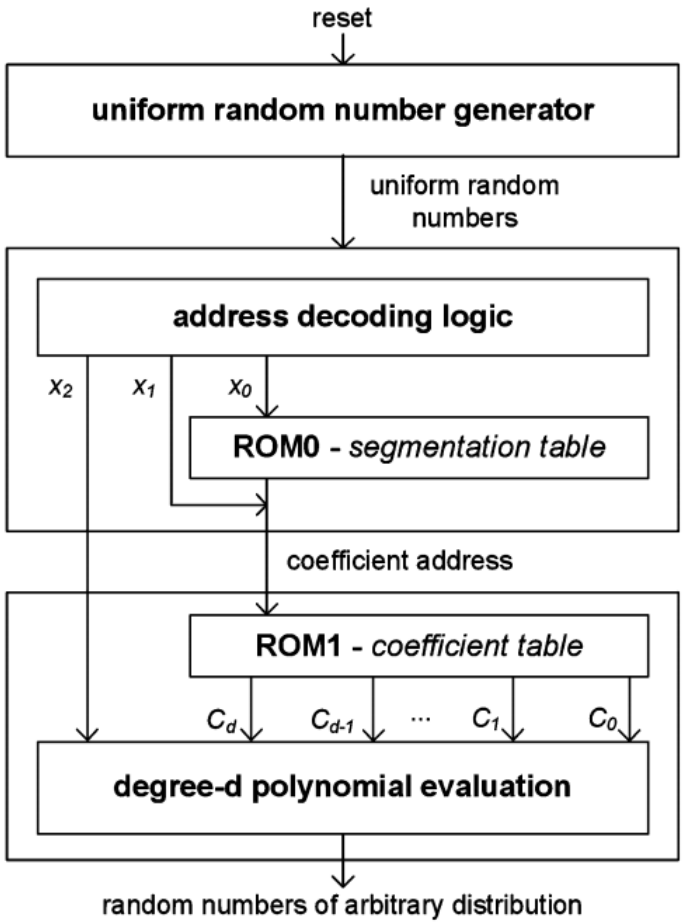


Figura 9 – Arquitetura ICDF desenvolvida por Cheung *et. al.* (2007).

correlação entre os valores. A Figura 10 foi retirada do trabalho do autor supracitado e ilustra sua abordagem em diagrama de blocos.

3.4 Método de Segmentação Hierárquico (HSM, *Hierarchical Segmentation Method*

Métodos eficientes para avaliar numericamente funções complexas tratam-se de um problema recorrente na ciência e de um tema fecundo em artigos científicos. O panorama atual estabelece que, para sistemas que exigem precisão fixa e elevada taxa de transferência, geralmente implementados em *hardware*, métodos não-iterativos devem ser utilizados, sendo eles: tabela de consultas (LUTs), método de adição de tabelas, aproximação polinomial e aproximações racionais. (LEE *et. al.*, 2009)

Aproximações polinomiais e aproximações racionais representam o estado da arte atual. Ambos utilizam teoria de séries para aproximar determinadas funções matemáticas utilizando apenas polinômios - formas que podem ser avaliadas utilizando-se apenas

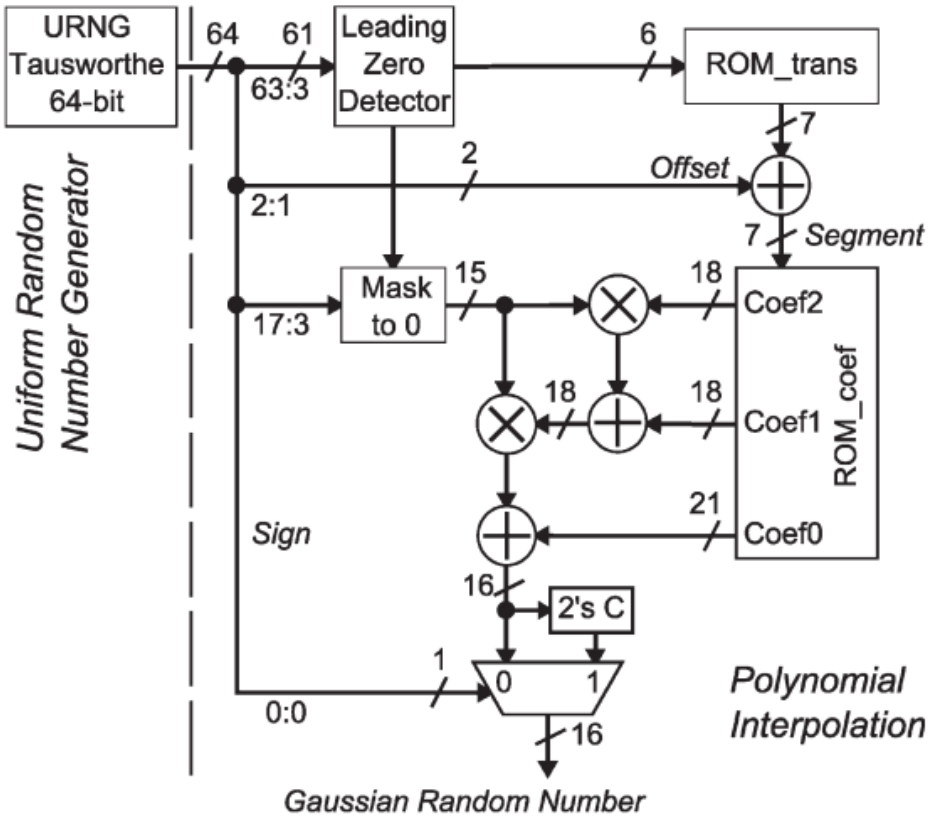


Figura 10 – Arquitetura ICDF desenvolvida por Gutierrez *et. al.* (2012)).

estruturas de soma e multiplicação. Independente da escolha, ambos utilizam métodos de segmentação para seccionar a função e obter melhores aproximações com menor custo operacional (aritmético e de armazenamento).

A segmentação uniforme e não hierárquica é o modelo mais comum. Além de ser o mais simples, fornece bons resultados quando a função é monótona e possui poucas não linearidades, mas falha em eficiência quando o comportamento da função não é ideal. Lewis (1994) apresentou o conceito de segmentação hierárquica quando propôs dois níveis de fracionamento uniforme: um externo e de intervalo fixo; e outro interno com o número de segmentos variando à depender do comportamento da função - logo, funções não-lineares exigiriam um número superior de divisões internas.

Utilizando o conceito de hierarquia, outros modelos de segmentação que melhor se adaptavam a determinados comportamentos matemáticos foram sendo propostos. Modelos com hierarquias flexíveis e com densidades variáveis resultam em um menor número de segmentos, mas não são adequados no que tange sua aplicação em sistemas dedicados de *hardware*, visto que exigem um endereçamento personalizado que dificilmente seria generalizado com pouca complexidade. (SASAO *et. al.*, 2004)

Lee *et. al.* (2009) organiza os modelos de segmentação que melhor se aplicam em

sistemas digitais e os categoriza em 4 (quatro) tipos: US, $P2S_L$, $P2S_R$ e $P2S_{LR}$. No modelo US (*Uniform Segmentation* ou Segmentação Uniforme), os intervalos possuem tamanhos iguais. O $P2S_L$ e $P2S_R$ (*Power of 2 Segmentation* ou Segmentação por Potência de 2) tem o mesmo comportamento mas partem de lados diferentes: $P2S_L$ começa da esquerda (*left*) e tem os seus intervalos ampliados em potência de 2; já o $P2S_R$ realiza o procedimento inverso (*right*). Por fim, o $P2S_{LR}$ aplica o processo partindo das duas pontas e utiliza o centro como ponto de encontro. A Figura 11, confeccionada por Lee *et. al.* (2009), ilustra a aplicação dos 4 métodos.

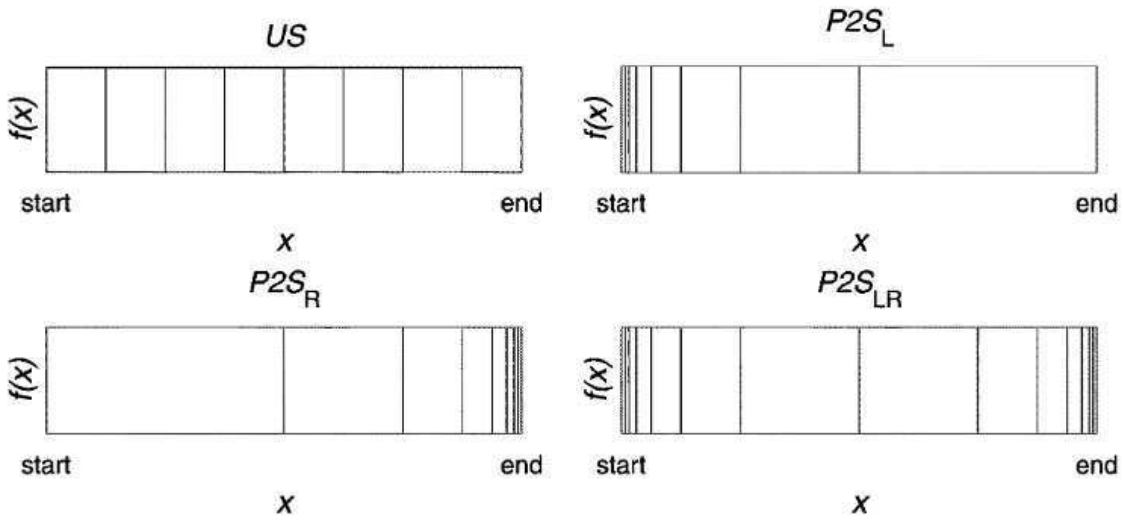


Figura 11 – Metodologias de segmentação propostas por Lee *et. al.* (2009) para aplicações em *hardware*.

Os modelos escolhidos por Lee *et. al.* (2009) foram selecionados pelo fato de possibilitarem a identificação natural do segmento do qual um determinado número faz parte por meio da decodificação utilizando a representação binária. As Equação 3.9, 3.10 e 3.11 mostram a quantidade máxima de segmentos que uma entrada de B_x bits consegue endereçar para cada uma das metodologias.

$$s_i = 2^{B_{x_i}},\ se\ US \tag{3.9}$$

$$s_i \leq B_{x_i} + 1\ ,\ se\ P2S_L/P2S_R \tag{3.10}$$

$$s_i \leq 2B_{x_i},\ se\ P2S_{LR} \tag{3.11}$$

A segmentação uniforme (US) com número de seções múltiplas de 2 permite a identificação binária por meio da combinação representativa de cada bit que compõe a palavra. Os métodos de potência de dois baseiam-se na propriedade do deslocamento binário como forma de duplicação. Logo, a quantidade de zeros e uns que precedem a aparição do primeiro elemento oposto partindo do bit mais significativo (MSB) possibilita o

endereçamento das seções fracionadas por meio do P2S. A Tabela 1, adaptada do trabalho de Lee *et. al.* (2009), ilustra o procedimento de endereçamento por P2S. Além disso, a formulação matemática que entrega a numeração exata do segmento pode ser consultada nas Equações 3.12, 3.13 e 3.14.

Número do Segmento	**Entrada** (B_x)
0	**0 0 0 0 0** | 0 0 0 ∼ **0 0 0 0 0 0** | 1 1 1
1	**0 0 0 0 1** | 0 0 0 ∼ **0 0 0 0 1** | 1 1 1
2	**0 0 0 1** | **0** 0 0 0 ∼ **0 0 0 1** | **1** 1 1 1
3	**0 0 1** | **0 0** 0 0 0 ∼ **0 0 1** | **1 1** 1 1 1
4	**0 1** | **0 0 0** 0 0 0 ∼ **0 1** | **1 1 1** 1 1 1
5	**1** | **0 0 0 0** 0 0 0 ∼ **1** | **1 1 1 1** 1 1 1
6	**1 0** | **0 0 0** 0 0 0 ∼ **1 0** | **1 1 1** 1 1 1
7	**1 1 0** | **0 0** 0 0 0 ∼ **1 1 0** | **1 1** 1 1 1
8	**1 1 1 0** | **0** 0 0 0 ∼ **1 1 1 0** | **1** 1 1 1
$9 = s_0 - 1$	**1 1 1 1 0** | 0 0 0 ∼ **1 1 1 1 0** | 1 1 1

Tabela 1 – Exemplo de segmentação $P2S_{LR}$ para $B_x = 8$. A tabela também pode ser interpretada para $P2S_L$ e $P2S_R$.

$$P2S_{L_addr} = \begin{cases} B_{x_i} - LZD(x_i),\ MSB(x_i) = 0 \\ B_{x_i},\ MSB(x_i) = 1 \end{cases} \tag{3.12}$$

$$P2S_{R_addr} = \begin{cases} 0,\ MSB(x_i) = 0 \\ LOD(x_i),\ MSB(x_i) = 1 \end{cases} \tag{3.13}$$

$$P2S_{LR_addr} = \begin{cases} B_{x_i} - LZD(x_i),\ MSB(x_i) = 0 \\ B_{x_i} + LOD(x_i) - 1,\ MSB(x_i) = 1 \end{cases} \tag{3.14}$$

Os métodos supracitados podem ser aplicados em conjunto com o conceito de hierarquia. Para aplicar a ideia do HSM em um determinada função, deve-se seguir o seguinte algoritmo: (a) determinar a função; (b) intervalo de aproximação; (c) precisão da entrada e saída (número de bits); e (d) o método de aproximação polinomial (Chebyshev, Splines, etc.). Determinado isso, deve-se escolher a quantidade de hierarquias que serão aplicadas e os devidos métodos de segmentação para cada hierarquia. Por fim, um erro máximo deve ser escolhido de forma a guiar a iterações e o número de segmentos que deve ser aplicado em cada um dos níveis. A Figura 12, retirada de Lee *et. al.* (2009), ilustra o resultado final de uma segmentação de dois níveis ($P2S_L$ e US) na função $f_2(x) = cos^{-}1(x)$.

3.5 Interpolação de Chebyshev

A interpolação de Chebyshev é um método que permite que pontos amostrais no intervalo de [-1,1] sejam aproximados em uma função matemática por meio dos polinômios

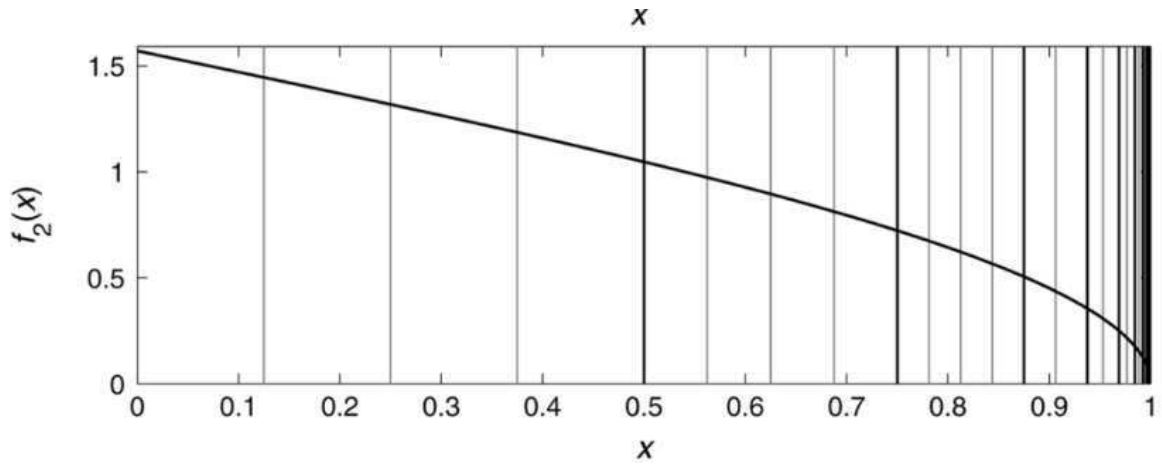


Figura 12 – Função $f_2(x) = cos^{-}1(x)$ segmentada em 2 hierarquias ($P2S_L$ e US).

de Chebyshev. Embora possua limitações no que diz respeito aos intervalos de aproximação, o método é amplamente utilizado pois interpola pontos minimizando os erros da Equação 3.15.

$$E(x) \equiv f(x) - P(x) = \frac{1}{(n+1)!} f^{n+1}(\xi_x) \prod_{i=0}^{n} (x - x_i) \tag{3.15}$$

Sabendo que dificilmente conseguiria minimizar os impactos do termo $f^{n+1}(\xi_x)$, Chebyshev focou-se em reduzir a influência da diferença presente no produtório e, utilizando-se das definições dos polinômios de Chebyshev, o matemático russo conseguiu provar que, caso a interpolação fosse realizada utilizando as raízes dos seus polinômios, o erro máximo da aproximação seria definido pela Equação 3.16.

$$E(x) \equiv |f(x) - P(x)| = \frac{1}{2^n (n+1)!} \max_{-1 \leq x \leq 1} |f^{n+1}(t)| \tag{3.16}$$

Chebyshev não só minimizou o impacto do produtório como provou que sua otimização seria a melhor possível de se alcançar. Os polinômios de Chebyshev são definidos pela recorrência descrita na Equação 3.17 (com $T_0 = 1$ e $T_1 = x$) e as suas raízes podem ser encontradas utilizando-se a Equação 3.18.

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x) \tag{3.17}$$

$$x_i = cos\left(\frac{2i+1}{2n+2}\pi\right) \tag{3.18}$$

Em experimentos ou situações nas quais o intervalo de aproximação está contido em $[a, b] \neq [-1, 1]$, é possível escalar os valores obtidos de forma que eles estejam coerentes

com o que propõem a minimização de Chebyshev. A conversão permite mapear s(-1) = a e s(1) = b; para tal, é realizada por meio da Equação 3.19.

$$x_i = \frac{b+a}{2} + \frac{b-a}{2} cos\left(\frac{2i+1}{2n+2}\pi\right),\ para\ i = 0, 1, ..., n \tag{3.19}$$

3.6 Multiplierless Constant Multiplier (MCM)

Durante anos e anos, operações básicas são submetidas à estudos extensos que possibilitem encontrar métodos e/ou algoritmos capazes de otimizar suas execuções. No passado, esses métodos eram desenvolvidos com o intuito de reduzir o esforço humano e avançar a ciência e a teoria matemática. Atualmente, os algoritmos são conjecturados para o ambiente da computação e buscam potencializar metodologias de computação focadas em álgebra binária, de modo que seja possível aproveitar cada vez mais o recurso computacional disponível - acelerando cálculos e simulações.

A multiplicação é um exemplo perfeito. Não obstante seja uma das operações mais comuns, sua execução exige lógica dedicada e, quando sintetizada em um *hardware* dedicado, consume área, potência e requer um tempo considerável para expedir o resultado esperado. Na Tabela 2, retirada do trabalho de Aswale *et. al.* (2015), podem ser analisados métodos e multiplicação implementados em RTL e seus atributos de síntese.

Multiplier Type	**Power Dissipation (mW)**	**Area (μm^2)**	**Delay (ns)**
Array multiplier (convetional)	8.909	83160	8.693
Baugh-Wooley Multiplier	10.880	87912	9.620
Modified Baugh-Wooley Multiplier	10.170	62646	6.219

Tabela 2 – Comparação pós síntese de multiplicadores realizada por Aswale *et. al.* (2015).

No Egito antigo, a multiplicação era feita por meio da multiplicação por dois ou, como era mais intuitivo para época, duplicação. O principal fator para que as multiplicações fossem executadas dessa maneira diz respeito às "limitações"da linguagem matemática utilizada pelos egípcios. Para eles, era difícil representar o conceito e as regras da multiplicação, mas a duplicação de um número podia ser feita apenas por duplicar o símbolo. (DEIF, 2014)

Logo, para multiplicar dois números era necessário gerar uma tabela com um número base em uma coluna e os múltiplos de 2 na outra. No processo de geração da tabela, o número base era consecutivamente multiplicado por 2 até que a coluna de potência de dois ultrapasse o menor dos números presente na multiplicação inicial. Em seguida, o menor número era decomposto em múltiplos de 2 e os valores correspondentes da

decomposição eram somados. Esse procedimento não só consolidou a multiplicação egípcia, como também provou que todos os números poderiam ser representados na base dois e alicerçou o sistema binário utilizado na computação moderna. A Tabela 3 e a Equação 3.20 exemplificam o procedimento.

Identificador	**Mul. de 2**	**Base**
n_1	1	9
n_2	2	18
n_3	4	36
n_4	8	72
n_5	16	144
n_6	32	288
n_7	64	576

Tabela 3 – Método de Ahmes aplicado para 9×47.

$$\begin{aligned} 9 \times 47 &= 9 \times (32 + 8 + 4 + 2 + 1) \\ &= (9 \times 32) + (9 \times 8) + (9 \times 4) + (9 \times 2) + (9 \times 1) \\ &= 288 + 72 + 36 + 18 + 9 \\ &= 423 \end{aligned} \tag{3.20}$$

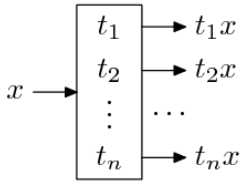
Baseado no método de Ahmes, um algoritmo mais eficiente para computar multiplicações em base binária foi desenvolvido e batizado como: Algoritmo do Camponês Russo. A proposta desse algoritmo é encontrar o resultado da operação de multiplicação com o menor número de deslocamentos e somas possíveis - utilizando como base do o maior número disponível na equação. Em suma, divide-se o menor número por 2 em uma coluna (descartando-se o resto) e multiplica-se o maior por dois na coluna adjacente. Linhas com números pares na coluna com o menor número são descartadas; o resultado é a soma das linhas remanescentes da coluna de maior valor. A Tabela 4 reproduz o resultado do exemplo anterior utilizando o algoritmo do Camponês Russo.

Par	**Menor Numero**	**Maior Numero**
o	9	47
x	4	94
x	2	188
o	1	376
Total		$188 + 376 = 423$

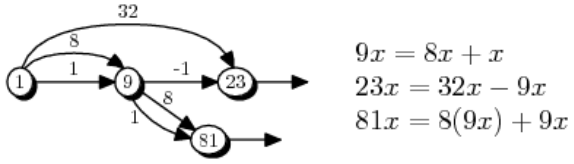
Tabela 4 – Tabela exemplo do algoritmo do Camponês Russo para $9x47$.

Devido às claras vantagens da utilização de tais algoritmos para implementações em *hardwares* dedicados e aplicações computacionais, novas metodologias de multiplicação e algoritmos foram desenvolvidos. No que tange multiplicações por números pré-determinados,

geralmente extraídos por meio de LUTs (*look-up tables*) e utilizados por circuitos generalista de multiplicação, surgiu o conceito de blocos de multiplicação sem multiplicadores (MCM, *Multiplierless Constant Multiplier*). Tais estruturas paralelizam multiplicações por meio da composição de uma árvore multiplicativa otimizada que recebe um valor e o multiplica por um conjunto de constantes utilizando apenas somas e deslocamentos (duplicação ou multiplicação por 2). A Figura 13 foi extraída do trabalho de Voronenko & Püschel (2007), o qual discorre sobre o desenvolvimento de algoritmos que fornecem as estruturas das árvores multiplicativas, assim como o conceito e geração de RTLs que implementam tais estruturas - sumarizando o conceito do MCM.



(a) Diagrama lógico do RTL do MCM.



(b) Exemplo de um ramo da árvore multiplicativa do MCM.

Figura 13 – Sumário do MCM extraído de Voronenko & Püschel (2007).

4 Metodologia

4.1 Modelo

O modelo foi integralmente desenvolvido em Python 3.6 e pode ser destrinchado em duas grandes etapas: (a) criação da função para calcular a melhor segmentação para a função cumulativa inversa da distribuição gaussiana; e (b) aplicação do modelo obtido para elaboração do modelo em alto nível do gerador de números aleatórios.

4.1.1 Modelagem matemática da segmentação (HSM)

Para a primeira etapa, foi desenvolvida uma função (`hsmComplete`), a qual tem a sua estrutura especificada no Código 1.

```
1 def hsmComplete (n, IntEnd, PS2, U, aproxPoints, degree, \
2     maxError, addrMax)
```

Código 1 – Cabeçalho da função `hsmComplete()`.

A primeira tarefa executada pela função supracitada consiste em segmentar o intervalo fornecido no formato de *turple* pelo argumento `IntEnd`: `IntEnd = [2**(-53), 0.5]`, por exemplo. A primeira hierarquia de segmentação é sempre P2S e pode ser configurada em $P2S_L$ ou $P2S_R$ a partir do argumento `PS2`, sendo $PS2_L = 1$ e $PS2_R = 0$. O primeiro argumento (`n`) define o número de segmentos a serem calculado para a primeira hierarquia.

Após definir os intervalos da seção P2S, a função aplica a segmentação uniforme (US) nas seções da hierarquia P2S, definindo os limites dos intervalos internos e finalizando o processo de segmentação hierárquica, visto que para a implementação do GRNG, apenas dois níveis são o suficiente. O número de segmentos internos é definido pelo argumento `U`, sendo: (a) U = 0, não possui hierarquia interna; (b) U = 1, otimiza o número de segmentos internos de acordo com o intervalo e o erro máximo; e (c) $U \geq 2$: designa o número exato de segmentos que devem ser computados.

Durante o processo de segmentação, mais especificamente no cálculo da segunda hierarquia, a função em `hsmComplete()` aplica a aproximação de Chebyshev a cada dos intervalos obtidos para garantir que os valores escolhidos ou o processo de otimização em andamento respeitam/respeita as exigências do sistema (erro máximo de aproximação em cada segmento).

Logo, para que o resultados fossem satisfatórios, tomou-se o cuidado para que todas as `aproxPoints`-amostras no intervalo do segmento em estudo fossem consideradas raízes do polinômio de Chebyshev de mesmo grau - como apontado na Seção 3.5 do Referencial Teórico. Além disso, as raízes foram aplicadas na Equação 3.19 como forma de escalar os intervalos e garantir que estivessem de acordo com as limitações do método proposto.

Por fim, antes de aplicar os valores de x e y da curva *probit* na função `chebfit()`, já implementada nas bibliotecas do Python 3.6, os valores dos pontos interpolados e, consequentemente, o intervalo interpolado, foi normalizado para valores de [0,1]. A normalização é fundamental visto que, além de garantir o funcionamento da função do Python, potencializa os cálculos do GRNG quando implementado em RTL. Para o caso onde o número de segmentos internos é fixo ($U \geq 2$), os passos do algoritmo que dizem respeito à interpolação e a validação dos segmentos internos está sumarizada a seguir.

1. Dois laços de repetição `for` administram a criação dos segmentos: o primeiro itera sobre o número de segmentos externos; e o segundo, sobre o número de segmentos internos de modo a criá-los;
2. Cada iteração do segundo `for` produz um segmento dentro do segmento externo selecionado pelo primeiro `for`;
3. Define-se, então, os limites do segmento interno que está sendo gerado;
4. Em seguida, são criados `aproxPoints`-raízes (`eX`) de Chebyshev escaladas para dentro dos limites estabelecidos no passo anterior;
5. Como referência para a etapa de interpolação, são calculados os valores reais da função *probit* (`rY`) avaliados nas raízes previamente computadas utilizando a função `ppf()` da biblioteca SciPy;
6. Os valores do eixo x (raízes do polinômio de Chebyshev no intervalo sobre análise) são normalizados em [0,1], tornando-se `n_eX`;
7. Com todos os dados prontamente processados, é realizada a interpolação do segmento em 3 passos:
    a) Na função `chebfit()`, disponível em Python, aplicam-se os valores normalizados de x (`n_eX`) junto com os valores reais do eixo y da função *probit* (`rY`);
    b) Os coeficientes de Chebyshev (`c_cheb_n`) computados no passo anterior são convertidos em coeficientes de polinômios tradicionais. As Equações 4.2, 4.3 e 4.4 exemplificam o caso para uma aproximação de 2º grau na qual a_0, a_1 e a_2 são os coeficientes obtidos na interpolação de Chebyshev ($p_{cheby}(x) =$

$a_0T_0 + a_1T_1 + a_2T_2$).

$$p(x) = c_0x^2 + c_1x + c_2 \tag{4.1}$$

$$c_0 = 2a_2 \tag{4.2}$$

$$c_1 = a_1 \tag{4.3}$$

$$c_2 = a_0 - a_2 \tag{4.4}$$

c) Por fim, os coeficientes passam por uma transformação para serem aplicados ao intervalo tradicional do segmento (sem normalização). Esse resultado serviu apenas para validação matemática e não é utilizado no modelo de RTL, dado que é mais vantajoso o uso dos coeficientes normalizados.

8. Após computados os coeficientes, calcula-se, ponto a ponto, o erro máximo da interpolação do segmento. Caso a função do cálculo do erro retorne um valor maior do que o valor máximo exigido, o modelo com `n` de segmentos externos e `U` segmentos internos é imediatamente descartado.

9. Caso a interpolação atenda às exigências de erro máximo, os seus limites e os coeficientes resultantes do processo são armazenados para que sejam posteriormente retornados pela função `hsmComplete()`.

Como mencionado anteriormente, o algoritmo acima aplica-se somente quando $U \geq 2$. Para $U = 1$, a função continua o processo de segmentação interno incrementando o número de segmentos da hierarquia uniforme (US) sempre que a interpolação com grau desejado de polinômios (`degree`) não atende às exigências de erro máximo (`maxError`).

O argumento `addrMax` é utilizado pela função para garantir que a segmentação obtida pelo HSM otimizado ou pelo HSM com segmentos internos fixos tem endereçamento viável considerando um determinada precisão ou número de bits de entrada.

Por fim, a Tabela 5 sintetiza as variáveis retornadas pela função `hsmComplete()`.

4.1.2 Modelo do Gerador Gaussiano (GRNG)

A segunda parte do modelo diz respeito à geração da simulação em alto nível do gerador de números aleatórios gaussiano (GRNG). O desenvolvimento do protótipo em Python 3.6 tem como objetivo validar matematicamente a segmentação obtida e os métodos utilizados pela arquitetura para aproximar a função cumulativa inversa da distribuição normal.

Assim sendo, o modelo pode ser substanciado em X processos: (a) geração de uma amostra uniforme; (b) desmembramento do bits que serão usados para aritmética, sinal e endereçamento; (c) cálculo do valor real do ICDF para a amostra sorteada; (d) cômputo do

Variável	Descrição
`xU`	Vetor com os limites dos segmentos externos e internos do HSM.
`coefs`	Array de array contendo os coeficientes para os segmentos não normalizados.
`coefs_n`	Array de array contendo os coeficientes para os segmentos normalizados.
`meanError`	Vetor com a média do erro de interpolação em cada segmento.
`UList`	Vetor com o número de segmentos internos para cada segmento externo.
`nUSeg`	Similar a \texttt{UList}, usado para cálculos internos.
`impossibleAddrFlag`	Sinalizador que indica falha ou sucesso do processo de HSM.
`maximumErrors`	Vetor composto pelo erro máximo de aproximação por segmento.
`flagError`	Vetor contendo um sinalizador de erro máximo ultrapassado para cada segmento.

Tabela 5 – Descrição detalhada das variáveis retornadas pela função `hsmComplete()`.

segmento contemplado do HSM (endereço da LUT/MCM); (e) mascaramento e avaliação do ponto utilizando o polinômio interpolado resultante do processo de HSM e interpolação anterior; (f) inclusão do sinal (complemento de 2); (g) reprodução do procedimento k-vezes e avaliação dos parâmetros estatísticos da distribuição obtida (média e desvio padrão).

Como o intuito inicial era apenas validar o sistema de geração de variáveis aleatórias, a produção de valores uniformemente distribuídos foi realizada por meio da função nativa do Python 3.6: `randrange()`, sendo o menor valor igual a 0 e o maior, 9007199254740991 (ou $2^{53} - 1$). Com o auxílio da função `bin()`, o número resultante foi convertido para base binária e, em seguida, separado na parcela de endereçamento (o número binário integral, exceto o MSB), na parcela para aritmética (os quinze últimos bits) e no bit de sinal (o MSB).

Em seguida, o número integral e a parcela de aritmética são transformada em representação de ponto fixo e o valor real do número sorteado é obtido com a função `ppf()` - completando a etapa (c). No estágio (d), o endereçamento é realizado por meio de um contador de zeros líderes, ou contador de zeros que precedem o primeiro um. Esse modelo é utilizado para identificar o segmento P2S primário, como propõem Lee *et. al.* (2009). Para a hierarquia US interna, o último bit (LSB) indica qual dos dois segmentos internos deve ser escolhidos - no modelo isso é feito apenas adicionando-se ou não uma unidade na soma de todos os itens da lista `UList` até o endereço proposto para o $P2S_L$ na Equação 3.12.

A primeira parte do item (e) foi implementada enumerando-se a quantidade de uns consecutivos que precediam o primeiro 0. Se o número fosse maior do 35 (o último número antes da parcela dos 53 bits dedicados à aritmética), o bits seguintes eram mascarados para 0 conforme descreve Gutierrez *et. al.* (2012). Logo após, o valor é aplicado à função `polyval()` junto com os coeficientes identificados para que a função *probit* seja avaliada conforme interpolação do modelo.

Finalmente, se o MSB do número uniforme sorteado no item (a) for igual a 1, aplica-se o compĺemento de 2 no resultado obtido a partir do modelo supracitado; caso contrário, o número permanece positivo - concluindo o passo (f). Para que fosse possível calcular os parâmetros estatísticos do gerador aleatório em Python, os itens de (a)-(f) são reproduzidos `repTimes`-vezes e, só então, calcula-se a média, o desvio padrão e cria-se o histograma da distribuição, fechando todos os processos necessários para a confecção do modelo.

Ademais, é importante destacar que, mesmo em alto nível, a maior parte da etapas foi executada com as variáveis em representação binária na tentativa de manter o modelo o mais coerente possível. Algumas funções de Python 3.6 que colaboraram para a tarefa foram: `bin()`, `int()` e `zfill()`.

4.2 RTL: URNG e GRNG

Essa seção dedica-se a descrição das técnicas e métodos utilizados para o desenvolvimento dos RTLs que constituem esse trabalho. Além disso, a seção apresenta as propostas de arquiteturas a serem implementadas, justificando as escolhas. A discussão dos resultados obtidos pode ser analisada no Capítulo 5.

4.2.1 Gerador Uniforme de Tausworthe (URNG)

Considerando as vantagens das técnicas digitais de geração de números uniformes (LFSR) quando dedicadas em aplicações de sistemas discretos (computadores e similares), além dos ótimos resultados de periodicidade documentados em artigos, o algoritmo de combinado de Tausworthe foi o gerador de números aleatórios uniformes (URNG) adotado nesse projeto.

O desenvolvimento do gerador foi completamente fundamentado no algoritmo proposto por L'Ecuyer (1996) - e previamente detalhado na Seção 3.2. Assim, baseado nas configurações que obtiveram máxima distribuição segundo as pesquisas teóricas de L'Ecuyer (1996), esse projeto utilizou dois gerados de 32 bits compartilhando os mesmos parâmetros, mas alimentados por sementes (*seeds*) diferentes. Os valores utilizados em cada um dos LFSR que compõem o Tausworthe combinado (L = 32, J = 3) podem ser consultados nas Equações 4.5, 4.6 e 4.7.

$$\{k_0, k_1, k_2\} = \{31, 29, 28\} \tag{4.5}$$

$$\{q_0, q_1, q_2\} = \{13, 2, 3\} \tag{4.6}$$

$$\{s_0, s_1, s_2\} = \{12, 4, 17\} \tag{4.7}$$

Seguindo rigorosamente os passos propostos pelo algoritmo de L'Ecuyer (1996), o *hardware* obtido é exatamente igual ao da Figura 7 implementado por Cheung *et. al.* (2007). No entanto, sabendo que o Tausworthe nada mais é do que um conjunto de LFSR com os resultados combinados por meio de uma porta XOR, é possível realizar algumas simplificações na implementação e generalizar o RTL permitindo que ele seja parametrizável. A simplificação é ilustrada na Figura 14, na qual um exemplo considerando $\{L, k, q, s\} = \{8, 7, 3, 2\}$ dissocia os dois termos da última operação do algoritmo de Tausworthe ($A = A \oplus B$) de modo que seja possível visualizar a equação de simplificação mais facilmente.

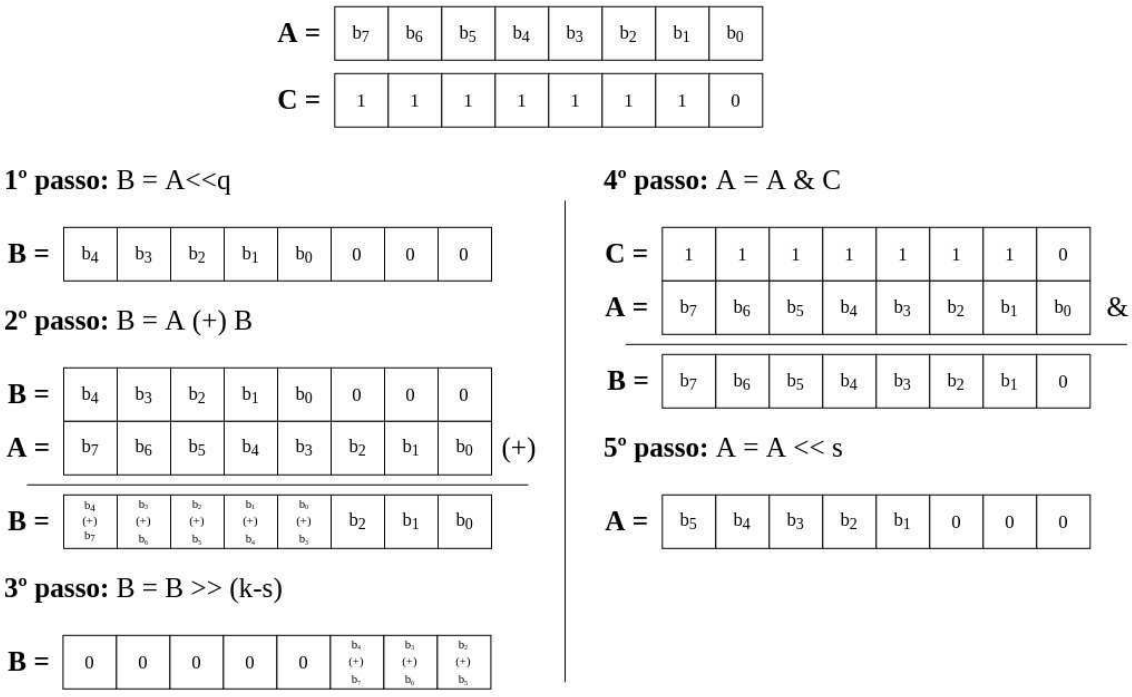


Figura 14 – Cinco primeiros passos de L'Ecuyer (1996) para a aplicação do LFSR com termos separados (lado esquerdo e direito).

Como é possível perceber nos passos do lado esquerdo (1-3) da Figura 14, o resultado final das operações do algoritmo de L'Ecuyer e, consequentemente, das operações de LFSR, respeitando as condições das Equação 3.6, sempre converge para um número composto por $(k - s)$ zeros e uma cauda formada pela XOR entre os q últimos bits de A, e os q bits após os q últimos. Por outro lado, as etapas 4 e 5 detalham o estado final da segunda parcela

que deve compor a soma (XOR) final ilustrada na Figura 15. Como é possível constatar a partir das operações do lado direito da Figura 14, a disposição terminal do vetor A assume sempre $s + (L - k)$ zeros na seção menos significativa, seguida pelos primeiros $L - (s + (L - k)) = (k - s)$ valores originais de A que sucedem os $(L - k)$ dígitos iniciais. Os zeros presente no início do vetor derivam do deslocamento da operação AND com C (que possui os $(L - k)$ primeiros bits iguais a zero) e o deslocamento em s bits para a esquerda do passo 6. Os MSBs são apenas efeito da propagação das operações mencionadas - é ainda possível notar que o vetor não começa do primeiro valor de A original, pois a máscara anula o efeitos desses bits.

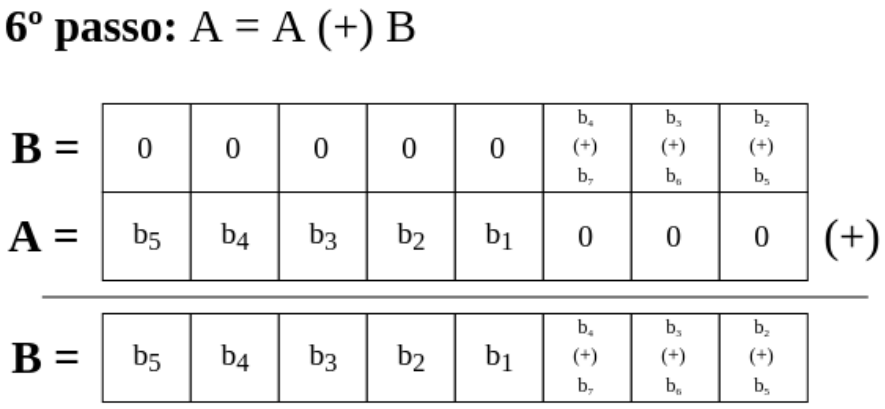


Figura 15 – Passo final de L'Ecuyer (1996).

Como se torna evidente na Figura 15, os 3 passos iniciais geram a primeira e parcela e os dois seguinte, a segunda. Além disso, é possível provar que não há intersecção entre essas duas parcelas (desde que as Equações 3.6 sejam respeitadas), dado que B sempre possui $(k - s)$ zeros à esquerda (MSBs) e A, $(s + (L - k))$ zeros à direita (LSBs). Se somarmos $(k - s) + (s + (L - k)) = L$, que é o número exato de bits do vetor completo, comprovando que não existe cruzamento entre valores que das duas parcelas - considerando que $b_x \oplus 0 = b_x$. Consequentemente, tomando $r = (k - q)$, a implementação de um LFSR pode ser alcançada aplicando-se a Equação 4.8.

$$s_i = \Big\{ A\Big[(L-1-s) : (L-k)\Big],\ A\Big[(L-1) : (k-s)\Big] \oplus A\Big[(L-1-q) : (r-s)\Big] \Big\} \quad (4.8)$$

É importante ressaltar que a Equação 4.8 foi formulada para linguagens de programação as quais possuem o seu primeiro elemento referenciado pelo índice zero - como o C e o Python. Assim sendo, ao invés de aplicar a arquitetura padrão do Tauworthe combinado, foi utilizado a generalização descrita pela Equação 4.8 para implementar um módulo de Tausworthe parametrizável com J = 3. Entre as possíveis configurações do bloco, estão: (a) L, tamanho da palavra; (b) k, maior grau do trinômio; q, menor grau do

trinômio; e s, passo de amostragem do LFSR. Como J = 3, as 3 últimas variáveis devem ser configuradas em trio.

Finalmente, para ambas as arquiteturas do GRNG, foram instanciados 2 módulos de Tausworthe com L = 32 e os parâmetros com máxima distribuição detalhadas nas Equações 4.5, 4.6 e 4.7.

4.2.2 Arquitetura tradicional (LUT)

Para esse trabalho, foi adotado como método para a geração números aleatórios gaussianos o algoritmo da inversão (ICDF). Como mencionado na Seção 3.3.1, a técnica de inversão utiliza uma função matemática que é capaz de mapear um grupo de variáveis (x) uniformemente distribuídas em um conjunto de variáveis com distribuição normal (y). A Equação 4.9 generaliza a definição.

$$y = f(x) \tag{4.9}$$

Apesar da técnica ser extremamente simples quando implementada em *software*, sua implementação em *hardware* é desafiadora, visto que avaliar funções transcendentais (como é o caso das inversas cumulativas) exige módulos excessivamente onerosos do ponto de vista de um RTL. Para reverter isso, o atual estado da arte dos geradores de distribuições não-uniformes aplicam a segmentação da ICDF utilizando o HSM (*Hierarchical Segmentation Method*). Assim sendo, ao invés de computar funções complexas, a avaliação da curva de transformação (ICDF) é realizada por meio de uma aproximação polinomial de cada uma das seções resultantes da divisão hierárquica.

Dessa forma, a arquitetura do GRNG exige que os coeficientes, provenientes da interpolação por partes, sejam armazenados e propriamente endereçados. Para tal, a implementação tradicional do gerador emprega LUTs (*look-up tables*) associadas a um sistema de endereçamento proveniente do método de segmentação (HSM) que possibilita que os coeficientes sejam devidamente entregues ao módulo responsável por avaliar o polinômio. Por fim, o cômputo do polinômio é efetuado pelo algoritmo do Horner - metodologia clássica de estimação de polinômio que otimiza o número de multiplicadores e somadores necessários para a operação. A Equação 4.10 ilustra a aplicação do método de Horner e a Figura 16, retirada do trabalho de Liu (2016), exemplifica a arquitetura discutida.

$$p(x) = c_0x^2 + c_1x + c_2 \tag{4.10}$$

$$p(x) = (c_0x + c_1)x + c_2 \tag{4.11}$$

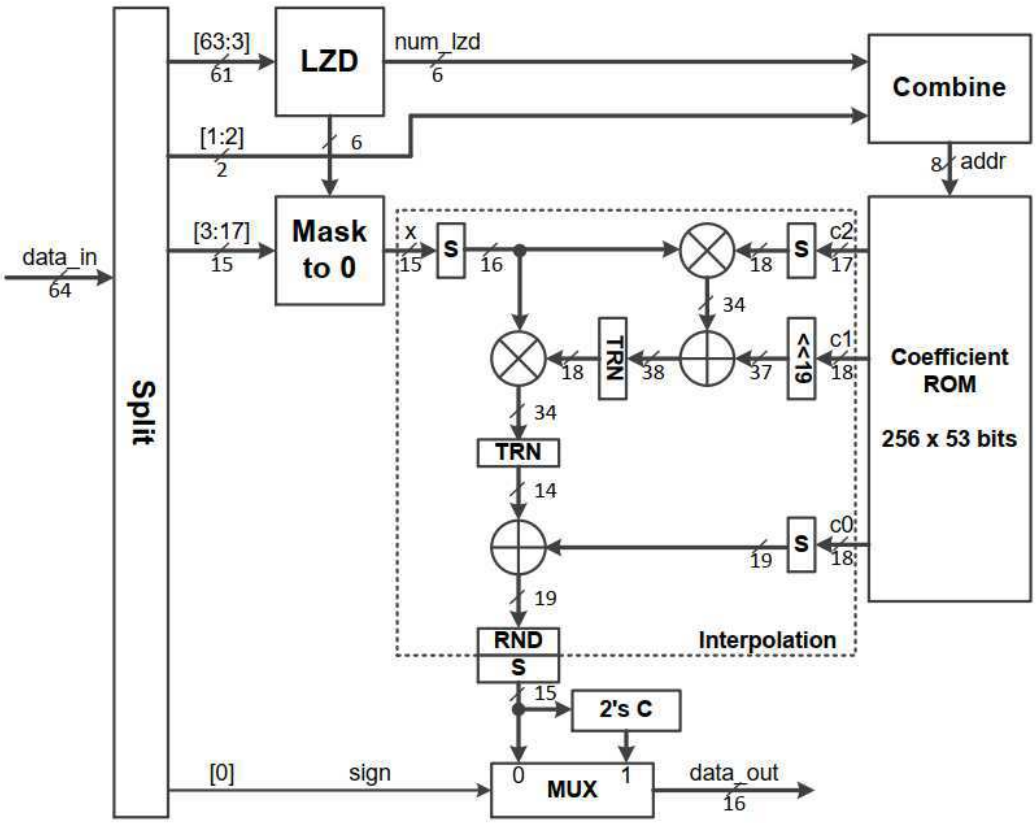


Figura 16 – Arquitetura ICDF desenvolvida por Liu (2016).

Neste trabalho, umas das arquiteturas implementadas se baseia no sistema de LUTs. Os coeficientes gerados a partir do modelo de HSM desenvolvido em Python são armazenados em três módulos de memória que são endereçado por meio da Equação 3.12, exigindo a presença de um módulo de LZD + um concatenador. Além disso, a escolha da amostra utilizada na interpolação foi implementada de acordo com a proposta de Gutierrez *et. al.* (2012), fazendo com que um módulo de mascaramento fosse necessário - e evitando a necessidade de *barrel shifters*. A Figura 17 ilustra, na íntegra, a arquitetura do RTL desenvolvido para esse trabalho.

Como é possível perceber a partir da Figura 17, o método divide os coeficientes em 3 LUTs, significando que o modelo aproxima os segmentos de curva em polinômios de 2º grau. Além disso, é importante ressaltar que o modelo para esse RTL foi desenvolvido para alcançar valores de 8.2σ, ou seja, números da dimensão de 2^{-53} da função *probit*. Os coeficientes apresentam a precisão recomendada pela literatura (GUTIERREZ *et. al.*, 2012) A saída representa uma variável normal de 16 bits em ponto fixo, o qual possui 5 bits inteiros e 11 fracionários.

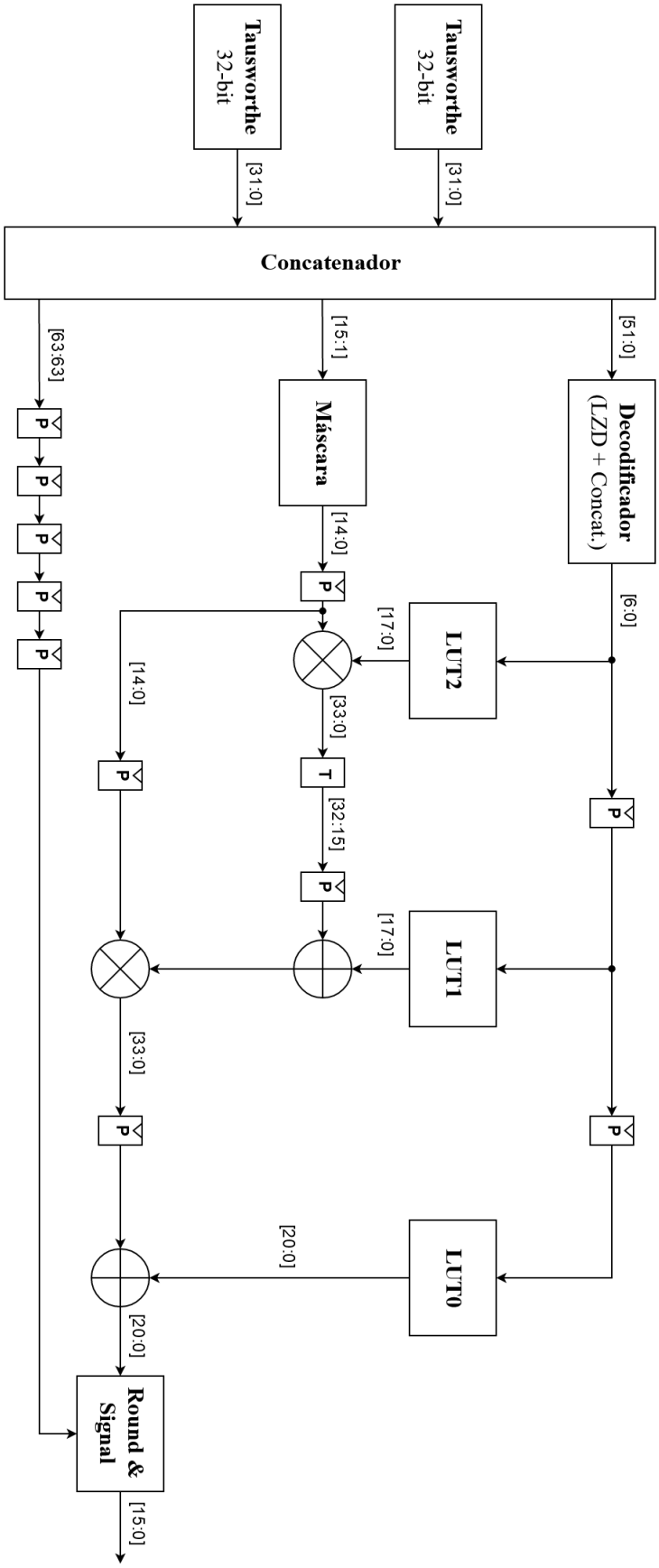


Figura 17 – Arquitetura ICDF com LUTs desenvolvida para esse projeto.

4.2.3 Arquitetura com MCM

Além de implementar o GRNG utilizando LUTs, esse projeto objetiva aplicar os conceitos de MCM (*Multiplierless Constant Multiplier*) no desenvolvimento de uma nova arquitetura para geradores de números aleatórios não-uniformes. A ideia é que seja possível testar a viabilidade de tais estruturas em arranjos que exigem a avaliação de polinômios.

Como detalhado na Seção 3.6, as estruturas de MCM empregam os métodos de multiplicação elaborado pelos egípcios para criar módulos capazes de multiplicar uma entrada por um número fixo de constantes previamente definidas usando somente somadores e deslocamentos (*shift registers*). Quando interpretada no âmbito do desenvolvimento de *hardware*, a aplicação blocos de MCMs pode ocasionar em ganhos significativos em frequência e área, visto que multiplicadores genéricos em RTL costumam ser extremamente onerosos (tanto em área quanto em velocidade).

Como é possível inferir, a função do MCM em um gerador de números aleatórios fundamentado no método do ICDF resume-se em avaliar o polinômio interpolador e, portanto, substituir o complexo de interpolação constituído pelas LUTs e pelos somadores e multiplicadores utilizados pelo algoritmo de Horner. A Figura 18 ilustra a arquitetura final do GRNG concebido com a inserção dos blocos de MCM.

A Figura 18 elucida o fato de que alguns componentes não puderam ser integralmente substituídos na abordagem utilizando MCM. Entre esses itens estão: (a) a necessidade de um multiplicador (bloco de elevação ao quadrado); e (b) a *look-up table* contendo os coeficientes livres do polinômio.

A razão para o estado atual da arquitetura reside na conformação intrínseca de um polinômio e, obviamente, nas limitações das estruturas multiplicativas do MCM. A Equação 4.12 auxilia na visualização das limitações que serão esclarecidas do próximo parágrafo.

$$p(x) = c_0x^2 + c_1x + c_2 \tag{4.12}$$

Como já fora mencionado, o MCM multiplica apenas constantes previamente conhecidas e, portanto, é incapaz de realizar a operação de elevar a entrada ao quadrado, exigindo um multiplicador especializado em multiplicar um número por ele mesmo - função do módulo Quadrado, explicitado no diagrama da Figura 18. Além disso, como o MCM realiza apenas multiplicação, os coeficientes livres (c_2) não poderiam ser armazenados ou incluídos no fluxo de multiplicativo do MCM, exigindo que a LUT0 que armazena esses elementos continuasse no fluxo de dados.

No que diz respeito a função das demais estruturas, é relativamente fácil de associar: o MCM2 recebe o resultado da operação x^2 e entrega, em sua saída, os resultados de sua

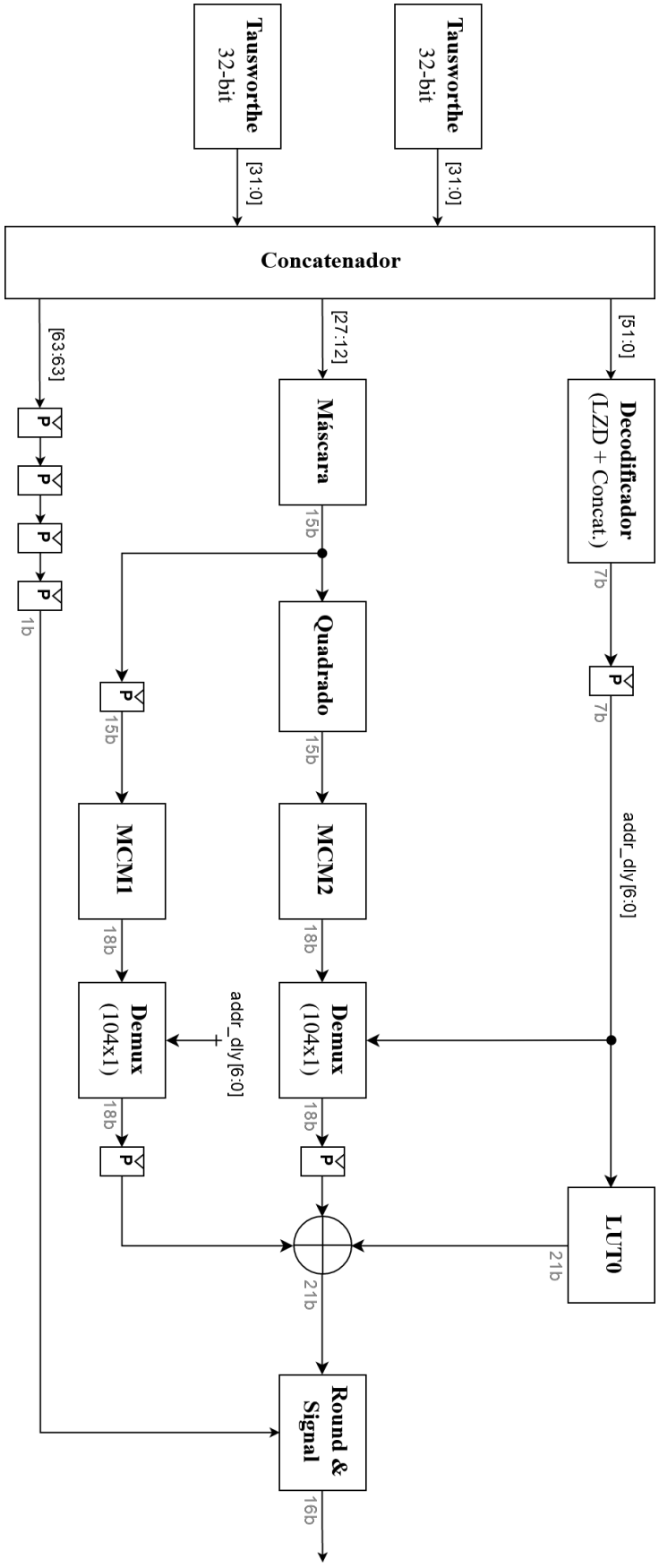


Figura 18 – Arquitetura ICDF com MCMs desenvolvida para esse projeto.

entrada multiplicada por todos os possíveis valores para o coeficientes c_0; o MCM1 realiza função semelhante, mas tomando a entrada como x multiplicando-a por todos os possíveis valores de c_1; os módulos de demultiplexação processam os resultados do MCM1 e MCM2 a partir do endereçamento proveniente do decodificador e entregam apenas o resultado referente ao segmento interpolador selecionado. O resto do fluxo de dados ocorre como na arquitetura tradicional: somam-se as partes computadas e o número é arredondado e devidamente sinalizado de acordo com o bit de sinal extraído da amostra uniformemente distribuída.

Por fim, é importante salientar que o RTL dos MCMs foi inteiramente implementado utilizando a ferramenta de geração de blocos multiplicadores da Spiral (spiral.net). As entradas (coeficientes) necessárias para geração foram obtidas por meio da execução do modelo em Python desenvolvido para o cálculo do HSM desse projeto após passarem pela conversão numérica exigida pelo algoritmo da Spiral.

4.3 Síntese

A síntese foi aplicada nas duas arquiteturas de GRNG: LUT e MCM. Para tal, foram utilizados dois TDK (*Tool Development Kit*): uma biblioteca de 28nm FDSOI disponibilizado pela NXP e uma outra biblioteca livre de 45nm disponibilizada pela Silvaco para aplicações em estudos universitários.

O fluxo de síntese foi conduzido na ferramenta *Genus Synthesis Solution* da Cadence. As arquiteturas foram levadas paulatinamente à sua máxima frequência de operação utilizando configurações máximas de síntese genérica, mapeamento e otimização. Foi tomado o cuidado para que não houvessem *latches* por erros de descrição na etapa de implementação do *hardware*.

As frequências obtidas e discutidas no Capítulo 5 são resultantes apenas do esforço de síntese lógica. Não houve qualquer manipulação de características físicas para que determinados parâmetros de *timing* fossem cumpridos - como o uso de latência na árvore de relógio ou a exigência de atraso máximo em caminhos críticos. Logo, a comparação direta entre os atributos (área, potência e frequência) podem ser considerados imparciais.

5 Resultados e Discussões

5.1 Modelo e Segmentação

Para se obter a melhor configuração para a segmentação da função *probit*, utilizou-se a função `hsmComplete()`, descrita na Seção 4.1.1, dentro de um laço de repetição para que fosse possível variar o número de segmentos externos de natureza $P2S_L$.

Além disso, a função foi inicialmente configurada para que a segmentação interna (US) pudesse ser variável e otimizada de modo a assumir qualquer número de segmentos internos desde que cumprisse as exigências de erro: 2^{-11}, valor equivalente à 1 ulp do resultado final do gerador de números gaussianos. Ademais, é importante salientar que o intervalo englobado só estende-se até 0.5 pois a função *probit* é simétrica e os valores obtidos por um segmento podem ser espelhados invertendo o sinal dos coeficientes ou do resultado final. O Código 2 ilustra a estrutura supracitada.

A Tabela 6 apresenta os resultados do experimento supracitado para os valores de $P2S_L$ variando de 40 à 52. Como é possível notar por meio dos valores apresentados na Tabela 6, o número de segmentos começa a diminuir a medida que nos aproximamos de 48 segmentos primários. A partir desse valor, os ganhos com o aumento de um segmentos reduzem, mas continuam consideráveis.

A principal razão para os números exorbitantes visualizados nos segmentos de menor fracionamento inicial é relativa ao comportamento da função *probit* que, ao se aproximar de valores próximos de 0, começa a adquirir derivadas extremamente elevadas. Assim, na medida em que o intervalo final da função *probit* era fracionado em mais dois segmentos, o processo de segmentação interna era facilitado, reduzindo o número de segmentos totais necessários para se atingir as exigências de erro máximo.

Dado que o objetivo do experimento era descobrir qual o número de segmentos primários ($PS2_L$) que culminavam em menos segmentos totais ($PS2_L$+US), a configuração com 52 segmentos externos foi o valor escolhido como referência para o desenvolvimento do GRNG. A Figura 19 é uma captura da saída do código em Python que explicita o número de segmentos uniformes referentes à cada um dos 52 segmentos externos calculados.

```
Segmentos Primários: 52
Segmentos internos por segmento externo:
[2, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
```

Figura 19 – Número de segmentos uniformes (internos) relativos à cada um dos segmentos externos ($PS2_L$).

Como é possível notar, apenas 7 segmentos não estão subdivididos internamente em

```
interval = [2**(-53), 0.5];
P2S = 1;
U = 1;
aproxPoints = 1000;
degree = 2;
maxError = 2**(-11);
nBits = 53;

indexStart = 40;
indexEnd = 53;
for i in range(indexStart, indexEnd):
    print (i)
    x, coefs, coefs_n, meanError, uL, nUSeg, \
        impossibleAddrFlag, maximumErrors = hsmComplete \
        (i, interval, P2S, U, aproxPoints, degree, \
        maxError, nBits);
    print (uL)
    Uopt.append(sum(uL));
    impossibleFlags.append(impossibleAddrFlag);

print ("Number of total segments: " + str(Uopt))
print ("Impossible flags: " + str(impossibleFlags))
print ("Minimum sections: " + str(min(Uopt)));
print ("Number of primary sections: " + str([(indexStart + i) \
    for i,UoptValue in enumerate(Uopt) \
    if UoptValue == min(Uopt)]));
```

Código 2 – Algoritmo para encontrar a melhor configuração de HSM para a função *probit*.

mais 2 segmentos US. Caso o HSM continuasse assim, seria preciso descrever um *hardware* específico para lidar com essa ausência de padrão no número de segmentos internos. Logo, para reduzir a complexidade do RTL de endereçamento e, consequentemente, otimizar o resultado final, a função `hsmComplete()` foi executada novamente para 52 segmentos externos, mas fixando o número de segmentos internos em 2 para todos os casos. Esse processo resultou em um modelo final com: 52 segmentos primários com 2 segmentos US para cada segmentos externo, resultando em 104 segmentos totais.

Por fim, a Figura 20 apresenta o gráfico do erro máximo de aproximação referente a cada um dos 104 segmentos obtidos no modelo de HSM otimizado para esse trabalho.

5.2 Gerador de Números Aleatórios Gaussianos (GRNG)

Essa seção tem como intuito apresentar os resultados obtidos por meio das arquiteturas de ambos os GRNGs implementados em RTL: LUT e MCM. Para tal, os resultados serão divididos em dois tipos: estatísticos e elétricos (físicos). Na Seção 5.2.1

Número de Segmentos Primários	**Número de Segmentos Totais (externos + internos)**
40	8270
41	4176
42	2130
43	1108
44	598
45	344
46	217
47	154
48	123
49	108
50	101
51	98
52	97

Tabela 6 – Número de segmentos totais otimizados para um erro máximo de 2^{-11} em relação ao número de segmentos primários pré-configurados.

```
primarySegmentation = 52;
interval = [2**(-53), 0.5];
P2S = 1;
U = 2;
aproxPoints = 1000;
degree = 2;
maxError = 2**(-11);
nBits = 53;

x, coefs, coefs_n, meanError, uL, nUSeg, \
    impossibleAddrFlag, maximumErrors, flagError = \
    hsmComplete (primarySegmentation, interval, P2S, \
    2, aproxPoints, degree, maxError, nBits);
```

Código 3 – Chamada da função final da função `hsmComplete()`.

serão abordados os resultados estatísticos obtidos por meio de simulações; já na Seção 5.2.2 serão apresentados os atributos de síntese de cada uma das arquiteturas desenvolvidas.

5.2.1 GRNG: Atributos Estatísticos

Após a implementação de todos os módulos detalhados nas Figuras 17 e 18, foi criado um *testbench* para simular o funcionamento dos RTLs e, consequentemente, validar sua funcionalidade por meio dos atributos estatísticos do conjunto de amostras de suas saídas.

Para que o módulo funcionasse em simulação, foi necessário fornecer apenas 9 entradas: (a) relógio; (b) sinal de reset; (c) sinalizador de relógio válido; e as 6 sementes

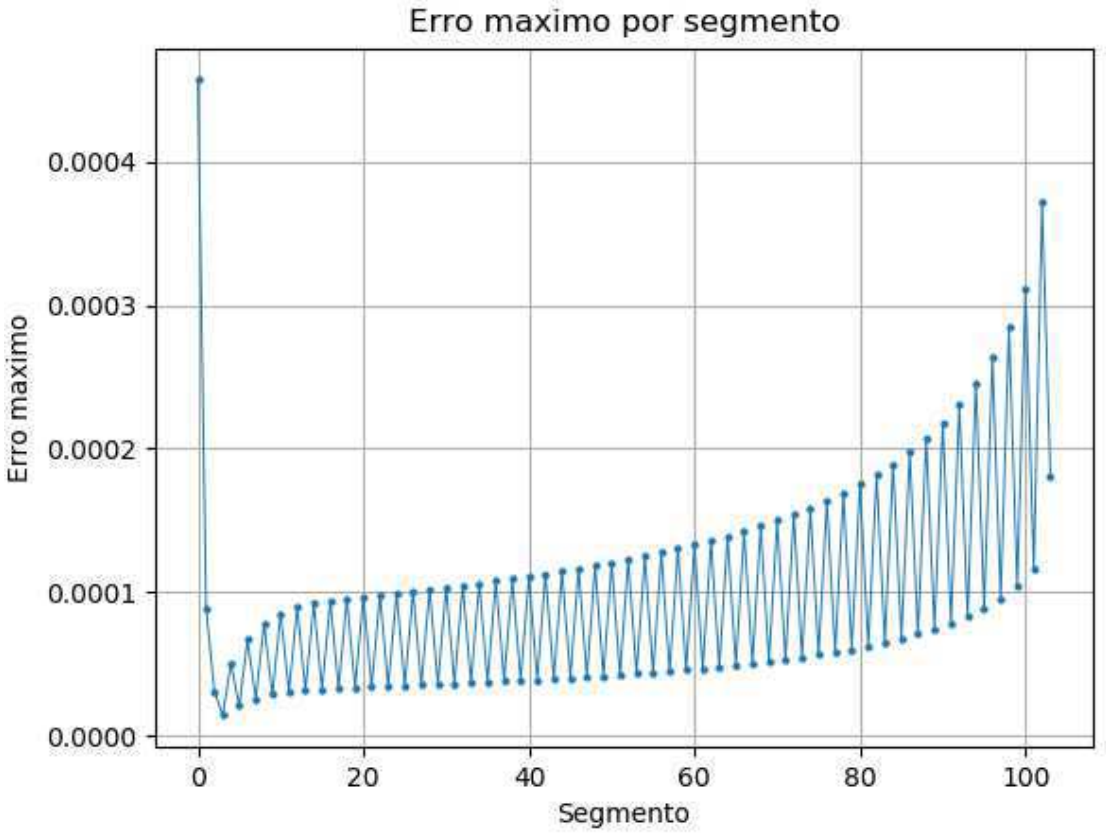


Figura 20 – Error máximo por segmento da segmentação hierárquica.

que alimentam os dois módulos de geração uniforme (URNG). As sementes poderiam ser mantidas fixas como parâmetro do URNG, mas isso reduziria a flexibilidade do gerador final. O Código 4 exemplifica a instanciação do módulo GRNG (tanto LUT quanto MCM) no *testbench*.

```
grng grng (
    .clk(clk),
    .rstn(rstn),
    .taus0_seed0(taus0_seed0),
    .taus0_seed1(taus0_seed1),
    .taus0_seed2(taus0_seed2),
    .taus1_seed0(taus1_seed0),
    .taus1_seed1(taus1_seed1),
    .taus1_seed2(taus1_seed2),
    .grng_out(grng_out)
);
```

Código 4 – Instanciação do módulo GRNG no *testbench*.

Para que fosse possível testar a funcionalidade dos módulos, cada *testbench* (LUT e MCM) foi simulado permitindo que um determinado número de ciclos de *clock* agissem

sobre o RTL. Considerando os nível de *pipeline* de cada um dos módulos, o *testbench* da LUT levava 7 ciclos após o reset para entregar a primeira saída e válida; já o MCM exige apenas 6. A Figura 21 ilustra a forma de onda do *testbench* do módulo GRNG com MCM visualizada por meio do SimVision (ferramenta de visualização de *debug* da Cadence).

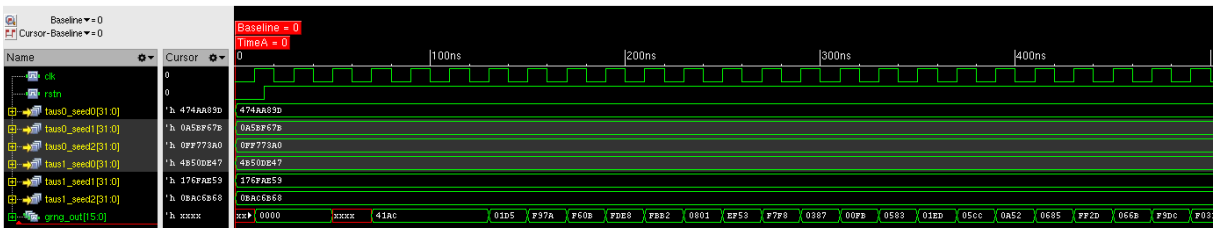


Figura 21 – Print da forma de onda no SimVision (Cadence) do GRNG com arquitetura MCM exibindo clock, reset, sementes e saída durante alguns ciclos.

Após se passarem os ciclos necessários para que a saída se tornasse válida, o *testbench* salva as amostras resultantes do módulo de geração de amostras normalmente distribuídas em um arquivo de dados. Em seguida, o arquivo era processado por um *script* em Python para que fosse possível extrair as características do conjunto de amostras. Para tal, empregaram-se as seguinte etapas: (a) converter amostras do formato binário para sua representação em ponto fixo (5,11); (b) cálculo da média e desvio padrão do conjunto de valores transformados; e (c) geração de um histograma da distribuição obtida.

Cada um dos *testbenchs* foi executado 30 vezes gerando um total de 1 milhão de amostras por ensaio. Os arquivos de saída eram prontamente processados pelo algoritmo em Python para que as informações estatísticas do teste fossem extraídas. Obviamente, as sementes que determinam o comportamento do GRNG eram modificadas à cada novo ensaio, visto que, se assim não o fizesse, os resultados dos ensaios seriam apenas réplicas uns dos outros. A Tabela 7 apresenta os valores finais obtidos para cada uma das arquiteturas.

	Atributos Estatísticos	
	Média	**Desvio Padrão**
Arquitetura MCM	-2.5273811849e-05	1.00005091684
Arquitetura LUT	-2.52697591146e-05	1.00005589696

Tabela 7 – Tabela de atributos estatísticos comparativa entre as arquiteturas.

Como é possível nota após a análise dos dados da Tabela 7, existe uma pequenas diferença entre os valores de média e desvio padrão obtidos em cada uma das arquiteturas. Essa diferença é oriunda da forma como o polinômio de interpolação é avaliado em cada método, Horner para o caso da LUT e avaliação tradicional para o caso do MCM. Como ocorrem arredondamentos em determinados passos de cada metodologia e como esses

passos não são necessariamente iguais, diferenças de 1 ulp entre os resultados finais ocorrem em algumas amostras.

Além das médias e desvios padrões, o histograma para cada uma das arquiteturas também foi elaborado pelo algoritmo em Python a partir das amostra obtidas na saída dos ensaios. Para a obtenção do histograma, foi realizado um número maior de iterações durante a simulação, totalizando 16 milhões de repetições para que fosse possível observar com mais clareza o alcance horizontal que o modelo matemático se propunha à obter. As Figuras 22 e 23 apresentam os resultados de um dos ensaios com 16 milhões de amostras, comprovando mais uma vez a distribuição obtida a partir dos geradores.

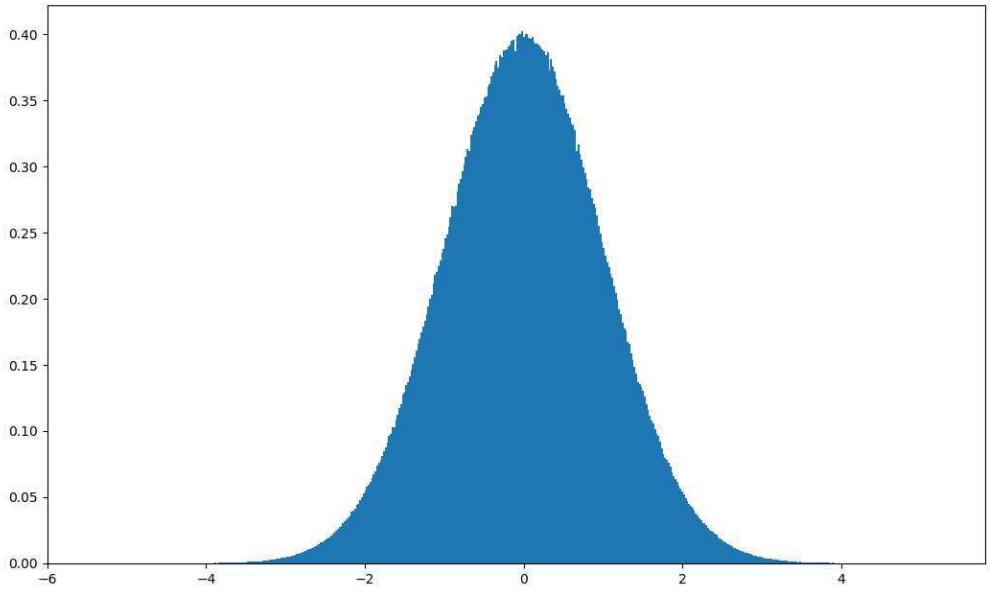


Figura 22 – Histograma de 16 milhões de amostras gerado pela arquitetura MCM com média $E = 0.000219$ e desvio padrão $\sigma = 0.999812$.

5.2.2 GRNG: Atributos Físicos

A síntese, como mencionado na Seção 4.3, foi executada utilizando duas bibliotecas: uma de 28nm e outra de 45nm. Em cada uma das tecnologias, testes exaustivos foram realizados para que fosse possível encontrar a máxima frequência de execução das arquiteturas propostas, avaliando-se outros parâmetros como área e potência. Além da análise de máxima frequência, os módulos foram sintetizados para uma frequência base de 100MHz (frequência de referência). Os resultados da síntese na ferramenta da Cadence (Genus Synthesis Solution) podem ser analisados nas Tabela 8 e 9.

Como é possível notar pelos valores da tabela, a arquitetura tradicional mostrou-se consideravelmente melhor em todos os aspectos. Primeiro, alcançou frequências maiores

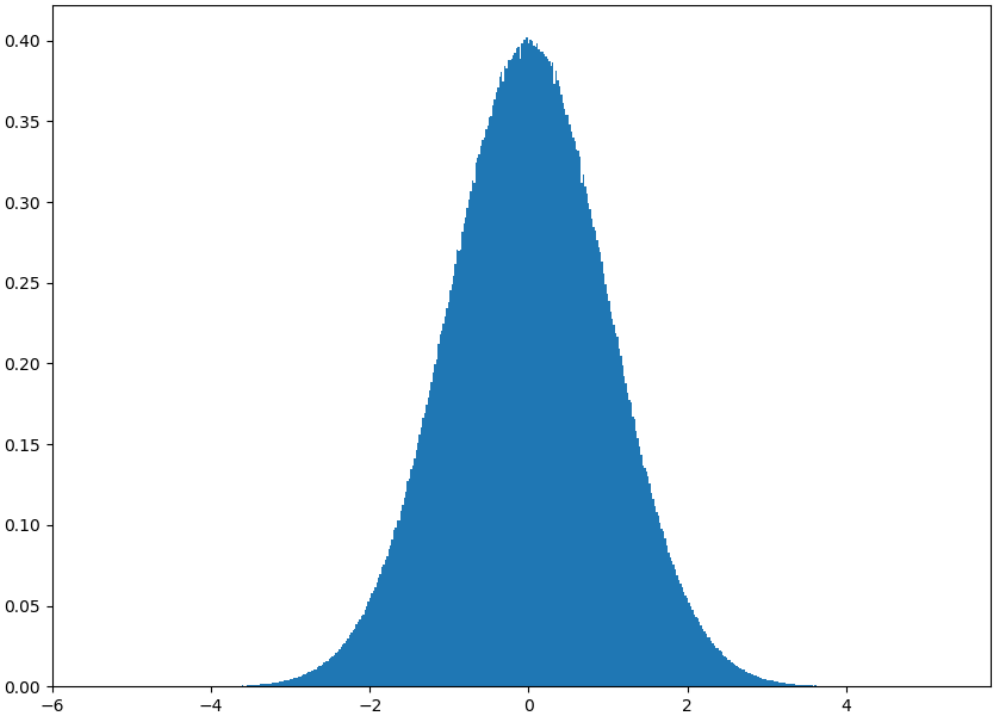


Figura 23 – Histograma de 16 milhões de amostras gerado pela arquitetura LUT com média $E = 0.000219$ e desvio padrão $\sigma = 0.999817$.

em todas as tecnologias. Além disso, apresentou atributos área e uma potência menores do que os do seu concorrente direto, a arquitetura utilizando o MCM.

Uma das razões para que os resultados fossem tão expressivamente negativos para a arquitetura utilizando MCM está relacionada ao fato de não ter sido possível eliminar completamente os blocos de multiplicação da arquitetura, visto que, sem utilizar o algoritmo de Horner (inviável quando inserido numa arquitetura com MCMs), o polinômio de 2º grau exige que a variável de entrada seja multiplicada por ela mesma (operação realizada pelo módulo Quadrado da arquitetura).

Outro fator que contribuiu para o desempenho físico da arquitetura em MCM diz respeito à quantidade de coeficientes (104 no total) que o a árvore multiplicativa deve fornecer em sua saída para toda iteração do GRNG. Assim, não importando qual o segmento selecionado, a arquitetura de um gerador isolado sempre computava o resultado para todas as possíveis 104 multiplicações dos segmentos. Em outras palavas, em um RTL individual do GRNG, 103 resultados eram inutilizados a cada ciclo de clock.

Apesar dos resultados, devido as razões apresentadas, o MCM ainda pode ser considerado como uma solução relevante para geradores de números aleatórios com funções

	Tecnologia 45nm			
	MCM		**LUT**	
	100 MHz	**225 MHz**	**100 MHz**	**310 MHz**
Células (un)	15180	41170	3057	5065
Área (μm^2)	21357.672	43161.958	5408.312	6577.116
Dinâmica (nW)	3683130.260	17838858.309	941011.963	3019589.316
Fuga (nW)	295626.691	696617.882	72882.234	98383.020
Total (nW)	3978756.951	18535476.191	1013894.198	3117972.336
Folga (ns)	6.8	0.0	720.5	0.0

Tabela 8 – Parâmetros físicos para as duas arquiteturas na tecnologia de 45nm em diferentes frequências.

menos não-lineares e mais fáceis de se aproximar. Isso se dá pelo fato dessas funções necessitarem de menos coeficientes para a aproximar a curva com um determinado erro e isso diminuiria a complexidade da estrutura do MCM, tornando-o competitivo em relação à combinação LUT + Horner.

	Tecnologia 28nm			
	MCM		**LUT**	
	100 MHz	**606 MHz**	**100 MHz**	**796 MHz**
Células (un)	13080	48100	3005	5857
Área (um2)	9643.706	24818.368	2587.808	3600.845
Dinâmica (nW)	849284.854	8481828.757	292063.658	2022966.237
Fuga (nW)	23.329	71.800	6.397	9.822
Total (nW)	849308.183	8481900.556	292070.055	2022976.059
Folga	2209.4	0.0	5133.2	0.0

Tabela 9 – Parâmetros físicos para as duas arquiteturas na tecnologia de 28nm em diferentes frequências.

6 Considerações Finais

A partir do desenvolvimento deste trabalho, foi possível verificar e esclarecer a importância da geração de número aleatórios para o desenvolvimento científico e tecnológico nas mais diversas áreas do conhecimento humano. Além disso, notou-se também a emergente necessidade da implementação de geradores de números aleatórios diretamente em *hardware* devido a exigência cada vez maior de estruturas de alto desempenho em sistemas de modelagem/simulação e segurança - esferas que requerem a aplicação direta de números aleatórios para funcionarem adequadamente.

Baseando-se na perspectiva supracitada, o trabalho cumpriu o objetivo principal e implementou, utilizando o método do ICDF, o gerador de números aleatórios com distribuição gaussiana (GRNG) por meio de duas arquiteturas: (a) interpolação tradicional (LUT + Horner); e (b) interpolação com MCM.

A segmentação e aproximação da função *probit* foi integralmente modelada em Python e obteve resultados satisfatórios, visto que foi possível otimizar a fragmentação da função para o menor número de segmentos viáveis de serem endereçados com apenas 53 bits (número de bits necessários para se atingir um alcance de 8.2σ na curva da distribuição normal). Assim sendo, o resultado final foi uma estrutura de HSM de duas hierarquias: $P2S_L + US$, contando com 52 segmentos primários e 2 segmentos secundários por segmento primário.

Ademais, as simulações em RTL executadas para cada uma das abordagens permitiram inferir a qualidade das distribuições geradas. A interpolação utilizando LUT + Horner obteve média igual à $E = -2.52697591146 \times 10^{-5}$ e desvio padrão, $\sigma = 1.00005589696$; já a arquitetura em MCM manifestou média equivalente à $E = -2.5273811849 \times 10^{-5}$ e desvio padrão, $\sigma = 1.00005091684$. Os histogramas das Figuras 22 e 23 corroboram com a excelência das distribuições.

Não obstante a qualidade de ambas tenham sido prontamente demonstrada por meio de atributos estatísticos, o GRNG com interpolação por meio de LUT + Horner mostrou-se categoricamente superior à proposta alternativa (MCM) quando avaliados após a síntese lógica, apresentando maior frequência de operação, menor área, e menor consumo nas duas tecnologia às quais as arquiteturas foram submetidas. Como esclarecido na Seção 5.2.2, esse resultado pode ser imputado à dois fatores dominantes: (a) incapacidade de excluir todos os multiplicadores da arquitetura mesmo utilizando módulos de MCM; e (b) inutilidade da maior parte dos resultados entregues pelo MCM à cada ciclo de clock (desperdiçando 103 resultados por MCM sempre que uma nova amostra era calculada). Mesmo assim, é possível inferir o potencial de aplicação das estruturas de MCM em

geradores com distribuições mais simples (não gaussianas) ou em módulos de interpolação que não exijam um número tão grande de coeficientes.

Como trabalhos futuros, seria interessante testar o impacto da substituição da dupla de Tausworthe 32-bits por um único Tausworthe de 64-bits. Outrossim, avaliar o desempenho de estruturas de MCM em interpoladores de menor grau ou com menos coeficientes - como fora proposto no parágrafo anterior.

Finalmente, conclui-se que, apesar de complexo de extenso, o projeto foi integralmente finalizado, cumprindo-se todos os objetivos elencados na Seção 2 e obtendo-se resultados legitimamente satisfatórios. Ademais, é importante salientar o relevante papel desse trabalho no enriquecimento científico e profissional do seu autor.

Referências Bibliográficas

ALLEN, B. **Pseudo-Random vs. True Random**. Disponível em: <https://boallen.com/random-numbers.html>. Acesso em: 01 de novembro de 2019.

ASWALE, P. S.; MAHAJAN, M. P.; NIKUMBH, M. V.; VAIDYA, O. S. **Implementation of Baugh-Wooely Multiplier and Modified Baugh Wooely Multiplier Using Cadence (Encounter) RTL**. International Journal of Science, Engineering and Technology Research (IJSETR). 2ª ed, volume 4. 2015.

CHEUNG, R.; LEE, D.; LUK, W.; VILLASENOR, J. **Hardware Generation of Arbitrary Random Number Distributions from Uniform Distributions Via the Inversion Method**. IEEE Trans. VLSI, vol. 15, no. 8. Agosto, 2007.

DUTANG, C. WUERTZ, D. **A note on random number generation**. 2009.

FELLER, W. **An Introduction to Probability Theory and Its Applications**. John Wiley & Sons, Inc., 3ª ed. 1968.

GILLE-GENEST, A. **Pseudo-Random Numbers Generators**. 2012.

GUTIERREZ, R.; TORRES, V.;VALLS, J. **Hardware Architecture of a Gaussian Noise Generator Based on the Inversion Method**. 2012.

HAAHR, M. **Introduction to Randomness and Random Numbers**. Disponível em: <https://www.random.org/randomness/>. Acesso em: 06 de ago. de 2018.

LANE, D. M. **History of the Normal Distribution**. Rice University , University of Houston Clear Lake e Tufts University. 2013.

L'ECUYER, P. **Uniform Random Number Generator**. Elsevier Science, Amsterdan. 2005.

L'ECUYER, P. **Maximally Equidistributed Combined Tausworthe Generators**.

Mathematics of Computation. 1996.

L'ECUYER, P. **Uniform random number generation**. Annals of Operations Research. 1994.

LEWIS, D. **Interleaved memory function interpolators with applicationto an accurate LNS arithmetic unit**. IEEE Transactions Computers, vol. 43 , no. 8. 1994.

ORLOFF, J.; BLOOM, J. **Continuous Random Variables**. 2014.

OWEN, A. B. **Monte Carlo theory, methods and examples**. 2013.

PARHAMI, B. **Computer arithmetic**. Oxford University Press, 2ª ed. 2010.

RIBEIRO, M. I. **Gaussian Probability Density Functions: Properties and Error Characterization**. 2004.

ROSS, S. M. **A first course in probability**. 8ª ed. 2010.

SASAO, T.; NAGAYMA, S.; BUTLER, J. **Programmable numerical func-tion generators: Architectures and synthesis method**. 2005.

SCHRYVER, C.; SCHIMIDT, D.; WEHN, N.; KORN, R.; MARXEN, H.; KOSTIUK, A.; KORN, R. **A Hardware Efficient Random Number Generator for Nonuniform Distributions with Arbitrary Precision**. International Journal of Reconfigurable Computing. 2012.

TEZUKA, S. **Random number generation based on polynomial arithmetic modulo two**. IBM TRL Research Report. 1989.

TSANG, W. W.; HUI, L.C.K.; CHOW, K.P.; CHONG, C. **Tuning the Collision Test for Stringency**. HKU CSIS Tech Report. 2000.

THOMAS, D. B.; LUK, W.; LEONG, P. H. W; VILLASENOR, J. D. **Gaussian Random Number Generators**. 2007.

VORONENKO, Y.; PUSCHEL, M. **Multiplierless Multiple Constant Multiplication**. ACM Transactions on Algorithms, vol. 3, no. 2. 2007.

WANG, D. COMPAGNER, A. **On the use of reducible polynomials as random number generators**. Math. Comp. 1993.