José Wemerson Farias Lima

Implementação em Hardware de um Gerador de Ruído Branco de Alta Performance

Campina Grande, Brasil

Dezembro de 2019

José Wemerson Farias Lima

Implementação em Hardware de um Gerador de Ruído Branco de Alta Performance

Trabalho de Conclusão de Curso submetido à Coordenação de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande como parte dos requisitos necessários para a obtenção do grau de Engenheiro Eletricista.

Universidade Federal de Campina Grande - UFCG

Centro de Engenharia Elétrica e Informática - CEEI

Departamento de Engenharia Elétrica - DEE

Coordenação de Graduação em Engenharia Elétrica - CGEE

Orientador: Marcos Ricardo Alcântara Morais

Campina Grande, Brasil

Dezembro de 2019

Agradecimentos

Primeiramente, agradeço a Deus pelo dom da vida. Em seguida, aos meus pais, José Lima de Oliveira e Ana Lúcia Apolinário Farias Lima, por todo o apoio e suporte que me deram nessa longa jornada até aqui e por sempre acreditarem que eu seria capaz.

Ao meu irmão, Lucas, agredeço por ter me feito evoluir pessoal e profissionalmente.

Agradeço aos mitos que conheco, grupo de estudantes, que sempre estiveram presentes ao longo dessa caminhada, em várias noites de estudos e momentos de descontração. Agra- deço também por todas as risadas que foram dadas, por todo apoio e incentivo que nos demos ao longo dessa jornada.

Agradeço aos grupos que participei ao longo da graduação, PET e X-MEN Lab, que me trouxeram engradecimento profissional e pessoal.

Aos meus professores, por todos os ensinamentos e oportunidades que me foram dadas, em especial ao professor Marcos Morais, que me orientou nessa reta final e sempre acreditou no meu potencial.

"O medo se vai quando ouço a voz do alto me dizer: Se valente!"

Marcos Almeida

Resumo

O ruído branco está presente na natureza e impacta os meios de comunicações, por isso é importante emular seu efeito para o uso nos sistemas atuais que alcançam frequências de operação cada vez mais altas. Assim, este trabalho tem o objetivo de obter uma implementação em *hardware* de alta performance que gere valores gaussianos com precisão de 9.15 σ. O método utilizado nesse trabalho já foi desenvolvido por Guangxi Liu, mas aqui há modificações arquiteturais e matemáticas. Para atingir o objetivo, foi utilizada a técnica de ICDF (*Inverse Cumulative Density Function*) em que foram feitas interpolações de segunda ordem para converter valores distribuídos uniformemente em valores Gaussianos. A análise matemática foi desenvolvida com objetivo de calcular os novos coeficientes a serem utilizados na interpolação, os quais possuem menos bits que os originais. A implementação em *hardware* foi feita com HDL (*Hardware Description Language*), em *SystemVerilog*, fazendo uso da técnica de *pipeline* para aplicar altas frequências de *clock*, em seguida foi feita síntese lógica na tecnologia de 28 nm - SMIC, analisando os resultados de área, consumo de energia, *timing* e simulação *Gate Level*, e análise dos resultados matemáticos para verificar a confiabilidade dos dados gerados.

Palavras-chave: Ruído branco, HDL, alta performance, síntese lógica.

Abstract

The white noise is present in nature and impacts the communication systems, that is why it is important to emulate its effect for use in today's systems that reach even higher operating frequencies. Thus, this work aims to obtain a high performance hardware implementation that generates gaussian values with accuracy of 9.15 σ. The method used in this work was already developed by Guangxi Liu, but here there are architectural and mathematical modifications. To achieve this goal, the ICDF (Inverse Cumulative Density Function) technique was used, in which second-order interpolations were made to convert uniformly distributed values to Gaussian values. The mathematical analysis was developed in order to calculate the new coefficients to be used in the interpolation, which have fewer bits than the original ones. The hardware implementation was done with HDL (Hardware Description Language), in SystemVerilog, using the technique of pipeline to apply high frequencies of clock, then logic synthesis was made in 28 nm technology - SMIC, analyzing the area, power consumption, timing and gate level simulation, and analysis of mathematical results to verify the reliability of the generated data.

Keywords: White noise, HDL, high performance, logical synthesis.

Lista de tabelas

Tabela 1 – Parâmetros da distribuíção Normal Padrão. 5
Tabela 2 – Especificações definidas por Guangxi Liu. 6
Tabela 3 – Especificações definidas pelo autor para o PDK de 28 nm. 6
Tabela 4 – Tipos de variáveis sintetizáveis 19
Tabela 5 – Relatório de *Warning* pelo uso indevido do **always_comb**. 22
Tabela 6 – *Constraints* de *Timing* e Área 29
Tabela 7 – Resultados de área e *timing*. 29
Tabela 8 – Relatórios do consumo de potências (mW). 29

Lista de ilustrações

Figura 1 – Fluxo de *front-end*. 3
Figura 2 – Curva da distribuíção Normal Padrão. 4
Figura 3 – Macroarquitetura do bloco AWGN 5
Figura 4 – Estrutura do AWGN. 7
Figura 5 – Microarquitetura URNG. 8
Figura 6 – Esquemático de portas lógicas XOR do URNG. 9
Figura 7 – Microarquitetura da LUT de coeficientes. 10
Figura 8 – Microarquitetura do bloco multiplicador. 11
Figura 9 – Algorítmo de geração de distribuíção gaussiana por meio de ICDF e interpolação polinomial. 12
Figura 10 – Curva ICDF de distribução Normal Padrão. 13
Figura 11 – Curva ICDF de distribução Normal Padrão com segmentação de intervalos. 14
Figura 12 – Distribuíção uniforme gerada pelo URNG. 15
Figura 13 – Evolução dos elementos do Verilog para o SystemVerilog. 17
Figura 14 – Estrutura de síntese lógica. 26
Figura 15 – Modo inapropriado de estruturar um Design. 27
Figura 16 – Modo apropriado de estruturar um Design. 27
Figura 17 – Objetos de uma netlist 28
Figura 18 – Comparação de curva Gaussiana ideal com a gerada pelo AWGN. . . . 31
Figura 19 – Formas de ondas da saída do AWGN na ferramenta **simvision**. 31

Lista de abreviaturas e siglas

AWGN	*Additive White Gaussian Noise*
ASIC	*Application Specific Integrated Circuit*
BER	*Bit Error Rate*
FSM	*Finite State Machine*
HDL	*Hardware Description Language*
ICDF	*Inverse Cumulative Density Function*
IP	*Intellectual Property*
GTL	*Gate-Level*
LUT	*Lookup Table*
PPA	*Power, Performance and Area*
RTL	*Register-transfer level*
SMIC	*Semiconductor Manufacturing International Corporation*
URNG	*Uniform Random Noise Generator*

Sumário

1 Introdução **1**
1.1 Objetivos 2

2 Fluxo de *front-end* para o AWGN **3**
2.1 Epecificação 4
2.1.1 Especificação Funcional 4
2.1.2 Especificação Arquitetural 7
2.1.2.1 URNG Tausworthe 8
2.1.2.2 *Lookup Table* de Coeficientes 9
2.1.2.3 Blocos Multiplicadores 10
2.2 Modelo do sistema 11
2.3 Design do circuito digital 15
2.3.1 Declaração de dados e sinais 18
2.3.2 Declaração de vetores 20
2.3.3 Descrição do RTL 21
2.3.3.1 always_comb 21
2.3.3.2 always_ff 23
2.3.4 Descrição de RTL do AWGN 24
2.4 Síntese lógica 26
2.4.1 Síntese Lógica do AWGN 28
2.5 Resultados 29

3 Conclusões **33**

4 Referências **35**

1 Introdução

Os softwares que estimam o desempenho de um sistema de comunicação consomem tempo, exigindo várias iterações para estimar com precisão os resultados em uma simulação. Por outro lado, o mundo da tecnologia está evoluindo e exige aplicações de *hardware* mais rápidas, com taxas de amostragem mais altas e maior *throughput*, promovendo a necessidade da emulação de *hardware* como um meio de atender às demandas de mercado.

Essas demandas são desafios no processo de desenvolvimento de um IP (*Intellectual Property*), composto de etapas que podem ser divididas pelos setores presentes em uma empresa da área: modelagem, responsável por trazer o "modelo" digital do que o IP deverá fazer; design, o qual implementa a funcionalidade do IP em baixo nível; verificação, responsável por checar a corretude do design perante ao "modelo" entregue pela equipe de modelagem; e por último, *backend*, setor encarregado de fazer a distribuição das células no próprio IP físico.

Ao tratar do uso de *hardware*, é importante saber os problemas que podem acontecer na transmissão dos dados. Assim, o desempenho de um sistema de comunicação digital pode ser quantificado pela probabilidade de erros em bits sob a presença de ruído branco. Para simular um sistema de comunicação, é necessário emular não apenas o transmissor e o receptor, mas também o canal de transmissão. Dessa forma, os projetos de *hardware*, equipados com gerador de ruído preciso e rápido, oferecem o potencial de customizar e acelerar uma simulação por várias ordens de magnitude.

Para emular um canal de transmissão que contenha ruído branco, o AWGN (*Additive White Noise Gaussian*), gerador aditivo de ruído branco gaussiano é frequentemente usado como modelo de canal no qual o único comprometimento da comunicação é uma adição linear de ruído branco. O modelo não considera desvanecimento, seletividade de frequência, interferência, não linearidade ou dispersão. No entanto, produz modelos matemáticos simples e tratáveis, úteis para obter informações sobre o comportamento subjacente de um sistema antes que esses outros fenômenos sejam considerados.

O AWGN proposto produz amostras de ruído branco, o qual é um ruído aleatório que possui uma densidade espectral uniforme com a mesma amplitude em toda a faixa de frequência e é assim chamado porque é análogo à luz branca, que é uma mistura de todos os comprimentos de onda visíveis da luz.

1.1 Objetivos

Esse trabalho tem como objetivo descrever as principais etapas do projeto de design de *front-end* de um ASIC (*Application Specific Integrated Circuit*) específico, como ilustrado na Figura 1.

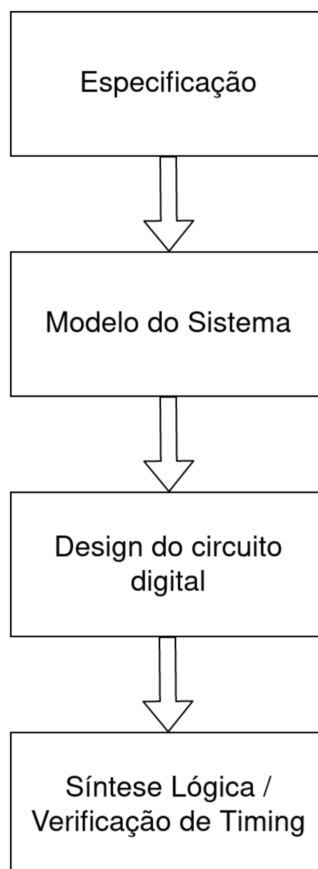
O IP do AWGN desenvolvido gera ruído branco pelo método ICDF e é implementado em *hardware*, com modificações no design original, de forma a obter menos esforço arquitetural para alcançar frequências de *clock* mais altas que às registradas na literatura, com resulução de 16 bits e alcance de 9,15 σ no valor das amostras de saída.

Além da implementação por linguagem de descrição de *hardware*, também é feita a síntese lógica, convertendo o RTL (*Register-transfer level*) em uma *Netlist*, na tecnologia de 28 nm - SMIC. Já na seção de resultados, são feitas considerações e análises extraídas da síntese lógica, simulação GTL (*Gate-Level*) e testes matemáticos.

2 Fluxo de *front-end* para o AWGN

O fluxo de *front-end* consiste em várias etapas, que estão ilustradas na Figura 1. Neste trabalho será explicado em detalhas cada uma dessas etapas no desenvolvimento do AWGN.

Figura 1 – Fluxo de *front-end*.



Fonte: O Próprio autor.

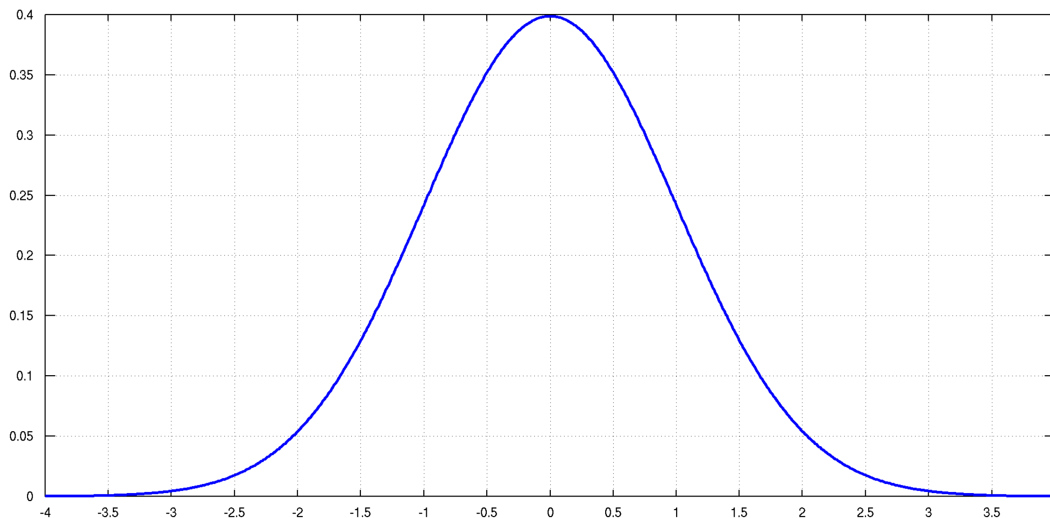
2.1 Epecificação

A especificação de um *hardware* deve vir das necessidades que um cliente leve até a empresa que o fará, as quais devem ser coerentes e alcancáveis para ter êxito na plataforma de desenvolvimento utilizada. Já que o projeto desenvolvido nesse trabalho não visa atender à necessidades comerciais, a especificação foi baseada nas métricas publicadas por outros autores do âmbito acadêmico. A especificação pode ser dividida em duas etapas: a especificação funcional, a qual irá definir as funcionalidade do bloco e sua interface de entradas e saídas; e a especificação arquitetural, para definir a microarquitetura e funcionamento interno dos blocos.

2.1.1 Especificação Funcional

O gerador de ruído gaussiano deve gerar ruído aditivo gaussiano branco de distribuição normal padrão, seguindo valores de referência para essa distribuíção de probabilidade: a média deve ser igual a 0 (zero) e a variância igual a 1 (um). Além desses dois parâmetros principais, um conjunto de saídas do gerador quando colocado em um histograma, deve seguir uma curva de distribuíção Gaussiana Normal Padrão, tal qual a Figura 2.

Figura 2 – Curva da distribuíção Normal Padrão.



Fonte: O Próprio autor.

Dessa forma, a saída deve seguir os parâmetros listados na Tabela 1.

O gerador de ruído gaussiano deve gerar ruído aditivo gaussiano branco de distribuíção Normal Padrão, que pode ser usada para medir o BER (*Bit Error Rate*) de um sistema. O bloco deve utilizar o método ICDF, que a partir de uma distribuíção uniforme produzirá a distribuíção Normal Padrão.

Tabela 1 – Parâmetros da distribuíção Normal Padrão.

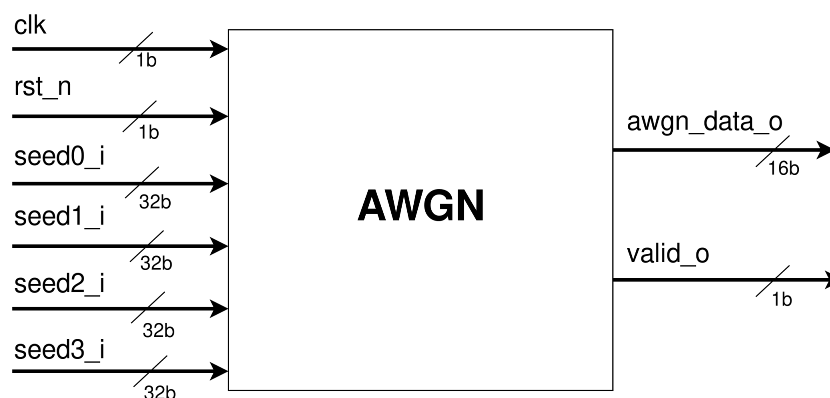
Parâmetro	**Valor**
μ	0
σ	1
$\mu \pm \sigma$	68,27%
$\mu \pm 2 * \sigma$	95,45%
$\mu \pm 3 * \sigma$	99,73%

Além disso, o bloco possui as seguintes características:

- O período da sequência do ruído gerado é de 2^{176};
- Distribuíção aleatória com faixa de valores de $\pm 9,15\sigma$;
- O ruído é quantizado em 16 bits, com 5 bits para a parte inteira e 11 bits para a parte fracionária;
- Baseada em uma aproximação polinomial por partes da função de distribuição cumulativa Normal inversa;
- Alto *throughput*, com uma saída a cada pulso de *clock*;
- Captura assíncrona das *seeds* de entrada.

O barramento de entrada e saída é customizado, e segue a macroarquitetura definida na Figura 3.

Figura 3 – Macroarquitetura do bloco AWGN



Fonte: O Próprio autor.

Além das especificações de funcionamento e *interface* de sinais, são prescritas as especificações PPA (*Power, Performance and Area*) base, retiradas do projeto feito por Guangxi Liu, listadas na Tabela 2.

Tabela 2 – Especificações definidas por Guangxi Liu.

Parâmetro	**Valor**
Área	16 739,52 μm^2
Número de *Gates*	13 078
Frequência máxima	400 MHz
Potência consumida	4.2133 mW

O autor Guangxi Liu realizou seu trabalho sobre o PDK (*Project Design Kit*) SMIC de 55 nm, mas nesse trabalho será utilizado o PDK SMIC de 28 nm. Isso significa que o tamanho do *gate* do transistor é menor, e por consequência, a área, *timing* e consumo de energia para uma mesma estrutura de *harware* terá melhores resultados para o PDK de 28 nm. E como a arquitetura original foi modificada neste trabalho com o propósito de alcançar frequências de *clock* mais altas, a partir da redução de esforço arquitetural dos multiplicadores e implementando a técnica de *pipeline* entre os circuitos combinacionais do bloco AWGN, não é correto comparar a performance entre blocos desenvolvidos neste trabalho e pelo autor Guangxi Liu.

Dessa forma, as métricas especificadas para este projeto no PDK de 28 nm, estão dispostas na Tabela 3.

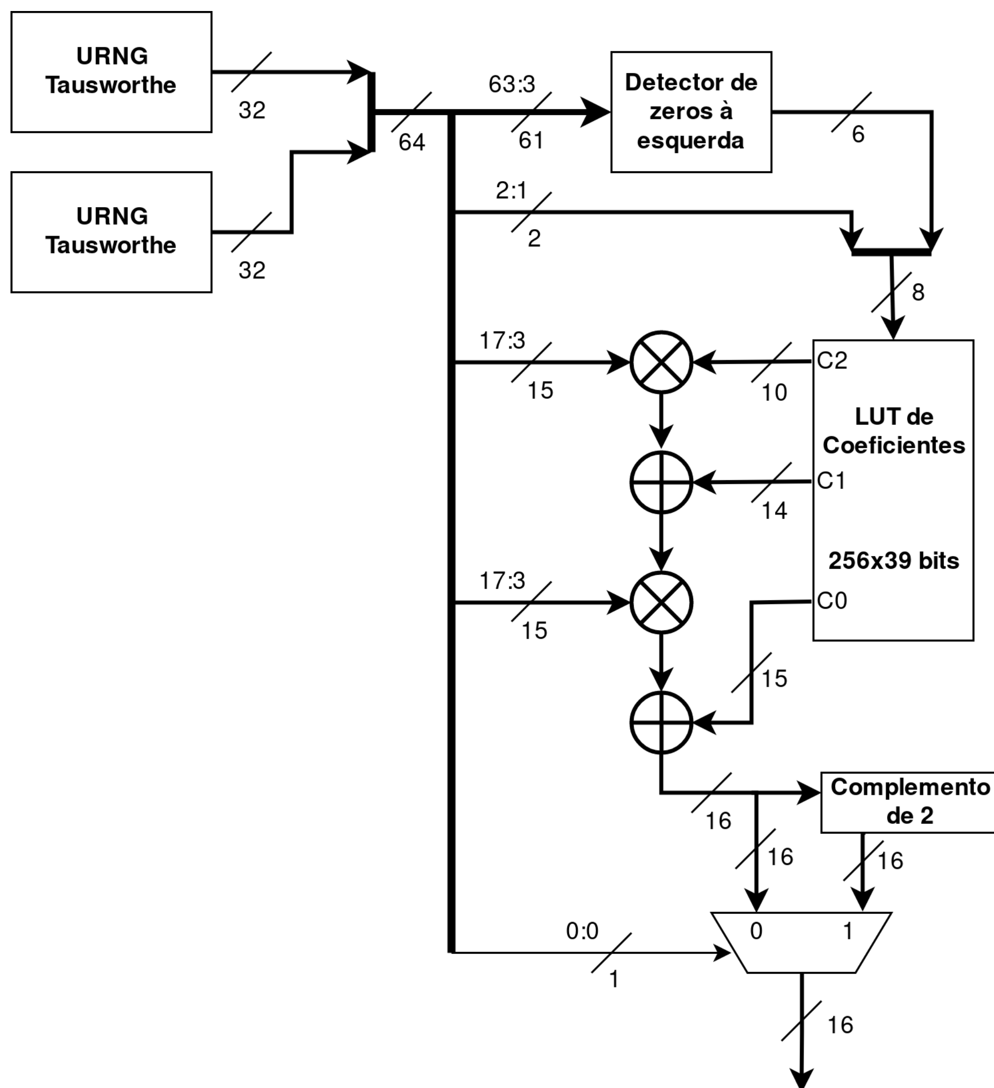
Tabela 3 – Especificações definidas pelo autor para o PDK de 28 nm.

Parâmetro	**Valor**
Área	10 000 μm^2
Número de *Gates*	5 000
Frequência máxima	1.5 GHz
Potência consumida	10 mW

2.1.2 Especificação Arquitetural

A especificação arquitetural designa-se à microarquitetura, para detalhar o funcionamento dos blocos internos, relações entre sinais de controle, barramentos, FSM (*Finite State Machine*) e demais estruturas internas. A partir de uma estrutura esquemática de como o bloco estará disposto, ilustrada na Figura 4, é feita toda a microarquitetura.

Figura 4 – Estrutura do AWGN.



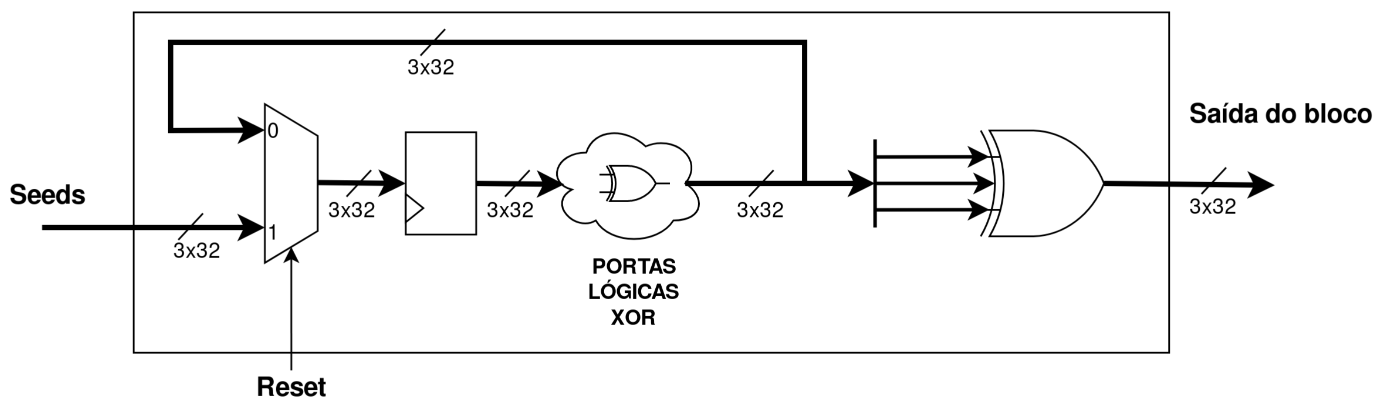
Fonte: O Próprio autor.

2.1.2.1 URNG Tausworthe

O método ICDF para gerar a saída gaussiana tem como parte inicial a geração de valores uniformemente distribuídos. Dessa forma, o URNG (*Uniform Random Noise Generator*) escolhido gera saídas alocadas em uma distribuíção uniforme, $x \in (0, 1)$. São utilizados dois URNG de 32 bits cada, que concatenados formam um barramento na saída de 64 bits.

Os geradores URNG produzem números pseudo-aleatórios, gerando uma sequência de bits a partir de uma recorrência linear módulo 2, e forma números fracionários, tomando por realimentação blocos de bits sucessivos. Para tentar tornar as saídas mais aleatórias, usa-se *seeds* como entrada do bloco,que são capturadas apenas no sinal de *reset*. A microarquitetura do bloco URNG é ilustrada na Figura 5.

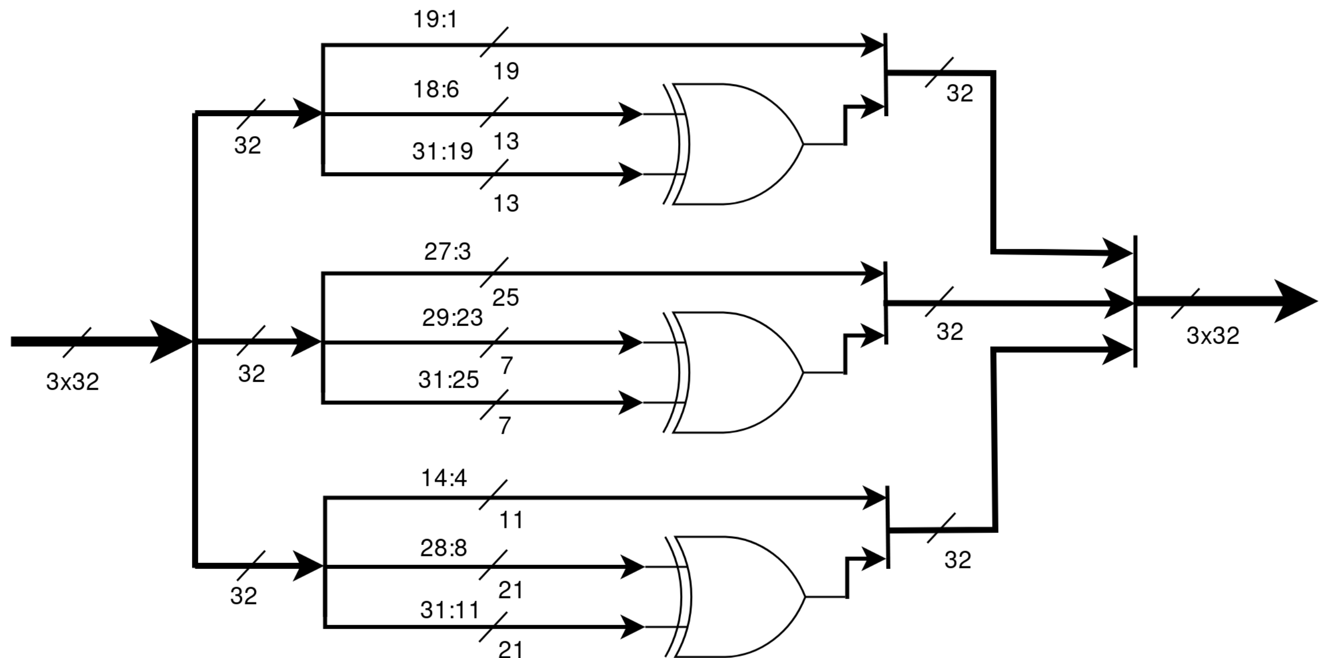
Figura 5 – Microarquitetura URNG.



Fonte: O Próprio autor.

A nuvem combinacional da Figura 5 identificada por 'PORTAS LÓGICAS XOR' segue a estrutura combinacional ilustrada na Figura 6.

Figura 6 – Esquemático de portas lógicas XOR do URNG.

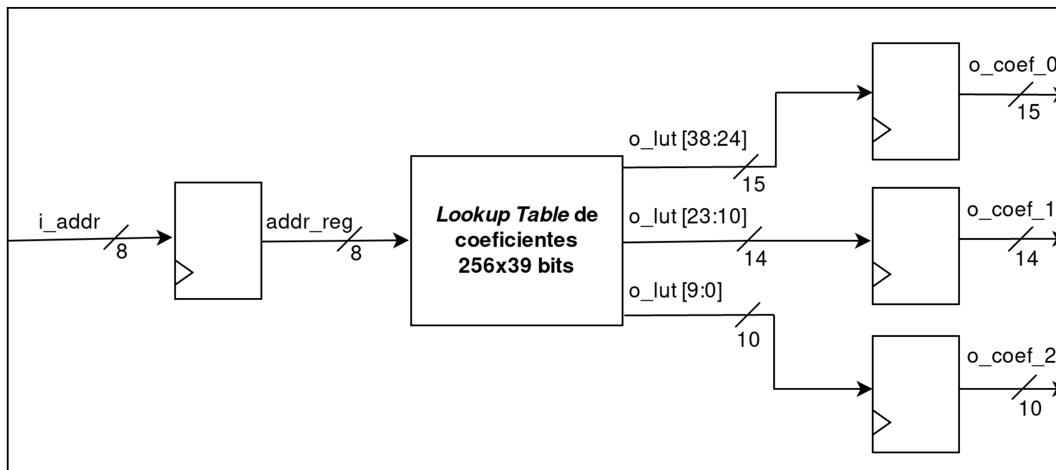


Fonte: O Próprio autor.

2.1.2.2 *Lookup Table* de Coeficientes

Esse bloco contém os endereços que relacionam qual segmento escolhido entre os pontos 0 e 0.5 da curva ICDF. A entrada é um endereço de 8 bits e a saída de 39 bits, em que os 15 bits mais significados são referentes ao coeficiente 0 (zero), os próximo 14 bits são referentes ao coeficiente 1 (um) e os 10 bits menos significativos são referentes ao coeficiente 2 (dois).

Figura 7 – Microarquitetura da LUT de coeficientes.



Fonte: O Próprio autor.

2.1.2.3 Blocos Multiplicadores

No *design* do bloco AWGN pelo método ICDF, as saídas são geradas por meio de interpolação polinomial de seguda ordem. Para tanto, uma equação de segunda ordem pode ser expressa de duas maneiras, que são descritas pelas equações 2.1 e 2.2.

$$y = a * x^2 + b * x + c \tag{2.1}$$

$$y = (a * x + b) * x + c \tag{2.2}$$

Pela equação 2.1, são feitas três multiplicações: uma para elevar o valor de x ao quadrado, outra para multiplicar o coeficiente a e a última para multiplicar o coeficiente b. Já pela segunda equação, são feitas apenas duas multiplicações. E sabe-se que os multiplicadores são considerados gargalos quando desaja-se ter bons resultados de PPA. Por isso são utilizados dois multiplicadores para fazer interpolação de segunda ordem a partir da equação 2.2.

Os dois blocos multiplicadores utilizados possuem a mesma interface de sinais, com duas entradas "i_a"(16 bits) e "i_b"(18 bits), e a saída do multiplicador "o_prod"(34 bits). Os estágios de *pipeline* são aplicados por um multiplicador próprio da Cadence, a partir de uma biblioteca chamada Chip-ware.

Figura 8 – Microarquitetura do bloco multiplicador.

Fonte: O Próprio autor.

2.2 Modelo do sistema

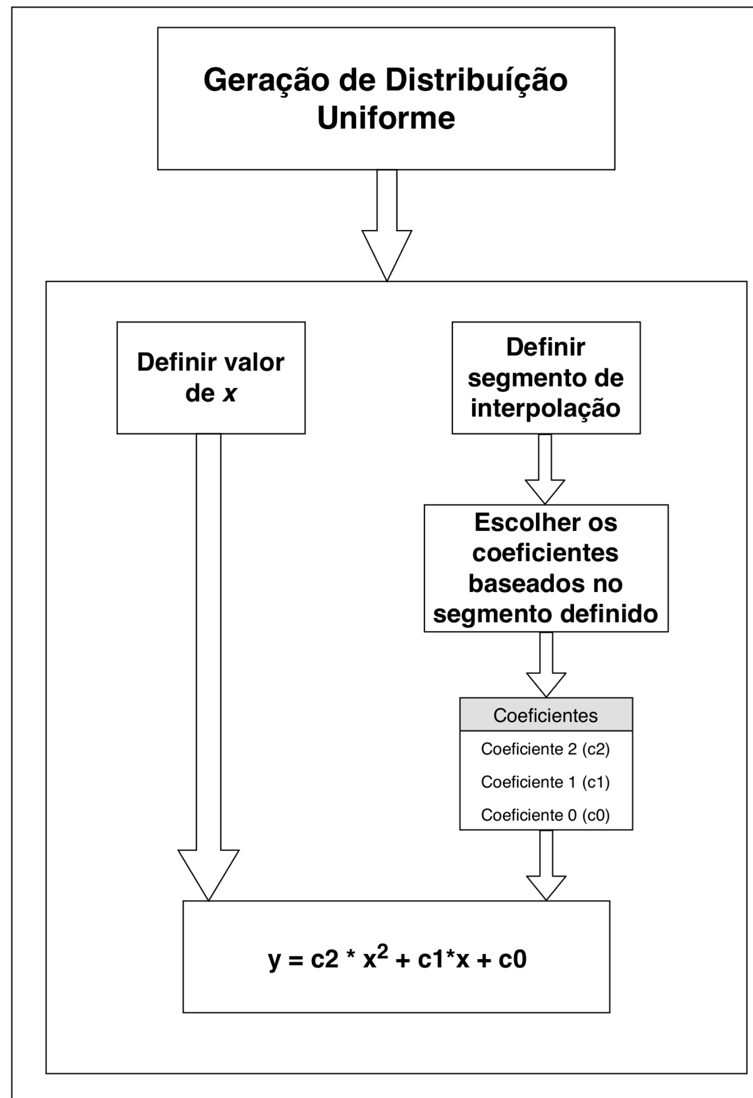
O método de inversão gera amostras gaussianas via ICDF da distribuição gaussiana. Uma amostra distribuída uniformemente, $x \in U(0, 1)$, é transformada em uma amostra y com a função de densidade de probabilidade desejada aplicando $y = ICDF(x)$. Nesse processo, há duas etapas principais: geração de uma distribuíção uniforme por dois URNG e interpolação polinomial. Esta última considera que em uma ICDF o gráfico é simétrico em relação ao ponto (0.5,0), por isso, apenas o aproximamos para resultados positivos. 1 bit do URNG é usado para negar aleatoriamente a saída. Um resumo do algorítmo é ilustrado na Figura 9.

O design desse trabalho usa um esquema de segmentação não uniforme para a aproximação da ICDF. Essa arquitetura é semelhante à uma proposta por Guangxi Liu para a segmentação hierárquica. A segmentação da função leva a dois blocos básicos no cálculo da ICDF: a) seleção de segmento e b) interpolação dentro do segmento selecionado. Normalmente, os bits mais significativos (MSBs) da saída URNG determinam o segmento no bloco a) e do restante dos bits, os mais significativos são usados como entrada para b).

A curva ICDF ideal é ilustrada na Figura 10, já a represenção da ICDF por meio da segmentação de intervalos não lineares e com resolução de 16 bits é ilustrada na Figura 11.

O método de segmentação mais comumente usado é o uniforme, onde todos os comprimentos de segmento são iguais e a contagem de segmentos normalmente é limitada a potências de dois. Embora o esquema uniforme leva ao cálculo simples do endereço

Figura 9 – Algorítmo de geração de distribuíção gaussiana por meio de ICDF e interpolação polinomial.

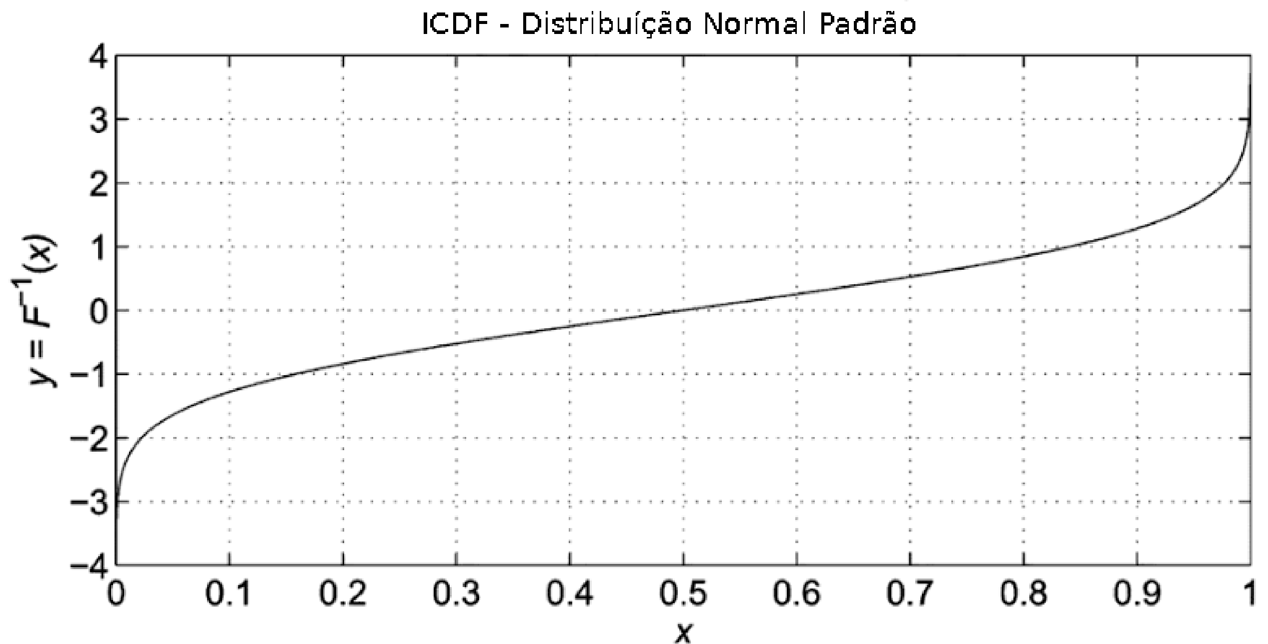


Fonte: O Próprio autor.

do coeficiente, a segmentação não uniforme permite que os comprimentos do segmento sejam personalizados por características da função a ser interpolada. Dessa forma, é feita a aplicação de segmentos hierárquicos para aproximar eficientemente os coeficientes da equação polinomial de acordo com o comportamento das distribuíção Normal Padrão. A não uniformidade nos segmentos é necessária porque a ICDF da distribuíção Normal Padrão possui variações abruptas nas bordas do gráfico, ou seja, próximo ao valor 0 (zero) e 1 (um) do eixo horizontal do gráfico na Figura 10.

Os valores do eixo horizontal do gráfico apresentando na Figura 11 são gerados a partir do gerador de distribuíção uniforme (URNG) definido na seção 2.1.2.1, tal que

Figura 10 – Curva ICDF de distribução Normal Padrão.



Fonte: O Próprio autor.

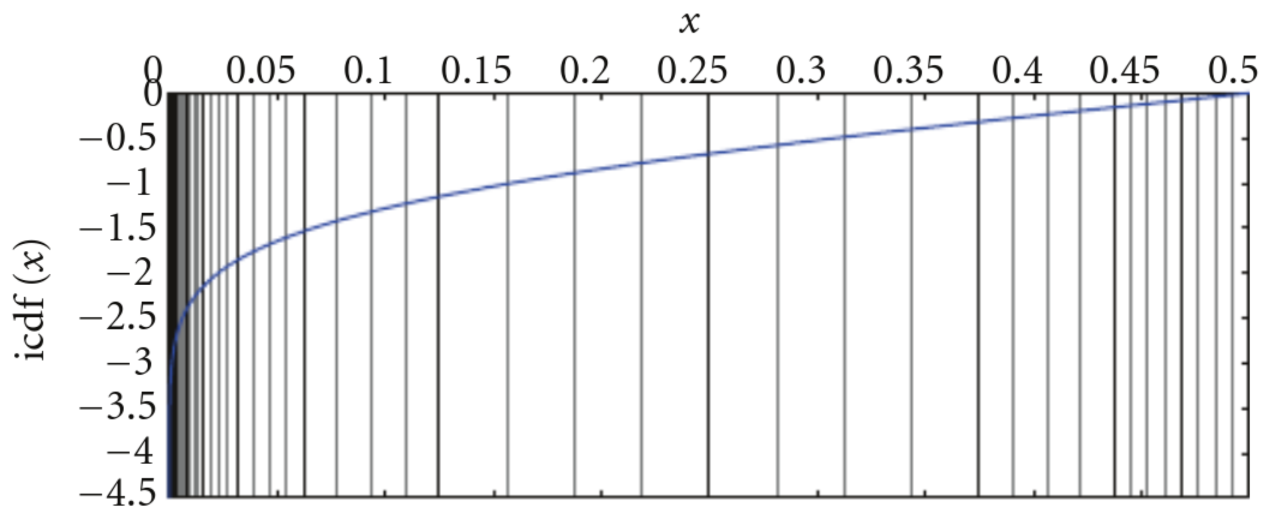
$x \in [0, 0.5]$. O histograma dos valores dessa distribuíção uniforme é ilustrado na Figura 12.

A curva definida na Figura 10 é dada pela equação 2.3. Mas como há uma limitação de bits que deve corresponder aos valores de x no eixo das abscissas, e a interpolação dos segmentos é dada por uma interpolação de segunda ordem. Dessa forma são necessários três coeficientes para compor a equação de segunda ordem, denominados aqui por $coef_0$, $coef_1$ e $coef_2$. Esses coeficientes foram definidos para cada segmento da Figura 11, dados os valores em decimal, foram convertidos para complemento de 2, para que assim possam compor a *Lookup Table* do AWGN.

$$ICDF(x) = \sqrt{2} * erf^{-1}(2x - 1) \tag{2.3}$$

Para converter os valores dos coeficientes de decimal para complemento de 2, foi desenvolvido o seguinte código na ferramenta Octave:

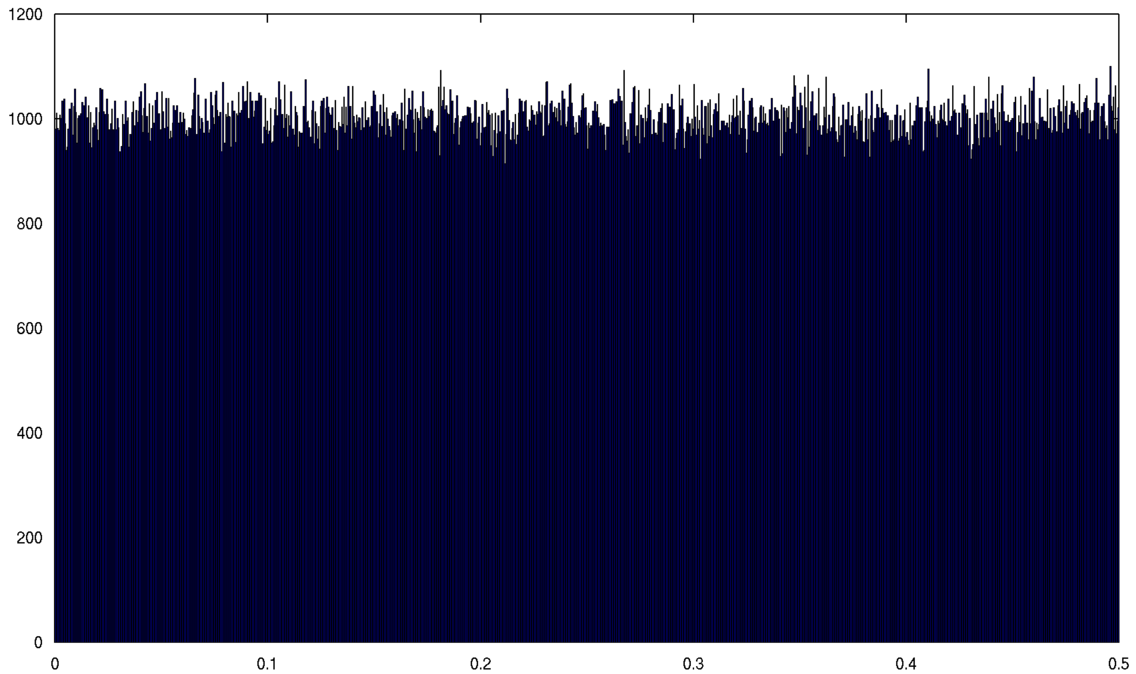
Figura 11 – Curva ICDF de distribução Normal Padrão com segmentação de intervalos.



Fonte: O Próprio autor.

```
function bin = f2b(in)
    dec = 4;
    m = 10;
    decpart = dec2bin(floor(in),dec);
    in = in - floor(in);
    n = -1;
    floatpart = [];
    for i = 1:m
        if in-2 ^ n <0
            floatpart = [floatpart '0'];
        else
            floatpart = [floatpart '1'];
            in = in - 2 ^ n;
        endif
        n = n - 1;
    endfor
    bin = [decpart floatpart];
```

Figura 12 – Distribuíção uniforme gerada pelo URNG.



Fonte: O Próprio autor.

2.3 Design do circuito digital

O design de circuito digital está dentro do fluxo de *front-end* e é voltado para converter as estruturas de modelos e diagramas de blocos, estes descritos anteriormente, em linguagem de descrição de *hardware* (HDL). HDLs são expressões baseadas em texto para descrever a estrutura dos sistemas eletrônicos e seu comportamento ao longo do tempo. Como linguagens de programação simultâneas, a sintaxe e a semântica do HDL incluem notações explícitas para expressar a simultaneidade. No entanto, ao contrário da maioria das linguagens de programação de *software*, os HDL também incluem uma noção explícita de tempo, que é um atributo primário de *hardware*.

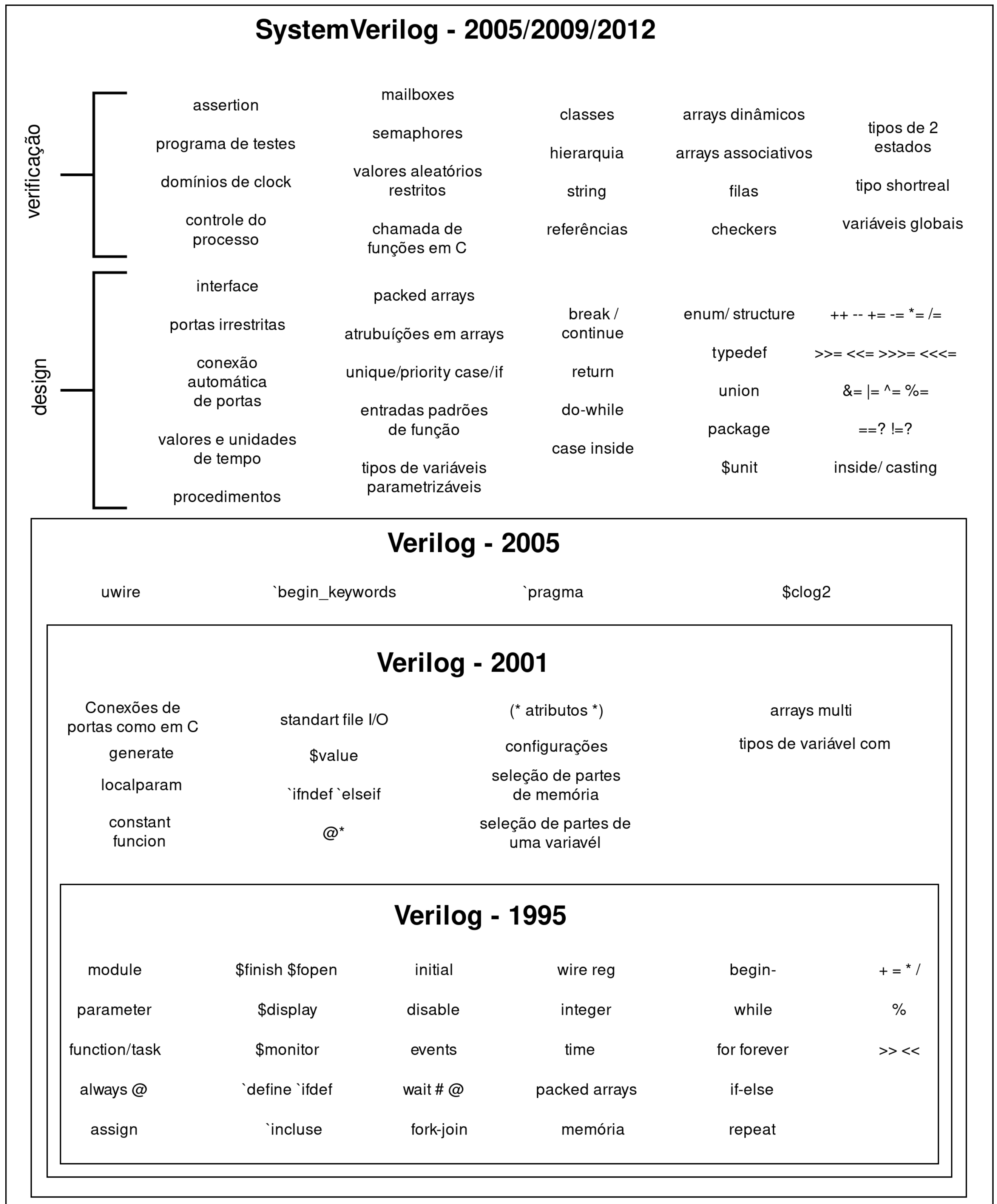
Existem diversos tipos de HDLs, tais como: ABEL; AHDL; AHPL; SystemC; Verilog; SystemVerilog, entre outras. Porém, uma muito importante foi Verilog, introduzido pela primeira vez em 1984 como uma linguagem de dupla finalidade a ser usada para modelar a funcionalidade do *hardware* e para descrever os teste de verificação. Mas para atender às demandas de ambas finalidades citadas anteriormente, o IEEE atualizou o padrão de linguagem Verilog, e surge um novo nome de linguagem de descrição de *hardware*, SystemVerilog.

É importante observar que o padrão SystemVerilog estendeu os recursos de verificação e modelagem do *hardware* já compatíveis com Verilog. O crescimento da linguagem é

ilustrado na Figura 13, a qual não se destina a ser abrangente, mas serve para ilustrar que um número substancial de extensões SystemVerilog para o Verilog original aprimorou a capacidade de modelar hardware.

Nesta seção é abordada uma descrição sobre as partes da linguagem SystemVerilog que são sintetizáveis, ou seja, quais estruturas da linguagem podem ser interpretadas pelo sintetizador e convertidas em estruturas primitivas - portas lógicas AND, OR, NOT, NAND, XOR, NOR e XNOR - e em células específicas da tecnologia utilizada, como meio somadores ou completos, subtratores, multiplicadores, flip-flops de *reset* síncrono ou assíncrono, memórias (SRAM, FIFO, Stacks, Registradores), interfaces de domínios de *clock*, barramentos (APB, AXI), decodificadores, entre outras estruturas.

Figura 13 – Evolução dos elementos do Verilog para o SystemVerilog.



Fonte: O Próprio autor.

2.3.1 Declaração de dados e sinais

A linguagem Verilog original tinha quatro valores de estados, em que cada bit de um vetor podia ser o valor lógico 0, 1, Z ou X, em que 0 e 1 são representações de nível lógico baixo ou alto, Z representa alta impedância ou fios desconectados, e X são níveis lógicos desconhecidos ou que não importam. O SystemVerilog adicionou a capacidade de representar valores de 2 estados, em que cada bit de um vetor pode ser apenas 0 ou 1. Além de ter adicionado as palavras-chave **bit** e **logic** à linguagem Verilog para representar 2 e 4 valores de estado, respectivamente. Os tipos **net** em SystemVerilog, como **wire**, usam apenas o conjunto de valores lógicos de 4 estados. Alguns tipos de variáveis usam conjuntos de valores lógicos de 4 estados, enquanto outras variáveis usam conjuntos de valores de bits de 2 estados.

As palavras-chave **bit** e **logic** também podem ser usadas sem definir explicitamente uma **net** ou variável. Nesse caso, uma **net** ou variável é inferida a partir do contexto. A palavra-chave **bit** sempre infere uma variável. A palavra-chave **logic** infere uma variável na maioria dos contextos, mas infere uma **net** se usada em como **input** (entrada) **inout** (entrada/saída) na declaração de portas do módulo. As seguintes declarações ilustram essas regras de inferência:

```
1
2  module exemplo_modulo (
3  // portas do modulo com tipos inferidos
4          input          i1, // infere net de 4 estados
5          input logic    i2, // infere net de 4 estados
6          input bit      i3, // infere net de 2 estados
7
8          output         o1, // infere net de 4 estados
9          output logic   o2, // infere net de 4 estados
10         output bit     o3  // infere net de 2 estados
11 );
12
13 // sinais internos com tipos inferidos e explicitos
14 bit                clock; // infere uma vari vel de 2 estados
15 logic              reset; // infere uma vari vel de 4 estados
16 logic       [7:0]  data;  // infere uma vari vel de 4 estados
17 wire        [7:0]  n1;    // declara explicitamente uma net,
18                           // infere logic de 4 estados
19 wire logic  [7:0]  n2;    // declara explicitamente uma net de 4
```

```
20                                  // estados
21 var         [7:0] v1;            // declara explicitamente uma vari vel ,
22                                  // infere logic
23 var   logic [7:0] v2;            // declara explicitamente uma vari vel
24                                  // de 4 estados
25
26 ...
27
28 endmodule
```

É importante saber que na síntese os sinais **bit** e **logic** são tratados da mesma forma, assim, os sinais de 2 ou 4 estados servem para simulação, mas não tem significância para o sintetizador.

Além das declarações dos tipos de dados, é necessário saber que há tipos de variáveis utilizadas em código procedural, também conhecidos como blocos **always**. O Verilog/SystemVerilog requer que o lado esquerdo das atribuições procedurais deve ser do tipo de alguma variável. Os tipos de variáveis sintetizáveis no SystemVerilog são expostas na Tabela 4:

Tabela 4 – Tipos de variáveis sintetizáveis

Tipo de variável	**Descrição**
reg	variável de 4 estados de uso geral de um tamanho de vetor definido pelo designer
integer	variável de 4 estados de 32 bits
logic	exceto nas portas de entrada/saída do módulo, infere uma variável de 4 estados de uso geral tamanho definido vector
bit	infere uma variável de 2 estados de uso geral de um tamanho de vetor definido pelo usuário
byte, shortint, int, longint	variáveis de 2 estados com tamanhos de vetor de 8 bits, 16 bits, 32 bits e 64 bits, respectivamente

2.3.2 Declaração de vetores

Os vetores são declarados especificando um intervalo de bits entre colchetes, seguido pelo nome do vetor. O intervalo é declarado como [número_de_bits_mais_significativos: número_de_bits_menos_significativos] O MSB e o LSB podem ser qualquer número, e o MSB pode ser o maior ou menor número. Segue um exemplo:

```
1 // Exemplo de declara   o de vetores packed, tambem chamados
2 // de arrays
3 wire  [31:0] a;  // vetor de 32 bits, little endian
4 logic [1:32] b;  // vetor de 32 bits, big endian
```

Na declaração de vetores, seleções de bits e seleções de parte (vários bits) de vetores sempre fizeram parte do Verilog e são sintetizáveis. O padrão Verilog-2001 adicionou seleção variável das partes do vetor, e que também são sintetizáveis. O padrão SystemVerilog refere-se a vetores como matrizes compactadas, para indicar que um vetor representa uma matriz de bits que são armazenados contiguamente. O único aprimoramento significativo que o SystemVerilog adiciona é a capacidade para dividir uma declaração de vetor em subcampos usando vários intervalos. Por exemplo:

```
1 logic [3:0][7:0] a; // vetor de 32 bits, dividido em 4
2                     // subcampos de 8 bits
3
4 a[2] = 8'hFF;    // atribui 8 bits ao subcampo 2 do vetor
5 a[1][0] = 1  b1  ; // atribui um   nico  bit para o subcampo 1
```

Matrizes *packed* multidimensionais e seleções dentro de matrizes *packed* multidimensionais são sintetizáveis. O uso desse recurso em SystemVerilog pode ser benéfico quando um projeto precisa fazer referência frequente a subcampos de um vetor. O exemplo acima simplifica a seleção de bytes do vetor de 32 bits.

O SystemVerilog permite declarar matrizes de dimensão única e múltipla de forma *unpacked*, seguindo o exemplo:

```
1 // Exemplo de declara   o de vetores unpacked
2 logic [7:0] vec [0:255]; // vector unidimensional de 256 bytes
3 logic [7:0] vec_2 [16] [16] [16]; // vetor tridimensional de bytes
```

2.3.3 Descrição do RTL

O SystemVerilog adiciona um número significativo de recursos de programação ao Verilog tradicional. As intenções desses aprimoramentos são 3: 1) ser capaz de modelar mais funcionalidades em menos linhas de código; 2) reduzir o risco de erros funcionais em um design e 3) para ajudar a garantir que a simulação e a síntese interpretem funcionalidade de design da mesma maneira.

No Verilog tradicional, os blocos procedurais **always** são usados para modelar lógica combinacional, sequencial e *latch*. As ferramentas de síntese e simulação não têm como saber que tipo de lógica um designer pretende representar. Em vez disso, essas ferramentas só podem interpretar o código dentro do bloco procedural e, em seguida, "inferir", que é apenas uma boa maneira de dizer um "palpite", a intenção do designer.

Erros simples de codificação na lógica combinacional podem gerar *latches* sem indicar que o comportamento de um *latch* existe até que os relatórios dos resultados da síntese sejam examinados.

O SystemVerilog adiciona dois novos tipos de blocos **always**, são eles: **always_comb** e **always_ff**. Simuladores e compiladores de síntese podem emitir avisos se o código modelado nesses novos blocos não corresponde ao tipo designado. Esses avisos são opcionais e não são exigidos pelo padrão SystemVerilog. No momento, não há ferramentas de simulação que forneçam esses avisos, mas os compiladores de síntese fazem.

2.3.3.1 always_comb

O bloco procedural **always_comb** indica a intenção do designer para representar uma lógica combinacional. Para inserir esse bloco com procedimentos condicionais, deve-se cobrir todas as possibilidades, ou seja, para condicionalidade de 1 bit, deve haver duas sentenças, **if ... else.**, como segue no exemplo a seguir.

```
1 always_comb begin
2     if (sel)
3         y = a;
4     else
5         y = b;
6 end
```

Um benefício importante a ser observado é que as ferramentas podem inferir uma lista combinatória de sensibilidade, porque as ferramentas sabem a intenção do bloco procedural. Um erro comum de codificação com o Verilog tradicional é ter uma lista

de sensibilidade incompleta. Este não é um erro de sintaxe e resulta em um problema simulação de RTL se comportando de maneira diferente do que o comportamento no nível de *gates* pós-síntese. Deixar indevidamente um sinal na lista de sensibilidade é um erro fácil de cometer em lógica de decodificação grande e complexa. É um erro que provavelmente será detectado durante a síntese, mas isso só ocorre após muitas horas de tempo de simulação terem sido investidas no projeto. O Verilog-2001 adicionou a construção **always @ *** para inferir automaticamente uma lista completa de sensibilidade, mas isso não é perfeito e, em alguns casos extremos, a simulação e a síntese inferem listas diferentes de sensibilidade. O bloco procedural **always_comb** do SystemVerilog possui regras muito específicas que garantem que todas as ferramentas de software inferiam a mesma lista de sensibilidade combinatória.

Outro benefício importante é que, como as ferramentas de software sabem que a intenção é representar a lógica combinacional, as ferramentas podem verificar se esse objetivo está sendo atendido. O exemplo de código a seguir ilustra um bloco procedural que usa **always_comb**, mas não se comporta corretamente como lógica combinacional. Seguido do exemplo de código é exibido o relatório e aviso emitidos pelo DC para este código.

```
1 module always_comb_teste (
2     input  logic a, b,
3     output logic c
4 );
5     always_comb
6         if (a) c = b;
7 endmodule
```

Tabela 5 – Relatório de *Warning* pelo uso indevido do **always_comb**.

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
c_reg	Latch	1	N	N	N	N	-	-	-

Quando este exemplo do código anterior foi lido na ferramenta de síntese DC, foi gerada uma "Tabela de registro"indicando o tipo de registro como "**latch**". Além disso, um aviso de elaboração foi emitido, observando que um **latch** foi modelado em um bloco **always_comb**.

Além do **always_comb** como estrutura de atribuíções combinacionais, há também o **assign**, utilizado para atribuíções de um sinal em outro, como também estruturas de

seleção, como no exemplo a seguir:

```
logic a, b, x, y, sel;

assign x = a + b; // pode ser substituido pelo always_comb

assign y = sel ?  a : b; // uso de operador ternario que faz
                         // a mesma funcao logica do Mux com
                         // o always_comb
```

2.3.3.2 always_ff

O bloco processual **always_ff** documenta a intenção do designer de representar o comportamento do flip-flop. **always_ff** difere de **always_comb**, pois a lista de sensibilidade deve ser especificada pelo designer, pois neste bloco as ferramentas de software não podem inferir o nome e a borda do *clock*. Uma ferramenta também não pode inferir se o designer pretendia ter *reset* síncrono ou assíncrono. Esta informação deve ser especificada como parte da lista de sensibilidade. As ferramentas de software ainda podem verificar se a intenção do comportamento do flip-flop está sendo atendida no corpo do bloco procedural. No exemplo a seguir, o **always_ff** é usado para o código que realmente não modela um flip-flop. O aviso emitido pelo DC segue o exemplo.

```
module always_ff_teste (
    input  logic a, b, c
    output logic out
);
    always_ff @(a, b, c)
        if (a)
            out = b;
        else
            out = c;
endmodule
```

Warning: teste.sv:5: Netlist for always_ff block does not contain a flip-flop. (ELAB-976)

Uma forma correta de estrurar um bloco procedural **always_ff** é ilustrada no código a seguir.

```
1  module always_ff_teste_2 (
2      input  logic       clk, rst,
3      input  logic       a, b, c
4      output logic [1:0] out
5  );
6      always_ff @(posedge clk, posedge rst)
7          if (rst)
8              out <= 2'b0;
9          else
10             out <= a + b + c;
11 endmodule
```

2.3.4 Descrição de RTL do AWGN

A descrição de *hardware* com a linguagem SystemVerilog fica mais trivial quando se tem a microarquitetura e diagrama de blocos do que se deseja descrever. Dessa forma, a estruturação do código fica mais fiel ao modelo proposto. Como ilustrado nas Figuras 5 e 6, a microarquitetura do bloco URNG pode ser interpretada seguindo as seguintes instruções:

- Um Multiplexador, ou simplesmente Mux, pode ser feito de duas formas: utilizando estruturas primitivas, diretamente com portas lógicas ou utlizar a estrutura de **always_comb**. As duas formas são descritas a seguir.

Como o bloco URNG necessita de um Mux de seleção de dois sinais de 32 bits, o módulo do código abaixo, **MUX2x1**, deve ser instaciado 32 vezes, já que ele faz a operação com apenas 1 bit de cada sinal de entrada.

```
1  // Exemplo de Implementacao de um Multiplexador 2 pra 1
2  // utilizando portas logicas
3
4  module MUX2x1(
5                input  logic A,
6                input  logic B,
7                input  logic S,
8                output logic X
9  );
```

```
10
11 // As entradas sao A e B
12 // O sinal de selecao S
13 // O sinal de saida    X
14 wire S0_inv;
15 wire a1,b1;
16
17 not u1( S0_inv, S );
18 and u2( a1, S0_inv, A );
19 and u3( b1, S, B );
20 or  u4( X, a1, b1 );
21
22 endmodule
```

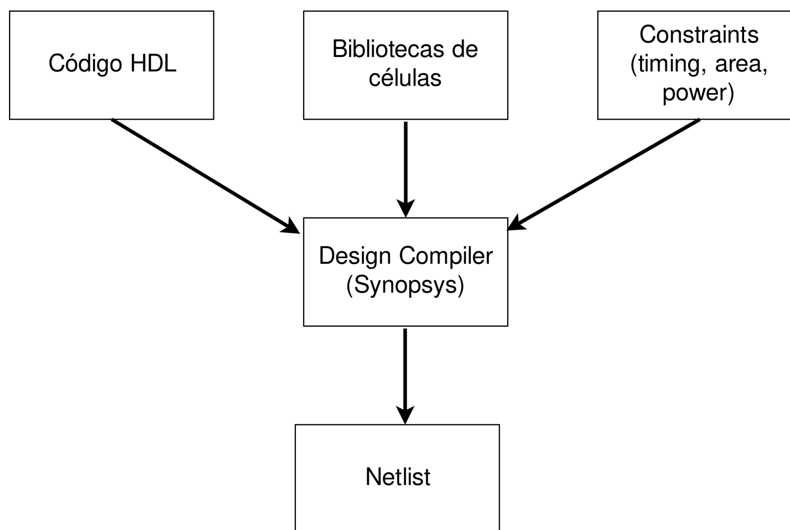
Por ser custoso em sua represenção, o código acima não é utilizado, mas sim o código abaixo, utilizando a estrutura **always_comb**.

```
1 // Exemplo de Implementacao de um Multiplexador 2 pra 1
2 // utilizando always_comb
3
4 module MUX2x1(
5               input  logic [31:0] A,
6               input  logic [31:0] B,
7               input  logic        S,
8               output logic [31:0] X
9 );
10  // As entradas sao A e B, o sinal de selecao S e
11 // o sinal de saida    X
12 wire S0_inv;
13 wire [31:0] a1, b1;
14
15 always_comb begin
16     if(sel == 1'b1)
17         X = A;
18     else
19         X = B;
20 end
21
22 endmodule
```

2.4 Síntese lógica

A síntese lógica feita no **front-end** tem o objetivo de converter o código de uma linguagem de descrição de **hardware**, neste trabalho usa-se o SystemVerilog, em estrutuas de *gates*. Os *gates* são as células da tecnologia do PDK que serão utilizadas. A partir do diagrama de blocos sobre a síntese lógica, ilustrado na Figura 14, nota-se que são necessários alguns passos para sintetizar um circuito.

Figura 14 – Estrutura de síntese lógica.



Fonte: O Próprio autor.

O primeiro passo para fazer uma síntese é ter o código HDL do circuito a ser sintetizado. Em paralelo deve-se ter a biblioteca de células a ser utilizada, e para este trabalho foi utilizada a biblioteca do PDK de 28 nm SMIC. Por fim, definir as *constraints*, ou seja, as especificações de como o circuito deve operar a nível de *gates*: definição do *clock* a ser utilizado, além de outras *constraints* de tempo; definir o quanto da área pode ser otimizada; e por fim definir até que ponto o consumo de potência pode ser otimizado.

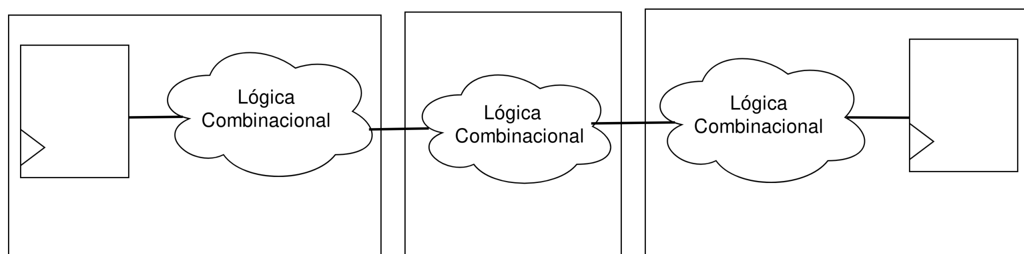
Para realizar uma síntese do circuito digital para melhores resultados (menos tempo de síntese; resultar em circuitos menores; alcançar as *constraints* mais facilmente) de síntese, é necessário saber de algumas informações. Primeiro, há um conceito de particionamento do Design, e ao invés de fazer a descrição do circuito inteiro em um único módulo, deve-se particionar em pequenos módulos e seguir as características abaixo:

- Dividir pela funcionalidade;
- Registrar todas as saídas;

- Não colocar lógica combinacional entre blocos;
- Separar designs com diferentes objetivos;
- Não manter blocos com códigos extensos.

O modo inapropriado de estruturar um Design é ilustrado na Figura 15, em que o caminho entre dois registradores é dividido em três diferentes blocos combinacionais. Nesse caso, a otimização é limitada porque os limites hierárquicos impedem compartilhamento de termos comuns.

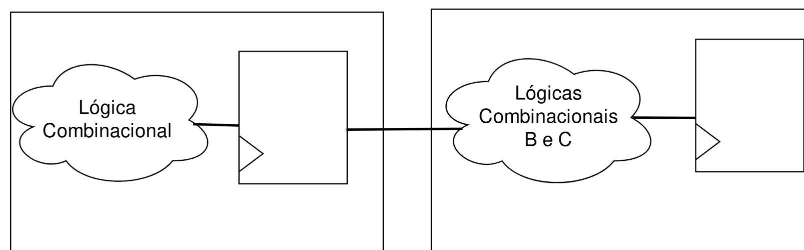
Figura 15 – Modo inapropriado de estruturar um Design.



Fonte: O Próprio autor.

O modo correto de estrutar mais de uma lógica combinacional e diminuir as estruturas hierárquicas entre dois registradores, é alocar em um mesmo bloco e registrar as saídas. A ilustração da Figura 16 é um exemplo.

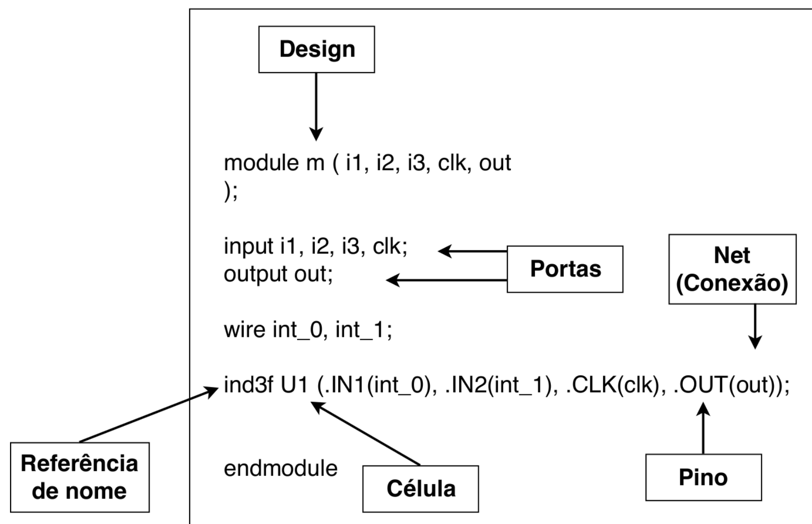
Figura 16 – Modo apropriado de estruturar um Design.



Fonte: O Próprio autor.

- Design: uma descrição do circuito que executa uma ou mais funções lógicas;
- Célula: uma instanciação de um design dentro de outro design;
- Referência de nome: o design original de uma célula;
- Portas: entrada, saída ou entrada/saída do Design

Figura 17 – Objetos de uma netlist



Fonte: O Próprio autor.

- Pino: entrada, saída ou entrada/saída da célula no Design;
- Net: o fio que conecta as portas ao pinos ou vice-versa;
- Clock: porta do Design ou pino da célula definido explicitamente como fonte do *clock*.

2.4.1 Síntese Lógica do AWGN

O bloco AWGN foi construído seguindo as recomendações desta seção, principalmente subdividindo os blocos internos e definindo *constraints* colocadas na Tabela 6, referentes a *Timing* e área, mas para consumo de potência não, já que o interesse desse trabalho foi atingir frequências de clock mais altas, e com isso não há como obter redução de consumo de potência, por isso essa *constraint* ficou livre e ficou a cargo do sintetizador fazer as menores otimizações possíveis.

Tabela 6 – *Constraints* de *Timing* e Área

Timing		Área	
clock	0.666 ns (1.5 GHz)	Otimizar operações compartilhadas	Sim
delay de entrada	0 ns	Otimizar flip-flog mesclado	Sim
delay de saída	0 ns	Preservar estrutura hierárquica	Sim
setup	6.66 ps	-	-
hold	6.66 ps	-	-
latência do clock (subida)	6.66 ps	-	-
latência do clock (descida)	6.66 ps	-	-

2.5 Resultados

Os resultados de síntese são abordados a seguir nas Tabelas 7 e 8. Os resultados de área e *timing* podem ser dados já logo após a síntese lógica, mas o relatório de consumo de potência deve ser feito só após a simulação *Gate-Level*.

Tabela 7 – Resultados de área e *timing*.

Timing		Área	
clock	0.666 ns (1.5 GHz)	Número de Células	4802
slack	0 ns	Área de Células	6879.85 μm^2
latência do clock (subida)	7 ps	Área de Conexões	2468.50 μm^2
latência do clock (descida)	7 ps	Área Total	9348.36 μm^2

Tabela 8 – Relatórios do consumo de potências (mW).

Corner	Interno	Switching	Leakage	Total
0.60V / 25 C	4.721	2.936	0.018	7.675
0.65V / 125 C	5.610	3.480	0.372	9.462
0.75V / 40 C	7.375	4.623	0.001	12.000

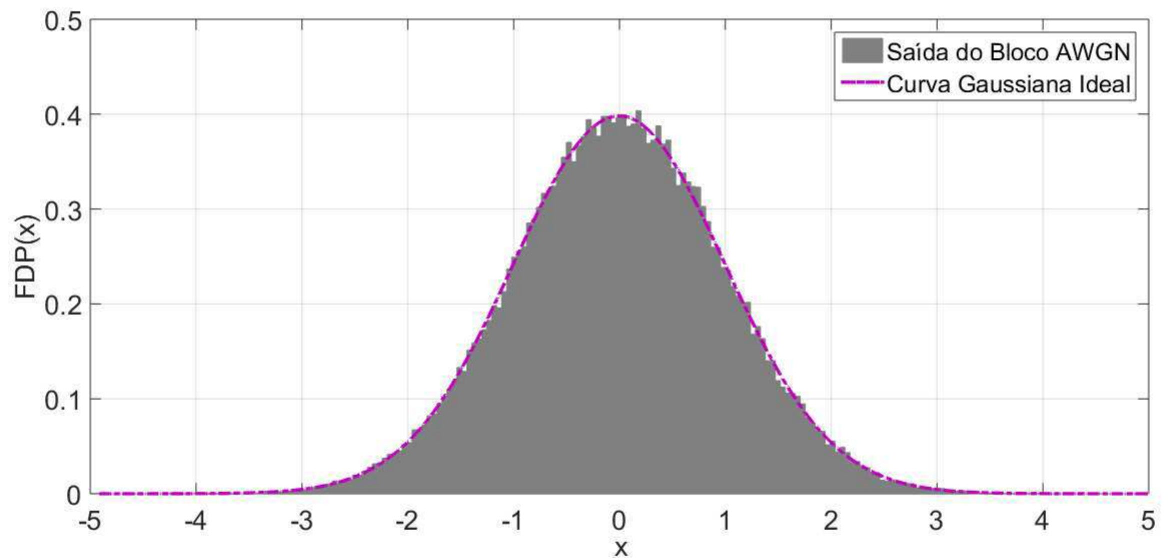
Para análise matemática, foram extraídos os resultados da simulação *Gate-Level* que estavam em complemento de dois e foram convertidos para valores em decimal a partir do código abaixo, desenvolvido na ferramente Octave. Após a conversão é plotado o histograma que ilustrado na Figura 18, representando a Função de Densidade de Probabilidade (PDF).

```
function x = twos2dec(t)

m = 16;
t = dec2bin(t,m);

t = char(t);

x = bin2dec(t);

nbits = m;
xneg = log2(x) >= nbits - 1;
for i=1:length(x)
if(xneg(i) == 1)
    x(i) = -( (bitcmp(x(i),m)) + 1 );
end
end

x = x/2^11;

hist(x,500);
```

Por conter vários estágios de *pipeline*, o bloco AWGN consequentemente tem uma maior latência, necessitando de 12 pulsos de *clock* após o sinal de *reset* para fornecer na saída o primeiro dado válido, e a partir de então, todo pulso de *clock* terá uma saída válida. Esse resultado é ilustrado na Figura 19.

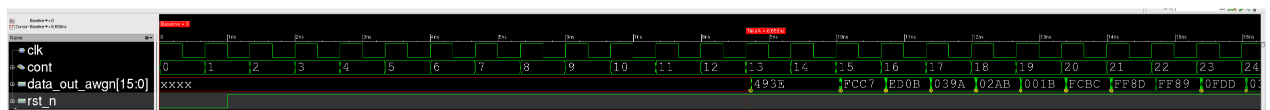
A partir dos resultados visualizados na Figura 18, foi medido o erro ponto a ponto do valor ideal com o valor resultante da saída do AWGN. O Octave possui um pacote de funções voltado para a probabilidade e estatística, e assim foi medida a similaridade da curva ideal com a curva empírica, resultado em 99.78 % de semelhança. Os erros existem pois não é uma solução ideal, já que a resolução de saída do AWGN possui 16 bits, sendo 11 para parte decimal, por isso o erro é de 2^{-11}.

Figura 18 – Comparação de curva Gaussiana ideal com a gerada pelo AWGN.



Fonte: O Próprio autor.

Figura 19 – Formas de ondas da saída do AWGN na ferramenta **simvision**.



Fonte: O Próprio autor.

3 Conclusões

O estudo e análise do AWGN é de grande importância para o área de Engenharia Elétrica, já que este está presente nas transmissões digitais de dados, havendo a necessidade de saber suas características. Observando a dificuldade de usar canais físicos para testar a transmissão e recepção de dados, o uso de um AWGN intregado ao sistema resulta em mais agilidade nos testes, como também reduz os custos.

Para que esse desenvolvimento pudesse ser feito, foram apresentadas diversas soluções e definições de como se deve fazer um projeto de *front-end* voltado para a microeletrônica, com detalhes de cada passo. A especificação ramificada em funcional e arquitetural foram de suma importância para um modelo coerente ao que foi projetado. O design do circuito digital só pôde ser elaborado com eficiência porque a especificação estava bem feita, desde os detalhes de funcionamente até o diagrama de blocos e microarquiteturas. Por fim, a síntese lógica elaborada com as devidas diretrizes para alcançar as especificações funcionais dadas no ínicio.

As especificações definidas no ínicio desse trabalho e os resultados obtidos, demonstram a qualidade do que foi gerado. Para tentar emparelhar com os dispositivos atuais, que muitos já não usam mais frequências de *clock* na casa de MHz e sim de GHz, e que também requerem alto *throughput*, ou seja, necessitam de grande taxa de transferência de dados, foi alcançado nesse trabalho a frequência de 1.5 GHz e com *throughput* de um dado a cada pulso do *clock*.

4 Referências

J.L Danger, A. Ghazel, E. Boutillon H. Laamari, “Efficient FPGA Implementation of Gaussian Noise Generator for Communication Channel Emulation,” The 7th IEEE International Conference on Electronicsm Circuits & Systemes (ICECS’2K), Kaslik, Lebanon, Dec 2000.

J. Dong, "Estimation of Bit Error Rate of any digital Communication System,"Télécom Bretagne, Université de Bretagne Occidentale, 2013. English.

R. Gutierrez, V. Torres, and J. Valls, “Hardware Architecture of a Gaussian Noise Generator Based on the Inversion Method,” IEEE Transactions on Circuits and Systems—II: Express Briefs, Vol. 59, NO. 8, August 2012.

K. Cushon, P. Larsson-Edefors and P. Andrekson, “Low-Power 400-Gbps Soft-Decision LDPC FEC for Optical Transport Networks”, Journal of Lighwave Technology, Vol. 34, NO. 18, September 15, 2016.

P. Atiniramit, “Design and implementation of an FPGA-based adaptative filter single-use receiver,” M.S. thesis, Virginia Polytechnic Inst. State Univ., Blacksburg, VA, 1999.

Y. Fan, Z. Zilic and M. W. Chiang, "A Versatile High Speed Bit Error Rate Testing Scheme,"International Symposium on Signals, Circuits and Systems. Proceedings, SCS 2003. (Cat. No.03EX720).

R. Andraka and R. Phelps, “An FPGA based processor yields a real time high fidelity radar environment simulator,” in Proc. Mil. Aerosp. Appl. Programm. Devices Technol. Conf., 1998.

D. B. Thomas, W. Luk, P. H. W. Leong, and J. D. Villasenor, “Gaussian random number generators,” J. ACM Comput. Surveys (CSUR), vol. 39, no. 4, pp. 11:1–11:38, 2007.

E. Boutillon, J. L. Danger, and A. Gazel, “Design of high speed AWGN communication channel emulator,” Analog Integr. Circuits Signal Process., vol. 34, no. 2, pp. 133–142, Feb. 2003

J. McCollum, J. M. Lancaster, D. W. Bouldin, and G. D. Peterson, “Hard- ware acceleration of pseudo-random number generation for simulation applications,” in Proc. 35th Southeastern Symp. Syst. Theory, 2003, pp. 299–303.

G. Box and M. Muller, “A note on the generation of random normal deviates,” Ann. Math. Statist., vol. 29, pp. 610–611, 1958.

R. C. C. Cheung, D. Lee, W. Luk, and J. Villasenor, “Hardware generation of arbitrary random number distributions from uniform distributions via the inversion method,” IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol. 15, no. 8, pp. 952–962, Aug. 2007.

D. Lee, W. Luk, J. Villasenor, G. Zhang, and P. H. W. Leong, “A hardware Gaussian noise generator using the Wallace method,” IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol. 13, no. 8, pp. 911–920, Aug. 2005.

P. L’Ecuyer, “Maximally equidistributed combined Tausworthe genera- tors,” Math. Comput., vol. 65, no. 213, pp. 203–213, Jan.1996.