Margareth Mee Gomes de Lima

Análise comparativa da precisão em ponto flutuante dos padrões IEEE 754 e Posit.

Campina Grande, Brasil

Maio de 2021

Margareth Mee Gomes de Lima

Análise comparativa da precisão em ponto flutuante dos padrões IEEE 754 e Posit.

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: Gutemberg Gonçalves dos Santos Júnior

Campina Grande, Brasil

Maio de 2021

Agradecimentos

Deus sabe que só cheguei até o fim deste curso por cause Dele, então gostaria de agradecer primeiramente a Ele, a razão da minha esperança. Gostaria de agradecer aos meus pais Adriana e Elias, por tudo o que fizeram e fazem por mim, só estou aqui por causa de vocês, essa conquista é de vocês também. Obrigada aos meus irmãos Gubio e Juliana por me aperrearem. Obrigada aos meus avós, tios e tias, eu não teria chegado até aqui sem vocês. Fazer esse curso longe de casa foi dolorido e vocês foram o meu refúgio.

Obrigada às minhas amigas/irmãs, não tenho como expressar o quanto vocês são importante na minha vida. Cada uma de vocês foi importante durante essa longa jornada, vocês acreditam mais em mim do que eu mesma. Obrigada à Asmaa, Amarílis, Ana Rita, Ivonne, Jamile, Jeovana, Mairly, Mikaela e Rebeca.

Agradeço também aos amigos que fiz ao longo do curso, vocês tornaram tudo mais fácil, leve e divertido. Obrigada à Melissa, Marina, Matheus, Raphael, Safire, Samuel e Ulisses.

"Toda a vida deles neste mundo e todas as suas aventuras em Nárnia haviam sido apenas a capa e a primeira página do livro. Agora, finalmente, estavam começando o Capítulo Um da Grande História que ninguém na terra jamais leu: a história que continua eternamente e na qual cada capítulo é muito melhor do que o anterior."

C.S. Lewis

Resumo

O crescente desenvolvimento tecnológico tem solicitado cada vez mais precisão e eficiência nos cálculos aritméticos. Recentemente, foi proposta uma nova representação de números em ponto flutuante criada como uma alternativa à representação convencional padrão IEEE-754. O presente trabalho realiza uma comparação da precisão em ponto flutuante do padrão IEEE e do novo formato posit para um mesmo número de bits. A fim de realizar essa comparação, foram utilizados os resultados gerados pelo cálculo da FFT de sinais discretos no tempo. Os resultados obtidos para 16 e 32 bits foram avaliados tomando como referência o formato float-64 bits.

Palavras-chave: Padrão IEEE-754; posit; ponto flutuante; precisão.

Abstract

Increasing technological development has demanded more and more precision and efficiency in arithmetic calculations. Recently a new floating-point representation of numbers was proposed to replace the conventional IEEE-754 standard. The present work is based on the comparison of floating-point accuracy of IEEE-754 standard and posit when using the same bit depth. In order to make this comparative analysis, the Fast Fourier Transform was calculated for discrete-time signals. The accuracy of the results generated with float and posit formats were determined relative to the 64 bits data type.

Keywords: IEEE-754 standard; posit; floating-point; accuracy.

Lista de tabelas

Tabela 1 – Valores de useed de acordo es 18

Tabela 2 – Comparação posit-32b e float-32b. 28
Tabela 3 – Comparação posit-16b e float-16b. 30

Lista de ilustrações

Figura 1 – Exemplo de representação em Ponto Fixo de 10,5. 5
Figura 2 – Padrão IEEE 754 para 32 bits. 8
Figura 3 – Representação do número 52,21875 no IEEE 754 para 32 bits. 9
Figura 4 – Maior número em float 32 bits. 9
Figura 5 – Menor número em float 32 bits. 10
Figura 6 – Padrão IEEE 754 para 64 bits. 10
Figura 7 – Maior número em float 64 bits. 10
Figura 8 – Menor número em float 64 bits. 11
Figura 9 – Padrão IEEE 754 para 16 bits. 11
Figura 10 –Padrão IEEE 754 para 128 bits. 12
Figura 11 –+0 Padrão IEEE 754 para 32 bits. 13
Figura 12 –-0 Padrão IEEE 754 para 32 bits. 13
Figura 13 –+Infinito IEEE 754 para 32 bits. 13
Figura 14 –-Infinito IEEE 754 para 32 bits. 13
Figura 15 –QNaN IEEE 754 para 32 bits. 14
Figura 16 –SNaN IEEE 754 para 64 bits. 14
Figura 17 –Exemplo de Unums tipo II - 5 bits. 16
Figura 18 –Representação genérica de um posit de n-bits. 17
Figura 19 –Regime e o valor k 18
Figura 20 –Valores positivos para um posit de 3 bit. 19
Figura 21 –Estrutura para posits com es = 2, $useed = 2^{2^{es}} = 16$ 20
Figura 22 –Exemplo de decodificação de um posit de 16 bits 20

Figura 23 –Desvio padrão para resultados em float-32b para N até 1024. 25
Figura 24 –Desvio padrão para resultados em float-32b para N até 131072. 25
Figura 25 –Desvio padrão para resultados em posit-32b para N até 1024. 26
Figura 26 –Desvio padrão para resultados em posit-32b para N até 131072. 27
Figura 27 –Desvio padrão para float 16b para N até 16384. 29
Figura 28 –Desvio padrão para posit 16b para N até 16384. 29

Lista de abreviaturas e siglas

IEEE - *Institute of Electrical and Electronics Engineers*

FFT - *Fast Fourier Transform*

NaN - *Not a Number*

Unum - *Universal Number*

Sumário

1 Introdução 1
1.1 Objetivos 2
1.2 Metodologia 2
1.3 Organização do trabalho 3

2 Fundamentação Teórica 4
2.1 Representação de números 4
2.1.1 Representação em Ponto Fixo 5
2.1.2 Representação em Ponto Flutuante 6
2.1.3 Padrão IEEE 754 7
2.1.3.1 Padrão IEEE 754 - 32 bits 8
2.1.3.2 Padrão IEEE 754 - 64 bits 9
2.1.3.3 Padrão IEEE 754 - 16 bits 11
2.1.3.4 Padrão IEEE 754 - 128 bits 11
2.1.3.5 Valores Especiais em IEEE 754 12
2.1.4 Unum's 15
2.1.4.1 Posits 17
2.2 Comparação entre Floats e Posits 21

3 Resultados e Análises 23
3.1 FFT 23
3.1.1 FFT em 32 bits 24
3.1.2 FFT em 16 bits 27

4 Conclusões 31

Referências 33

1 Introdução

Quando se trata de representação de números reais, desde os primórdios da computação, os sistemas de computadores realizaram a aproximação de números reais por meio de números em ponto flutuante (PF), mais conhecidos como floats. Apesar de sua vasta utilização, os floats apresentam diversos problemas tais como: desperdício de sequências de bits para representação dos NaN's - *Not-a-Number*, para um float de 32 bits IEEE 754 existem 16.10^6 *bit patterns* para representar um NaN; existência de zero positivo e negativo, onde cada um apresenta comportamentos e sequências de bits diferentes; *overflow* para $\pm$ infinito e *underflow* para 0, o primeiro aumenta o erro relativo por um fator infinito e o segundo leva à perda de informação.

Um exemplo deste último problema pode ser a subtração de números muito próximos, esta situação pode gerar erros de cancelamento catastróficos que ampliam o erro de arredondamento dos operandos. Em outras palavras, a aritmética em ponto flutuante pode silenciosamente levar a uma divergência entre os resultados calculados e os resultados exatos da operação. Na maioria dos casos, essa divergência parece ser inofensiva e aceitável, entretanto, muitos trabalhos mostraram casos onde a aritmética em PF leva a resultados extremamente errados e preocupantes (Kahan, 2004), (Kulish; Miranker, 1986), (Rump, 1987).

Em (Gustafson, 2015) John Gustafson propôs uma nova representação de números reais chamada Unum - *Universal Number*. Os Unums apresentam-se como uma alternativa à utilização dos floats, fornecendo algumas soluções para os problemas presentes na representação em ponto flutuante convencional. Desde a sua criação, várias versões de Unums foram propostas, atualmente tem-se as seguintes versões: tipo I, II e III. O Unum tipo III, mais conhecido como posit, é a versão pensada para competir diretamente com os floats. Diferente das versões anteriores, os posits não usam intervalos para representação de números nem possuem operandos com tamanho variável. Como os floats, eles arredondam se o resultado é inexato, contudo, possuem fortes vantagens em relação aos floats como uma maior faixa de variação dinâmica, maior precisão, resultados idênticos (em bitwise)

em diferentes sistemas, hardware mais simples e tratamento de exceção mais simples (L.Gustafson; I.Yonemoto,). Além disso, os posits não realizam o *overflow* para o infinito nem *underflow* para para 0, e o caso de NaN indica uma ação ao invés de uma sequência de bits.

O presente trabalho apresenta uma análise comparativa entre a precisão nos resultados de cálculos ariméticos usando o padrão IEEE-754 e o formato posit. Deseja-se investigar se o formato posit possui realmente uma maior precisão em relação ao float.

1.1 Objetivos

O presente trabalho consiste em realizar uma comparação da precisão nos cálculos de operações aritméticas usando a representação em ponto flutuante do padrão IEEE-754 e o formato posit. Para tanto, utilizar-se-á os resultados gerados a partir do cálculo da Transformada Rápida de Fourier ou *Fast Fourier Transform* - FFT com decimação no tempo.

Sendo N o número de amostras de um sinal discreto, o cálculo da FFT com decimação no tempo apresenta complexidade $O(N \log_2 N)$, isto é são realizadas $N \log_2 N$ multiplicações. Assim, o algoritmo da FFT proporciona um bom número de operações aritméticas que podem ser usadas para avaliar o desempenho da aritmética de ponto flutuante das representações que estão sendo avaliadas.

Pretende-se realizar a comparação dos formatos usando um tamanho de palavra de 32 bits e 16 bits. Além disso deseja-se observar a precisão quando se varia o número de amostras e a amplitude do sinal no tempo.

1.2 Metodologia

A metodologia empregada neste trabalho baseou-se no cálculo da FFT de sinais discretos no tempo para avaliação da precisão dos resultados. Primeiramente, foi escrito em Python um algoritmo para o cálculo da FFT com decimação no tempo. Este algoritmo foi implementado usado os seguintes formatos: posit-16b, float-16b, posit-32b, float-16b e float-64b.

Os resultados gerados pelo formato float-64b serão considerados como valores de referência, desse modo, para cada resultado será calculado um sinal de diferença entre o resultado obtido e o resultado em float-64b. Para cada sinal de diferença será calculado o desvio padrão, que será usado como parâmetro de avaliação dos cálculos realizados (TERNOVOY et al., 2020). Quanto maior o desvio padrão mais disperso é o sinal de

diferença, e assim, maior é a diferença entre o resulto obtido e a referência. Avaliou-se ainda a influência do número de amostras e a amplitude do sinal do sinal discreto no tempo sobre os resultados obtidos.

1.3 Organização do trabalho

O trabalho está estruturado de acordo com a seguinte sequência:

- Capítulo 01: Introdução, com a apresentação do tema e os seus respectivos objetivos e metodologia utilizada para o desenvolvimento do trabalho.
- Capítulo 02: Fundamentação teórica, apresentando as representações em ponto flutuante convencional e posit.
- Capítulo 03: Apresentação e discussão dos resultados obtidos.
- Capítulo 05: Conclusão e considerações finais sobre o trabalho desenvolvido.

2 Fundamentação Teórica

Este capítulo aborda os conceitos fundamentais necessários ao entendimento do presente trabalho. Inicialmente será apresentado a representação em ponto flutuante definida pelo padrão IEEE 754. Em seguida, será feita uma descrição da representação em posit e seu funcionamento.

2.1 Representação de números

Uma das questões que surgiram com o uso dos computadores foi como os números reais seriam representados dentro dos sistemas digitais, uma vez que os computadores usam, em última análise, apenas 0's e 1's para representar qualquer informação dentro do seu sistema. Foi necessário em dado momento criar um esquema de representação que definisse de maneira padronizada a codificação das informações usando bits. Esses esquemas deveriam definir, por exemplo, o número de bits que seriam usados, o significado de cada bit, qual seria o bit mais significativo, etc.

Outra questão que surge é como os números racionais ou irracionais seriam representados, uma vez que estes possuem uma parte fracionária. No caso dos números irracionais, adiciona-se ainda o fato de possuírem uma precisão infinita. Como é sabido, os computadores não são capazes de representar um número com precisão infinita, pois possuem uma memória finita em seu sistema. Portanto, o esquema de representação deve ser capaz de gerenciar o caso de números fracionários e aqueles com precisão infinita.

Vários esquemas de representação foram concebidos ao longo do desenvolvimento dos sistemas computacionais, tais como: sinal e magnitude, representação em excesso, complemento de um, complemento de dois, representação em ponto fixo e representação em ponto flutuante. Das representações clássicas citadas as únicas capazes de lidar com números fracionários são as duas últimas, as outras só são capazes de lidar com números inteiros. O complemento de dois é um sistema amplamente utilizado nos computadores modernos para aritmética de números inteiros, mas quando se trata de não-inteiros é

necessário recorrer ao ponto-fixo ou ponto flutuante.

2.1.1 Representação em Ponto Fixo

A representação em ponto fixo é assim chamada pelo fato de se ter um número fixo de bits destinados à parte inteira e à parte fracionária, portanto é como se existisse uma vírgula imaginária que separasse as duas partes. Uma vez definido o número de bits destinado à parte inteira e fracionária, não é possível alterá-lo, todo o sistema trabalhará considerando a posição da "vírgula"pré-definida, por isso o nome ponto fixo.

A vírgula assume uma posição fixa independentemente do valor que esteja sendo representado, seja ele muito grande (parte inteira grande) ou muito pequeno (parte fracionária grande). Em outras palavras, não é possível ceder bits à parte fracionária para poder representar um número menor, ou ceder bits à parte inteira para representar um número maior. A figura 1 apresenta um exemplo de representação em ponto fixo do número 10,5.

Figura 1 – Exemplo de representação em Ponto Fixo de 10,5.

Sinal	Expoente				Significando		
0	1	0	1	0	1	0	0
	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}

Fonte: Próprio autor.

Sendo n o número total de bits usados para representar o número, para o dado exemplo n = 8 bits. O número de bits usados para a parte inteira é t = 4 bits e para a parte fracionária é f = 3 bits. O bit de sinal indica se é o número é positivo ou negativo. Observa-se que o total de valores representáveis por tal sistema é 2^n, independentemente da posição da vírgula. Entretanto, a faixa de valores representáveis (mínimo, máximo) depende da posição da vírgula, deve-se, portanto, tomar cuidado para que os resultados das operações aritméticas estejam dentro da faixa de valores representáveis pelo formato escolhido, caso contrário, as operações produzirão resultados errados. Além disso, é importante notar que a faixa de valores não é contínua, isto é, são discretos e separados entre si por uma diferença igual a 2^{-f}.

As operações de soma e subtração em ponto fixo são realizadas exatamente da mesma maneira que para números inteiros. Entretanto, para que sejam somados (ou subtraídos) os número precisam apresentar a mesma posição para a vírgula, do contrário, o resultado gerado seria falso. Operações com números em ponto fixo com diferentes posições

para a vírgula também podem ser feitas, desde que um dos números seja convertido para a representação do outro.

A vantagem de se usar a aritmética de ponto fixo é que ela é é direta e fácil de ser aplicada, pois nada mais é do que operações com inteiros. Em outras palavras, pode-se reutilizar todo o *hardware* feito para a aritmética de inteiros para realizar a aritmética de números reais usando a representação em ponto fixo, o que representa uma economia considerável em termos de área de superfície de componentes digitais.

A desvantagem da representação em ponto fixo consiste no tamanho da faixa de valores e a precisão que consegue-se alcançar, quando comparada com a representação em ponto flutuante. Nesses quesitos a representação em ponto fixo mostra-se limitada e não fornece a precisão exigida em diversas aplicações.

2.1.2 Representação em Ponto Flutuante

Em diversas aplicações, principalmente as que exigem uma maior precisão, a faixa de números representáveis em ponto fixo acaba sendo insuficiente. Um exemplo é quando é necessário representar números muito grandes ou muito pequenos. Para entender esse conceito, imagine um número muito grande, geralmente o que se faz quando se quer representar um número muito grande é usar a notação científica, porque ela condensa várias informações no expoente. Da mesma forma, pode-se pensar na representação em PF como o equivalente da notação científica para sistemas computacionais. Como o próprio nome já expressa o ponto flutuante apresenta uma vírgula "flutuante", isto é, é possível deslocar a posição vírgula de acordo com o valor que deseja-se representar. É esta característica que torna a aritmética em ponto flutuante poderosa. O número em ponto flutuante pode ser representado basicamente por:

$$x = m \cdot e^b \tag{2.1}$$

onde m representa a mantissa, e o expoente e b a base. Os computadores usam quase que exclusivamente a base binária ($b = 2$), portanto, os valores que comumente variam são a mantissa e expoente, podendo assumir valores positivos e negativos. A precisão de um número em ponto flutuante é determinada principalmente pelo número de bits utilizados pela mantissa. Já a faixa de valores representáveis depende do número de bits do expoente.

Os números em ponto flutuante são naturalmente redundantes, pois um mesmo número pode ser representado de maneiras diferentes. Exemplo: $1 \cdot 10^5$, ou $0,1 \cdot 10^6$ ou $10 \cdot 10^4$. Para que não haja essa redundância e um número seja sempre representado da mesma forma é necessário que exista uma forma normalizada de representação, onde as mantissas sejam normalizadas. Uma mantissa é tida como normalizada quando a vírgula é

posta de tal modo que não existe parte inteira, apenas fracionária (no exemplo dado seria o caso $0,1 \cdot 10^6$), sendo o primeiro dígito à direita da vírgula diferente de zero. Em resumo, o processo de normalização é feito facilmente por meio de deslocamentos da mantissa para a direita ou esquerda e, respectivamente, incrementos ou decrementos do expoente.

Diferentes padrões foram criados para representar os números em ponto flutuante. O padrão utilizado depende da escolha do fabricante, da família de computadores, etc. No entanto, o padrão mais difundido e utilizado é o recomendado pelo IEEE - *Institute of Electrical and Electronics Engineers*, o padrão IEEE 754. A norma define formatos para representar números de vírgula flutuante (incluindo zero) e os valores não normalizados, bem como os valores especiais infinito e NaN, com um conjunto de operações de ponto flutuante que trabalham com esses valores. Também especifica quatro modos de arredondamento e cinco exceções (inclusive quando essas exceções ocorrem e o que acontece nesses momentos). A Norma IEEE 754-2019 define os formatos adequados para representar números em ponto flutuante de meia precisão (16bits), precisão simples (32 bits), precisão dupla (64 bits) e quádrupla (128 bits).

2.1.3 Padrão IEEE 754

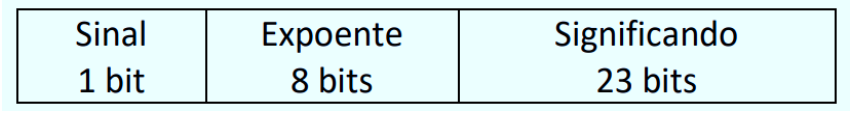
Os números em ponto flutuante começaram a ser usados em meados dos anos 50. Não havia uniformidade nos formatos usados para representar números em PF e os programas não eram portáteis de um fabricante para outro. Em meados da década de 1980, com o advento dos computadores pessoais, o número de bits usados para armazenar números em PF foi padronizado em 32 bits. Um comitê foi formado pelo IEEE para padronizar a representação dos números em PF nos computadores. Além disso, o padrão especificava uniformidade sobre aspectos como arredondamento, tratamento de exceções, como o caso de divisão por 0, representação de 0 e infinito.

Este padrão, denominado IEEE Standard 754 para números em ponto flutuante, foi adotado em 1985 por todos os fabricantes de computadores. O estabelecimento desse padrão permitiu obter a portabilidade de programas de um computador para outro sem que os resultados gerados fossem diferentes. Em sua primeira versão, a norma definiu formatos em ponto flutuante para números de 32 bits e 64 bits. Com o desenvolvimento da tecnologia e o surgimento de novas aplicações, foi criado o formato de 128 bits e de 16 bits. A versão de 1985 sofreu várias alterações desde então e muitas melhorias foram feitas, a versão mais recente é a IEEE 754 -2019.

2.1.3.1 Padrão IEEE 754 - 32 bits

O esquema de representação do padrão IEEE para um número de 32 bits (chamado de *single precision*) pode ser visualizado na Figura 2. Os 32 bits são divididos em: sinal (1 bit), mantissa ou significando (23 bits) e expoente (8 bits).

Figura 2 – Padrão IEEE 754 para 32 bits.



Fonte: Próprio autor.

Para aumentar a precisão do significando, o padrão IEEE 754 usa um significando normalizado, isso significa que seu bit mais significativo é sempre 1 (binário), o que é lógico, pois não faria sentido representar um 0 a esquerda. Já que o bit mais significativo é sempre 1, esse bit fica implícito na representação e presume-se que esteja à esquerda da vírgula do significando. Desse modo, no padrão IEEE para 32 bits, o significando tem 24 bits: 23 bits do significando são armazenados na memória e um bit '1' fica implícito como o 24º bit (mais significativo).

O expoente pode assumir valores positivos e negativos, para tanto, o sinal pode ser representado de duas formas. a primeira consiste em separar 1 bit dos 8 bits do exponente para representar o sinal, desse modo o exponente poderia assumir valores entre -127 a +127. A desvantagem desse método é que há duas representações para o exponente 0: +0 e -0. Para evitar isso, usa-se o formato *biased*, onde a faixa de variação torna-se 0 - 255. O valor do *bias* é 127. Portanto, um expoente igual a 0 significa que –127 está armazenado no campo do expoente. Um valor armazenado 198 significa que o valor do expoente é (198 - 127) = 71. Feitas essas considerações, o número em ponto flutuante de 32 bits pode ser visto como:

$$(-1)^s \cdot (1, f) \cdot 2^{e-127} \tag{2.2}$$

onde s é o bit do sinal, s = 0 é usado para números positivos e s = 1 para representar números negativos; f representa os bits do significando; e representa os bits do expoente. Observe que o bit '1' implícito do significando é mostrado explicitamente para maior clareza.

Exemplo: Representação do número decimal 52,21875 no IEEE 754-32bits.

$52,21875$ (decimal) $= 110100,00111$

(desloc. da vírgula) $= 1,1010000111 \cdot 2^5$
Significando $= 1010000111$
Expoente: $(e - 127) = 5$, $e = 132$

A representação binária do número 52,21875 pode ser vista na Figura 3. O maior

Figura 3 – Representação do número 52,21875 no IEEE 754 para 32 bits.

Sinal 1 bit	Expoente 8 bits	Significando 23 bits
0	10000100	10100001110000000000000

Fonte: Próprio autor.

número representável no formato de 32 bits pode ser visto na Figura 4, o valor de cada campo é descrito abaixo:

Significando: $1111...1 = 1 + (1 - 2^{-23}) = 2 - 2^{-23}$
Expoente: $(254 - 127) = 127$
Valor: $(2 - 2^{-23}).2^{127} \cong 3,403.10^{38}$

Figura 4 – Maior número em float 32 bits.

Sinal 1 bit	Expoente 8 bits	Significando 23 bits
0	11111110	11111111111111111111111

Fonte: Próprio autor.

O menor número normalizado representável no formato de 32 bits pode ser visto na Figura 5, o valor de cada campo é descrito abaixo:

Significando: 1.0
Expoente: $(1 - 127) = -126$
Valor: $(2^{-126} \cong 1,1755^{-38})$

2.1.3.2 Padrão IEEE 754 - 64 bits

O padrão em ponto flutuante IEEE 754 para números de 64 bits (dupla precisão) é muito semelhante ao padrão de 32 bits. A principal diferença é o número de bits alocados para o expoente e o significando. Com 64 bits disponíveis para armazenar números em PF,

Figura 5 – Menor número em float 32 bits.

Sinal 1 bit	Expoente 8 bits	Significando 23 bits
0	00000001	00000000000000000000000

Fonte: Próprio autor.

há mais liberdade para aumentar o número de bits do significando e aumentar a faixa de variação do expoente. Este padrão aloca 1 bit para o sinal, 11 bits para o expoente e 52 bits para o significando. O expoente usa um *bias* de 1023. Esta representação é mostrada logo abaixo:

Figura 6 – Padrão IEEE 754 para 64 bits.

Sinal 1 bit	Expoente 11 bits	Significando 52 bits

Fonte: Próprio autor.

O maior número representável no formato de 64 bits pode ser visto na Figura 7, o valor de cada campo é descrito abaixo:

Significando: $1111...1 = 2 - 2^{-52}$
Expoente: $(2046 - 1023) = 1023$
Valor: $(2 - 2^{-52}).2^{1023} \cong 10^{3083}$

Figura 7 – Maior número em float 64 bits.

Sinal 1 bit	Expoente 11 bits	Significando 52 bits
0	11111111110	1111.....1111

Fonte: Próprio autor.

O menor número normalizado representável no formato de 64 bits pode ser visto na Figura 8, o valor de cada campo é descrito abaixo:

Significando: $000...0 = 2 - 2^{-52}$

Expoente: $(1 - 1023) = -1022$
Valor: 2^{-1022}

Figura 8 – Menor número em float 64 bits.

Sinal 1 bit	Expoente 11 bits	Significando 52bits
0	00000000001	0000000000000000000000

Fonte: Próprio autor.

2.1.3.3 Padrão IEEE 754 - 16 bits

O esquema de representação do padrão IEEE 754 para 16 bits pode ser visto na figura 9, 5 bits são usados para o expoente e 10 bits para o significando. O expoente usa um *bias* de 15. Portanto, o intervalo do expoente é: –14 a +15.

Figura 9 – Padrão IEEE 754 para 16 bits.

Sinal 1 bit	Expoente 5 bits	Significando 10 bits

Fonte: Próprio autor.

O maior número positivo que pode ser representado é:

$$1,111..1 \cdot 2^{15} = 1 + (1 - 2^{-11}) \cdot 2^{15} = (2 - 2^{-11}) \cdot 2^{15} \cong 65504.$$

O menor número normalizado que pode ser representado é:

$$+1,000\ldots0 \cdot 2^{-14} = 2^{-14} = 0,61 \cdot 10^{-4}.$$

O menor número subnormal que pode ser representado é:

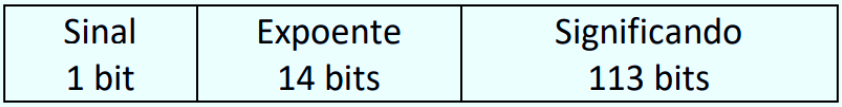
$$2^{-24} \cong 5,96 \cdot 10^{-8}$$

2.1.3.4 Padrão IEEE 754 - 128 bits

Com o avanço da tecnologia, representar números em PF usando 128 bits tornou-se viável. Essa representação é usada normalmente em computação numérica intensiva, onde grandes erros de arredondamento podem ocorrer. Neste padrão, além do bit de sinal, 14

bits são usados para o expoente e 113 bits são usados para o significando. A representação é mostrada na Figura 10.

Figura 10 – Padrão IEEE 754 para 128 bits.



Fonte: Próprio autor.

O maior número positivo que pode ser representado é:

$$(1 - 2^{-113}) \cdot 2^{16383} \cong 10^{4932}$$

O menor número normalizado que pode ser representado é:

$$2^{1-16383} = 2^{-16382} \cong 10^{-4931}$$

O menor número subnormal que pode ser representado é:

$$2^{-16382-113} = 2^{-16485}$$

2.1.3.5 Valores Especiais em IEEE 754

A seguir são destacadas algumas situações especiais que ocorrem no padrão IEEE 754.

Representação do Zero: Como o significando é assumido como tendo um 1 oculto como o bit mais significativo, todos os 0's na parte do significando do número serão considerados como 1,00... 0. Portanto, zero é representado no padrão IEEE com 0s em todos os bits do expoente e do significando. Todos os 0s para o expoente não podem ser usados para nenhum outro número. Se o bit de sinal for 0 e todos os outros bits 0, o número será +0. Se o bit de sinal for 1 e todos os outros bits 0, será –0. As representações de +0 e –0 podem ser vistas na Figuras 11 e 12.

Representação do Infinito: O infinito é representado quando todos os bits no campo do expoente são iguais a 1. O bit de sinal 0 presenta + ∞ e o bit de sinal 1 representa $-\infty$. As representações de $+\infty$ e $-\infty$ podem ser vistas nas Figuras 13 e 14.

Representação de NaN: um NaN - *Not a Number* ocorre quando se realiza uma operação que não é matematicamente possível ou indeterminada. Por exemplo, divisão de zero por zero, o resultado será indeterminado. O padrão IEEE define dois tipos de

Figura 11 – +0 Padrão IEEE 754 para 32 bits.

Sinal 1 bit	Expoente 8 bits	Significando 23 bits
0	00000000	00000000000000000000000

Fonte: Próprio autor.

Figura 12 – -0 Padrão IEEE 754 para 32 bits.

Sinal 1 bit	Expoente 8 bits	Significando 23 bits
1	00000000	00000000000000000000000

Fonte: Próprio autor.

Figura 13 – +Infinito IEEE 754 para 32 bits.

Sinal 1 bit	Expoente 8 bits	Significando 23 bits
0	11111111	00000000000000000000000

Fonte: Próprio autor.

Figura 14 – -Infinito IEEE 754 para 32 bits.

Sinal 1 bit	Expoente 8 bits	Significando 23 bits
1	11111111	00000000000000000000000

Fonte: Próprio autor.

NaN. Quando o resultado de uma operação não é definido (ou seja, indeterminado), ele é chamado de quiet-NaN (QNaN). Os exemplos de QNaN são: $0/0, (\infty - \infty), \sqrt{-1}$. O outro tipo de NaN é chamado de signaling-Nan (SNaN), usado para fornecer uma mensagem de erro. Quando uma operação leva a um *underflow* negativo, ou seja, o resultado de um cálculo é menor do que o menor número que pode ser representado em PF, ou o resultado é um *overflow*, ou seja, é maior do que o maior número que pode ser representado, o SNaN é usado. Outro exemplo de SNaN, é quando não se atribui um valor válido a uma variável

declarada e há uma tentativa de usá-la em uma operação aritmética, o resultado desta tentativa será um SNaN. O QNaN é representado por 0 ou 1 como bit de sinal, todos os bits do expoente iguais a 1, um 0 como o bit mais à esquerda do significando e pelo menos um 1 no resto do significando. SNaN é representado por 0 ou 1 como bit de sinal, todos os bits do expoente iguais a 1, um 1 como o bit mais à esquerda do significando e qualquer sequência de bits para os 22 bits restantes. A representações de QNaN e SNaN são vistas nas Figuras 15 e 16.

Figura 15 – QNaN IEEE 754 para 32 bits.

Sinal 1 bit	Expoente 8 bits	Significando 23 bits
0 ou 1	11111111	00010000000000000000000

Fonte: Próprio autor.

Figura 16 – SNaN IEEE 754 para 64 bits.

Sinal 1 bit	Expoente 8 bits	Significando 23 bits
0 ou 1	11111111	10000000000010000000000

Fonte: Próprio autor.

Números subnormais: um número é dito subnormal quando todos os bits do expoente são iguais a 0 e o bit oculto inicial do significando é 0. Assim, a representação de um número subnormal para uma palavra de 32 bits é descrita na equação 2.3.

$$(-1)^s \cdot (0, f) \cdot 2^{-127} \tag{2.3}$$

O significando f começa obrigatoriamente com um bit 0, mas depois deve apresentar pelo menos um bit igual a 1, caso contrário, o número será considerado como 0. No entanto, o padrão usa -126, ou seja, bias + 1 para o expoente em vez de -127, por alguma razão não tão óbvia, possivelmente porque usando -126 em vez de -127, a lacuna entre o maior número subnormal e o menor número normalizado são menores. Sendo assim, o maior número subnormal é $0,99999998 \cdot 2^{-126}$, que é bem próximo ao menor número normalizado 2^{-126}.

O menor número subnormal positivo é o valor de $2^{-23} \cdot 2^{-126} = 2^{-149}$. Um resultado menor do que o menor número que pode ser representado é chamado de *underflow*. Os

números subnormais são permitidos no padrão IEEE para que haja um *underflow* gradual. Além disso, o menor número que poderia ser representado na máquina é mais próximo de zero.

2.1.4 Unum's

Em 2015 John L. Gustavson propôs uma nova representação em ponto flutuante chamada Unum - *Universal Number*, que lida com alguns dos problemas presentes no padrão IEEE 754. A representação Unum faz distinção entre números que podem ser representados de maneira exata e os que não podem. No último caso, utiliza-se um intervalo aberto entre dois números reais adjacentes para representar um número que está contido dentro deste intervalo, mas que não pode ser representado de maneira exata. A aritmética Unum leva, portanto, em consideração se os operandos são números exatos ou intervalos. Além disso, ao contrário da representação em FP convencional, o Unum define um formato de comprimento variável que adapta dinamicamente o tamanho de bits da representação ao valor real a ser representado.

Até o presente momento existem três versões de Unums. O Unum tipo I é um superconjunto do formato em ponto flutuante do padrão IEEE 754, ele usa um "ubit" no final da fração para indicar se um número real é um float exato ou se é representado por um intervalo aberto entre floats adjacentes. A representação do tipo I apresenta três campos: sinal, expoente e fração. A grande diferença em relação ao padrão IEEE 754 é o fato de poder-se variar de modo automático os números de bits do expoente e fração, respeitando um valor máximo definido pelo usuário. Os Unums do tipo I oferecem uma maneira compacta de expressar a aritmética de intervalo, entretanto, seu comprimento variável exige gerenciamento extra. O tipo I herda algumas desvantagens do padrão IEEE 754, como representações redundantes e desnecessárias.

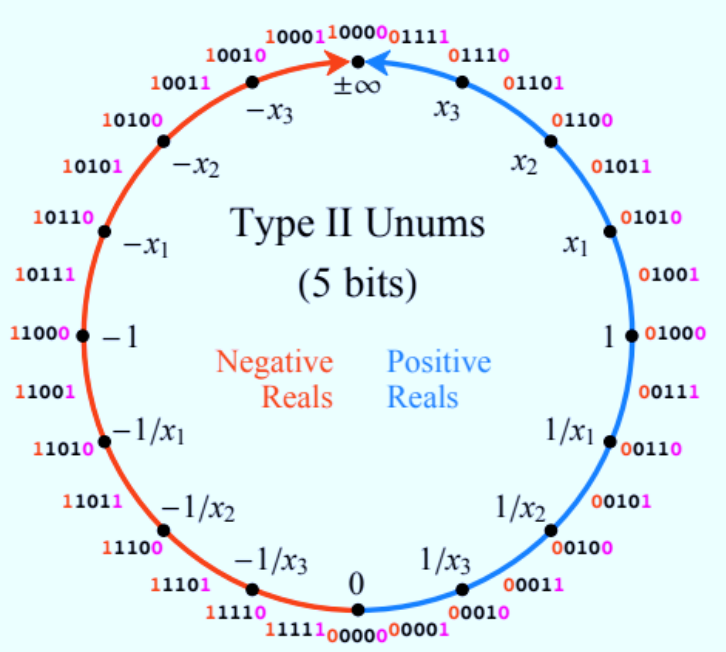
O Unum tipo II não é compatível com o padrão IEEE 754 e foram projetados visando oferecer uma maior velocidade e simplicidade em comparação com a versão anterior. Diferente do tipo I, o tipo II tem tamanho fixo para tentar evitar a complexidade de gerenciar um formato com tamanho variável. Eles têm uma representação decimal sem penalidade no desempenho e capacidade de obter o recíproco de um número com a mesma rapidez e facilidade com que podemos negar um número. Operações como adição, subtração, multiplicação, divisão e até potenciação podem ser feitas em um único ciclo de clock. Existe apenas uma maneira de representar um dado número real, o que é vantajoso quando se quer verificar se dois Unums representam o mesmo valor.

O funcionamento do Unum tipo II baseia-se na utilização de uma *lookup table* ou tabela de correspondência. O exemplo para um Unum de 5 bits pode ser visto na Figura

17. O número de bits para cada Unum é $n = 5$ bits, o quadrante superior direito do círculo contem um conjunto de de $2^{n-3} - 1$ números reais x_i (não necessariamente racionais). O quadrante superior esquerdo tem os seus correspondentes negativos, para encontrar o valor negativo é necessário apenas fazer uma reflexão sobre o eixo vertical. A metade inferior do círculo mantém os recíprocos dos números da metade superior (uma reflexão sobre o eixo horizontal), tornando as operações de multiplicação e divisão tão simétricas quanto adição e subtração. Observe que o bit mais significativo (em cor laranja) corresponde ao bit de sinal ('0' para positivo e '1' para negativo). O último bit corresponde ao ubit (em cor rosa) indica se o número é exato (ubit = '0') ou um intervalo (ubit = '1').

Os unums do tipo II têm muitas propriedades matemáticas ideais, mas dependem da consulta da tabela de correspondência para realizar a maioria das operações. Se eles tiverem n bits de precisão, haverá (no pior caso) 2^{2n} entradas para funções de 2 argumentos, embora as simetrias e outros aspectos acabem reduzindo esse valor. O tamanho da tabela limita o formato em cerca de 20 bits ou menos, considerando a tecnologia de memória atualmente disponível. Essas desvantagens motivaram a busca por um formato que mantivesse muitos dos méritos dos Unums tipo II, mas que fosse *hardware friendly*, isto é, computável usando a lógica de ponto flutuante existente.

Figura 17 – Exemplo de Unums tipo II - 5 bits.



Fonte: (L.Gustafson; I.Yonemoto,)

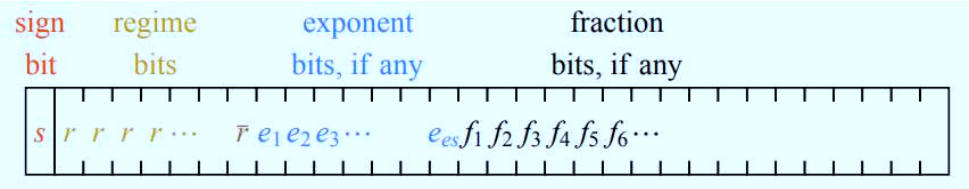
O Unum tipo III, mais conhecido como posit, combina muitas das vantagens do tipo I e tipo II, mas se destaca por ser menos radical e por ter sido projetado para uma

substituição imediata do float padrão IEEE 754, sem mudanças necessárias no código fonte. O tipo III utiliza um número fixo de bits, apesar disso ele se mostra muito flexível, sendo possível utilizar de dois até milhares de bits. Eles são projetados para uma implementação simples de hardware e software e usam o mesmo tipo de construção de circuito de baixo nível do float IEEE 754 (adições de inteiros, multiplicações de inteiros, deslocamentos, etc). Em geral, ocupam menos área de chip, pois são mais simples do que os floats em muitos aspectos.

2.1.4.1 Posits

A figura 18 mostra a estrutura de representação de um posit de n-bits.

Figura 18 – Representação genérica de um posit de n-bits.



Fonte: (L.Gustafson; I.Yonemoto,)

Observe que há quatro campos distintos. O campo de sinal (em vermelho), para o qual '0' = positivo, '1' = negativo. Se o número for negativo é necessário fazer complemento de 2 antes de decodificar os campos regime, expoente e fração. O campo de regime (em amarelo) são os bits que vêm após o bit de sinal, sua função será explicada mais adiante. Em seguida, tem-se o campo do expoente (em azul), considerado como um inteiro sem sinal, o número máximo de bits que podem ser usados para o expoente é es, mas isso não significa que todo número usa es bits para sua representação. Por fim, tem-se o campo da fração, onde considera-se que há um bit oculto igual a 1, seguindo a mesma lógica adotada pelos floats. Para entender os bits de regime, considere as sequências binárias mostradas na Figura 19. O campo regime começa com uma sequência de 0's ou de 1's, e termina quando se encontra um bit oposto ou quando o final da sequência é alcançado. Portanto, o que temos é uma sequência de bits r idênticos (em amarelo) e o bit oposto (em marrom). A partir do regime será definido o valor de uma constante k. Seja m o número de bits idênticos, se for uma sequência de 0's, então $k = -m$, se for uma sequência de 1's, então $k = m - 1$. A maioria dos processadores é capaz de realizar a função "encontrar o primeiro 1" ou "encontrar o primeiro 0" em *hardware*, portanto, a lógica de decodificação para bits de regime é viável.

Figura 19 – Regime e o valor k

Binary	0000	0001	001x	01xx	10xx	110x	1110	1111
Numerical meaning, *k*	−4	−3	−2	−1	0	1	2	3

Fonte: (L.Gustafson; I.Yonemoto,)

O regime indica um fator de escala de $useed^k$, onde $useed = 2^{2^{es}}$ e es é o número máximo de bits para o campo do expoente. A Tabela 1 mostra valores de useed em função de valores de es.

Tabela 1 – Valores de useed de acordo es

es	0	1	2	3	4
useed	2	$2^2 = 4$	$2^4 = 16$	$2^8 = 256$	$2^16 = 65536$

Fonte: (L.Gustafson; I.Yonemoto,)

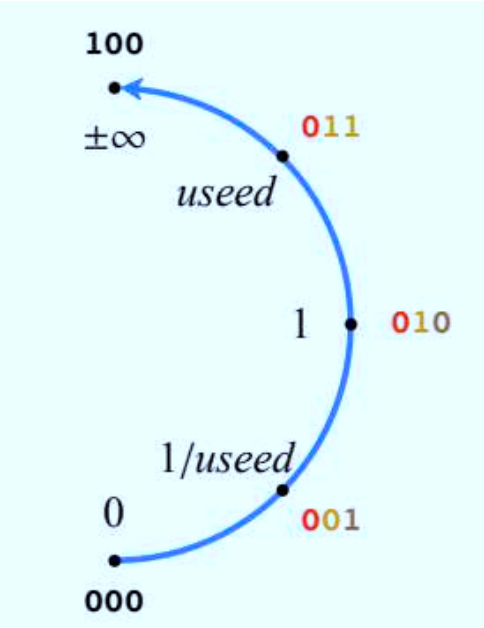
O campo do expoente é considerado como um inteiro sem sinal. Não há *bias* como no caso do float convencional. O expoente tem um tamanho variável, mas possui um valor máximo igual a es, o número de bits que o expoente usará dependerá de quantos bits restam após o bit de sinal e os bits do regime. O tamanho variável do expoente gera uma espécie de precisão cônica, onde números próximos a 1 em magnitude têm mais precisão do que números extremamente grandes ou extremamente pequenos, que são bem menos comuns em cálculos. Isso ficará mais claro com os próximos exemplos.

Se houver quaisquer bits restantes após os campos de sinal, regime e expoente, eles serão usados para representar a fração f. Da mesma forma que para o float convencional existe um bit oculto igual a 1. Uma das vantagens de se usar posits é que não há números subnormais como existem para o padrão IEEE 754, isso simplifica ainda mais a lógica de representação de posits.

Para ilustrar o funcionamento da representação em posit, considerar-se-á o exemplo da Figura 20, onde temos posits de 3 bits. Observe que, para maior clareza, a figura mostra apenas a metade direita, que é positiva. Existem apenas dois valores de exceção para posits: 0 (todos os bits iguais a 0) e $\pm\infty$ (1 seguido de 0's). Observe que os bits seguem a codificação por cores usada anteriormente: sinal(vermelho), regime (amarelo e marrom). Como o tamanho da palavra é muito pequeno (3 bits), não sobra espaço para os campos de expoente e fração. Observe que os valores positivos são dados por $useed^k$, onde $useed$ é calculado por $useed = 2^{2^{es}}$, e k é definido pelo campo de regime. Na figura 20 para "011", o sinal é '0' = positivo, o regime é igual a "11", o que faz $k = 2 - 1 = 1$, ou seja, "011"=

useed. Para "001", o sinal é '0' = positivo, o regime é igual a "01", o que faz k = -1, ou seja, "001" = $useed^{-1} = 1/useed$.

Figura 20 – Valores positivos para um posit de 3 bit.



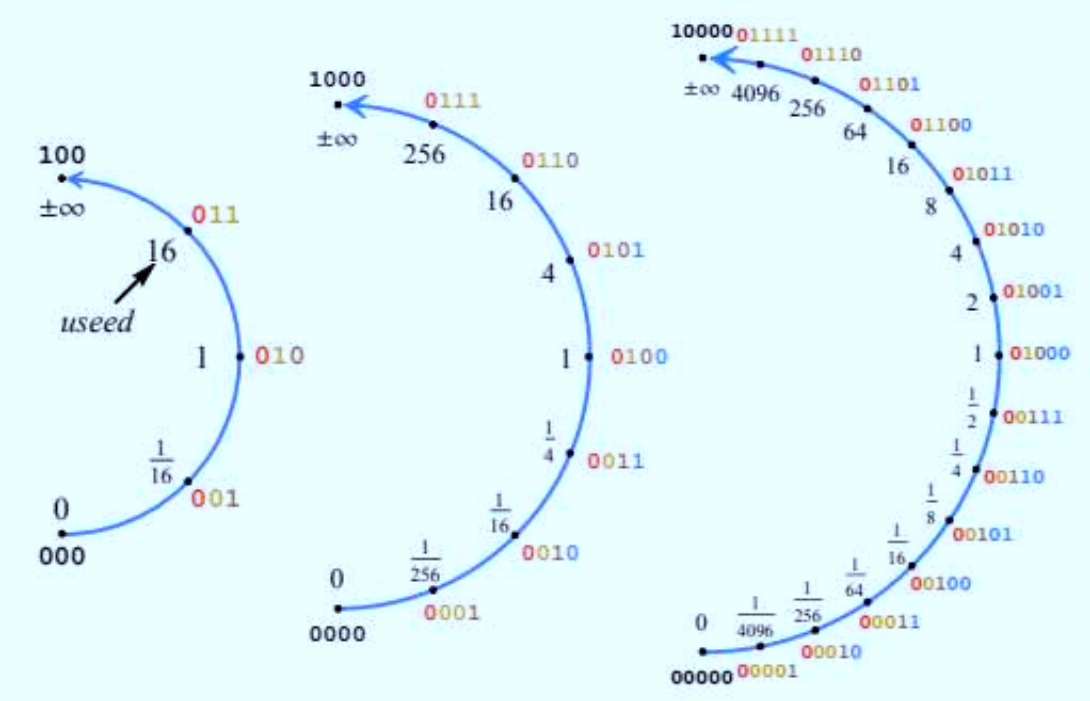
Fonte: (L.Gustafson; I.Yonemoto,)

A Figura 21 mostra o que ocorre quando o número de bits aumenta, temos na sequência 3 bits, 4 bits e 5 bits. Observe que es = 2, portanto, $useed = 2^{2^{es}} = 16$. Ao passar de 3 bits para 4 bits, 1 bit é adicionado. Quando o bit adicionado é 0, o valor permanece o mesmo (caso do número 16). Se o bit adicionado for '1', então um novo valor é criado entre os dois valores subjacentes anteriores (caso do número 4). Observe também que os bits do expoente (em azul) só aparecem a partir do posit de 4 bits e para valores próximos ao valor 1 (precisão cônica). Em nenhum dos 3 casos há bits para o campo de fração.

Supondo que a string de bits de um posit p seja um inteiro com sinal que varia de -2^{n-1} a $2^{n-1} - 1$. Seja k o inteiro representado pelos bits de regime, e o inteiro sem sinal representado pelos bits do expoente. Se o conjunto de bits de fração for $f_1 f_2 ... f_{fs}$, seja f o valor representado por $1.f_1 f_2 ... f_{fs}$. Então p representa:

$$x = \begin{cases} 0, & \text{para } p = 0 \\ \pm\infty, & \text{para } p = -2^{n-1} \\ sign(p) \cdot useed^k \cdot 2^e \cdot f, & \text{para qualquer outro p} \end{cases} \tag{2.4}$$

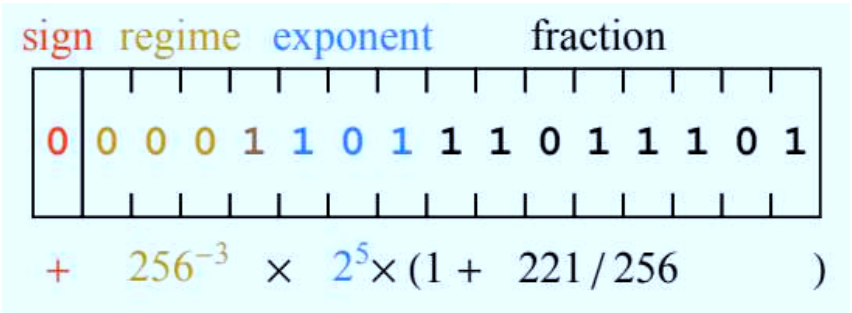
Figura 21 – Estrutura para posits com es = 2, $useed = 2^{2^{es}} = 16$



Fonte: (L.Gustafson; I.Yonemoto,)

O regime e os bits *es* cumprem a função dos bits expoentes em um float padrão. Juntos, eles definem a escala de potência de 2 da fração, onde cada incremento useed é um deslocamento de 2^{es}. Os valores máximo e mínimo do posit são $maxpos = useed^{n-2}$ e $minpos = 2 - n$. Um exemplo de decodificação de um posit é mostrado na Figura 22:

Figura 22 – Exemplo de decodificação de um posit de 16 bits



Fonte: (L.Gustafson; I.Yonemoto,)

O bit de sinal '0' significa que o valor é positivo. Os bits de regime "0001"têm três 0's, o que significa que $k = -3$, portanto, o fator de escala gerado pelo regime é 256^{-3}. Os bits do expoentes "101", representam 5 como um inteiro binário sem sinal e contribuem com

outro fator de escala de 2^5. Por último, os bits de fração 11011101 representam 221 como um inteiro binário sem sinal, então o valor da fração é $1 + 221/256$. O valor correspondente a esta representação em posit é $477/134217728 \cong 3,55393 \cdot 10^{-6}$

2.2 Comparação entre Floats e Posits

Apesar da sua vasta utilização, a representação em ponto flutuante do padrão IEEE apresenta alguns dificuldades, a saber:

- **Sequências de bits desperdiçadas**: o formato de ponto flutuante IEEE-754 de 32 bits tem cerca de $16 \cdot 10^6$ maneiras de representar um NaN, enquanto que o formato de 64 bits tem $9 \cdot 10^{18}$. Lembrando que um NaN representa uma exceção, como uma divisão por zero ou uma operação indefinida. Sendo assim, o padrão IEEE 754 acaba desperdiçando muitas sequências que poderiam estar sendo usados para representar outros valores.

- **Matematicamente incorreto**: o padrão IEEE-754 tem duas formas de representar o valor zero: um zero negativo e positivo, onde cada um possui comportamentos diferentes e sequências de bits diferentes.

- ***Overflow* para $\pm\infty$ e *underflow* para 0**: quando ocorre um *overflow* para $\pm\infty$ o erro relativo aumenta por um fator também infinito. Já um *underflow* para 0 leva à perda de informação do sinal.

- **Complexidade dos circuitos**: uma das razões pelas quais os pontos flutuantes IEEE 754 têm circuitos complicados é porque o padrão define o suporte para números subnormais (uma maneira de aumentar o número próximos a zero que podem ser representados em ponto flutuante, a fim de melhorar o precisão dos cálculos). O ponto flutuante para número subnormais tem um bit oculto de valor '0' em vez de '1'.

- **Sem *overflow* gradual e precisão fixa**: se a precisão for definida como o número de bits do significando, o padrão IEEE-754 tem precisão fixa para todos os números, exceto para números subnormais. Os números subnormais têm um número reduzido de dígitos significativos quando o valor se aproxima de zero. Os números subnormais preenchem o intervalo de underflow, isto é, o intervalo entre zero e os menores valores diferentes de zero. Assim, os floats têm um underflow gradual, em contrapartida o overflow gradual que não existe no padrão IEEE-754 (Goldberg, 1991).

Em contraste com o formato em PF, o formato posit apresenta as seguintes características:

- **Econômico**: nenhuma sequência de bits é redundante. Há uma única represen-

tação para infinito denotada por $\pm\infty$. Todos os outros padrões de bits são números reais diferentes de zero, distintos e válidos. $\pm\infty$ serve como um substituto para NaN.

- **Matemática elegante**: há apenas uma representação para zero.

- ***Overflow* e *underflow* graduais**: o número de dígitos do significando não são fixos para o formato posit. Na verdade, uma maior magnitude de expoente, reduz automaticamente o número de dígitos do significando, o que permite obter *overflow* e *underflow* graduais.

- **Circuitos simples**: não há números subnormais para o formato posit. Isso simplifica consideravelmente os circuitos desenvolvidos para trabalhar com este padrão. O bit implícito é sempre '1'.

3 Resultados e Análises

Após o estudo das representações em ponto flutuante float e posit, passou-se ao desenvolvimento do algoritmo de uma FFT com decimação no tempo usando a linguagem de programação Python a fim de comparar os resultados gerados em posit e float. Este capítulo apresenta os resultados dessas comparações.

3.1 FFT

No intuito de comparar os formatos de representação posit e float foi feita uma função em Python para o cálculo da Transformada Rápida de Fourier ou FFT. O algoritmo usado para escrever esta função foi baseado no trabalho de Cooley e Tukey (1965), que estabeleceu uma técnica conhecida como decimação no tempo, usada para eliminar cálculos redundantes na FFT. Para uso do números no formato posit em Python utilizou-se a biblioteca Numpy-Posit (acesso: 21/04/2021).

A FFT foi aplicada a um sinal discreto de onda quadrada $x(n)$ definido na equação 3.1, $x(n)$ possui N amostras:

$$x(n) \quad = \quad A \cdot square\left(\frac{2 \cdot \pi \cdot f_1 \cdot n}{f_s}\right) \tag{3.1}$$

onde A é uma constante que altera a amplitude do sinal, f_1 é a frequência do sinal, f_s é frequência de amostragem e $0 < n < N - 1$. O valor de $\log_2 N$ deve ser um número inteiro.

Através da variação desses parâmetros pode-se construir várias versões do sinal de onda quadrada para cálculo da FFT. Para este trabalho, é interessante observar a influência do aumento de N nos resultados da FFT, pois N determina o número de multiplicações dada por $Nlog_2N$. Assim, quanto maior o número de N maior será o número de multiplicações efetuadas. Outro parâmetro interessante é a amplitude A, pois permite verificar a capacidade de representação do formato que está sendo analisado. Desse modo, será feito uma série de testes variando tanto N quanto A para observar a influência desses parâmetros nos resultados obtidos em posit e em float.

Para realizar as comparações foi necessário tomar um valor de referência a ser considerado como o valor correto. Os valores definidos como valores de referência foram aqueles gerados pelo algoritmo da FFT usando float-64 bits. Desse modo, a cada comparação foi calculado o sinal de diferença entre o resultado obtido e o resultado de referência (equação 3.2).

Para cada sinal de diferença foi calculado o desvio padrão, usado como critério de avaliação dos resultados. Sabe-se que o desvio padrão de um sinal indica a dispersão dos seus valores, assim, quanto maior o valor do desvio padrão mais disperso está o sinal de diferença, isto é, mais diferente ele está do resultado em 64 bits. Portanto, o que se espera é que o desvio padrão seja o mais próximo de 0 quanto possível, quanto mais próximo de 0 melhor será o resultado.

$$sinal_{diferença} = saída - saída_{float-64b} \tag{3.2}$$

3.1.1 FFT em 32 bits

Para os testes em 32 bits foram gerados diferentes sinais de acordo com a definição 3.1, variando-se N e A conforme os valores a seguir:

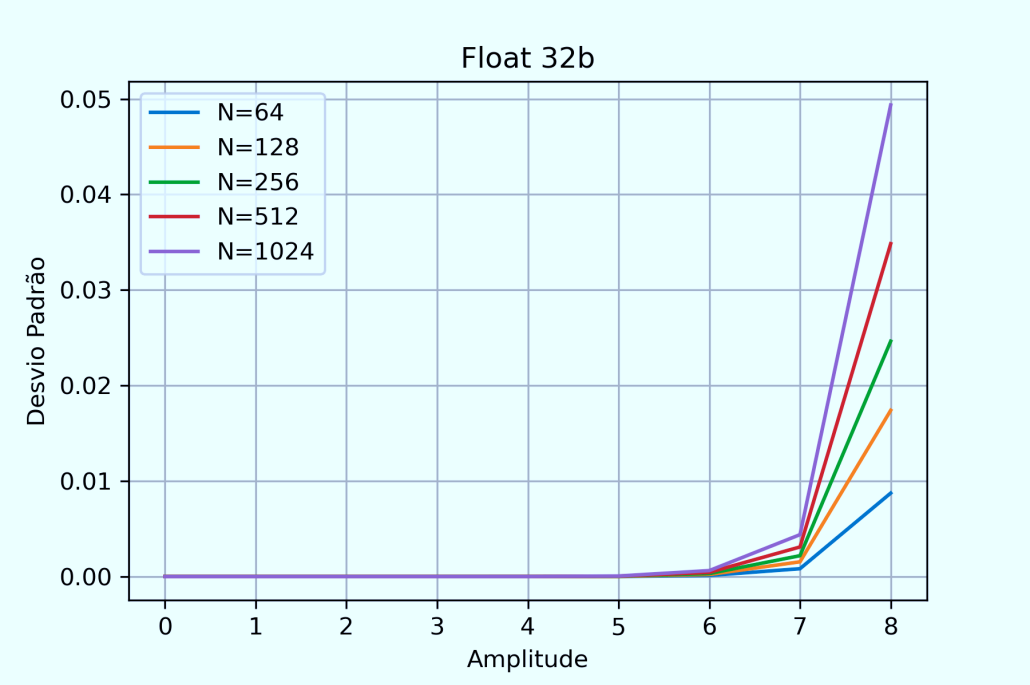
$$N = \{64, 128, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072\}$$
$$A = \{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1, 10, 100, 1000, 10^{4}\}$$

A figura 23 mostra o desvio padrão dos resultados para a FFT em float-32b. Observe que foram usados diferentes valores de N, e para cada N, diferentes valores de amplitude A. Os valores de A são indicados por meio de índices $0, 1, 2...8$ e correspondem aos valores já mencionados em ordem crescente. Primeiramente, percebe-se que para $A \leq 10^2$ (índice = 6), a diferença dos resultados para os diversos valores de N não são perceptíveis no gráfico, pois são valores muito pequenos. Para $A > 10^2$ os valores de desvio padrão crescem numa velocidade bem maior quando aumenta-se A ou N. O pior caso na figura 23 foi ($N = 1024$, $A = 1024$), para o qual o desvio padrão calculado foi $d = 4,93608 \cdot 10^{-2}$.

A figura 24 mostra novamente o desvio padrão para float-32b, mas dessa vez para N até 131072. O desvio padrão apresenta valores ainda maiores com o aumento de N, chegando para o caso ($A = 10^4$, $N = 131072$) a um valor de $d = 5,61961\dot{1}0^{-1}$. De maneira geral, percebe-se que, fixando-se A, o desvio padrão aumenta com o aumento de N. E ainda, fixando-se N, o desvio padrão aumenta com o aumento de A.

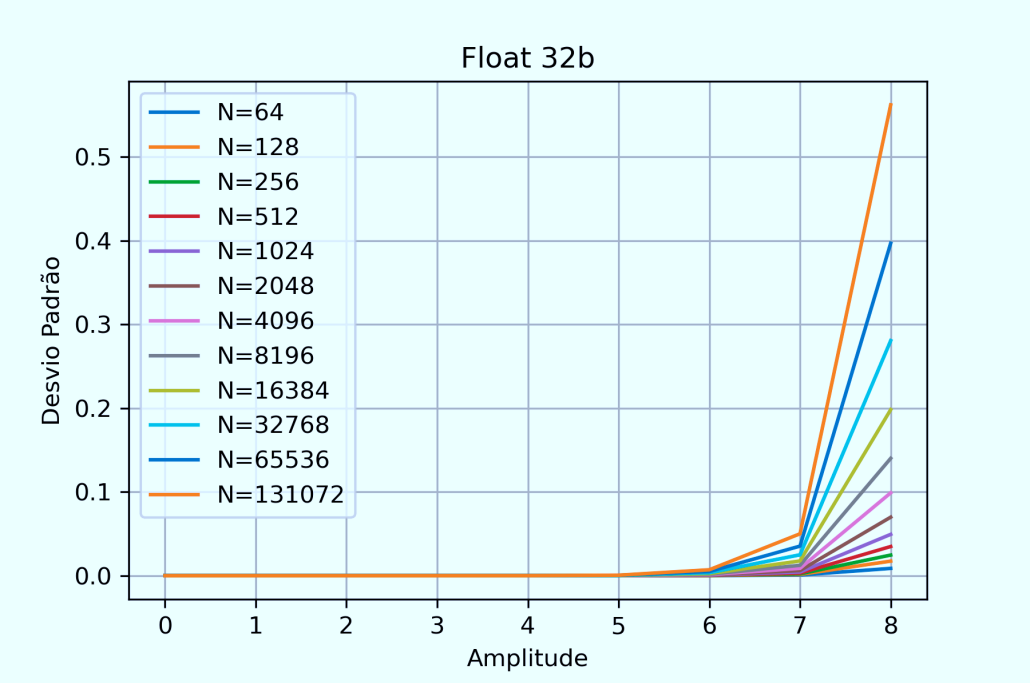
A figura 25 mostra o desvio padrão dos resultados para a FFT em posit-32b. Foi usada a mesma faixa de variação para N e A. Como anteriormente, a diferença nos

Figura 23 – Desvio padrão para resultados em float-32b para N até 1024.



Fonte: Próprio Autor.

Figura 24 – Desvio padrão para resultados em float-32b para N até 131072.

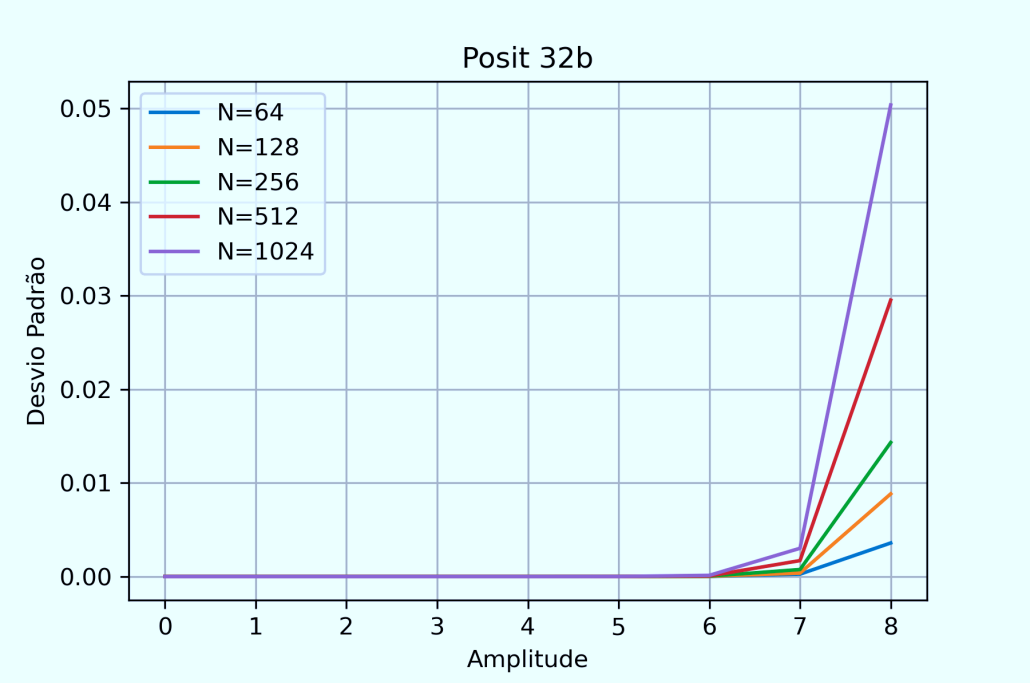


Fonte: Próprio Autor.

resultados para $A \leq 10^2$ (índice 6) não são perceptíveis no gráfico. Para $A > 10^2$ os valores do desvio padrão crescem em uma velocidade muito grande quando aumenta-se A e N. O pior caso para a figura 25 foi $d = 5,03708 \cdot 10^{-2}$ para ($N = 1024$, $A = 1024$). De maneira similar ao float-32b os resultados mostram que, fixando-se A, o desvio padrão aumenta com o aumento de N. E ainda, fixando-se N, o desvio padrão aumenta com o aumento de A.

A figura 26 mostra novamente o desvio padrão para posit-32b para N até 131072. Percebe-se que o desvio padrão apresenta valores ainda maiores com o aumento de N, chegando para o caso ($A = 10^4$, $N = 131072$) a um valor de $d = 2,87297\dot{1}0^{-1}$, valor 5 vezes maior do que o valor para o mesmo caso em float-16b.

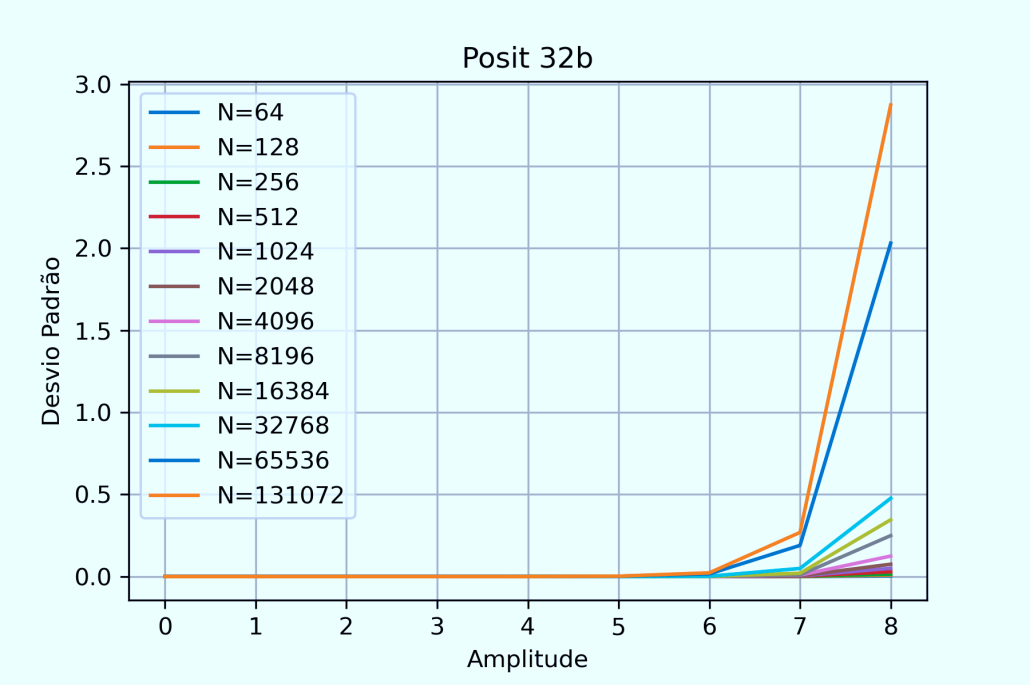
Figura 25 – Desvio padrão para resultados em posit-32b para N até 1024.



Fonte: Próprio Autor.

Os gráficos anteriores não são suficiente para perceber as diferenças dos resultados em posit-32b e float-32b. Por isso, foi construída a Tabela 2 que mostra, para cada caso, qual formato apresentou um menor desvio padrão, e portanto, o melhor resultado. Observa-se pela Tabela 2 que para $N \leq 512$ o formato posit-32b apresentou melhor resultado para qualquer amplitude A. Para $N = \{1024, 2048, 4096, 8192, 16384, 32768\}$ o float-32b se saiu melhor apenas para $A = 10^4$. Já para $\{N = 65536, 131072\}$, o float-32b passa a apresentar melhores resultados para qualquer valor de A.

Figura 26 – Desvio padrão para resultados em posit-32b para N até 131072.



Fonte: Próprio Autor.

Para explicar este comportamento deve-se considerar que o aumento de N e A aumenta a amplitude dos coeficientes de Fourier (considerando os valores dos coeficientes de Fourier antes da divisão por N). Dessa forma, o que os resultados parecem mostrar é que os posits trabalham melhor com números pequenos, pois ao aumentar-se N ou A chega-se a um ponto em os floats passam a apresentar melhores resultados. Essa ideia está de acordo com o fato dos posits terem uma precisão cônica, isto é, sua precisão é maior para valores próximos a 1, enquanto que para valores próximos ao máximo e mínimo a precisão diminui. Quanto mais próximo das extremidades da faixa de representação, maiores são os ”saltos” entre dois números adjacentes. Embora os posits consigam obter uma faixa de valores representável superior para um mesmo número de bits, eles perdem em precisão nas extremidades.

3.1.2 FFT em 16 bits

Usando ainda a definição do sinal $x(n)$ (3.1) aplicou-se a FFT para 16 bits. O mesmo algoritmo usado para 32 bits foi usado para 16 bits, alterando-se apenas o tipo das variáveis. Os valores de N e A que foram considerados para os testes se encontram a seguir.

$$N = \{64, 128, 512, 1024, 2048, 4096, 8192\}$$

Tabela 2 – Comparação posit-32b e float-32b.

N	A	**Melhor Resultado**
64	-	posit-32b
128	-	posit-32b
256	-	posit-32b
512	-	posit-32b
1024	-	posit-32b
	10^4	float-32b
2048	-	posit-32b
	10^4	float-32b
4096	-	posit-32b
	10^4	float-32b
8192	-	posit-32b
	10^4	float-32b
16384	-	posit-32b
	10^3, 10^4	float-32b
32768	-	posit-32b
	10^3, 10^4	float-32b
65536	-	float-32b
131072	-	float-32b

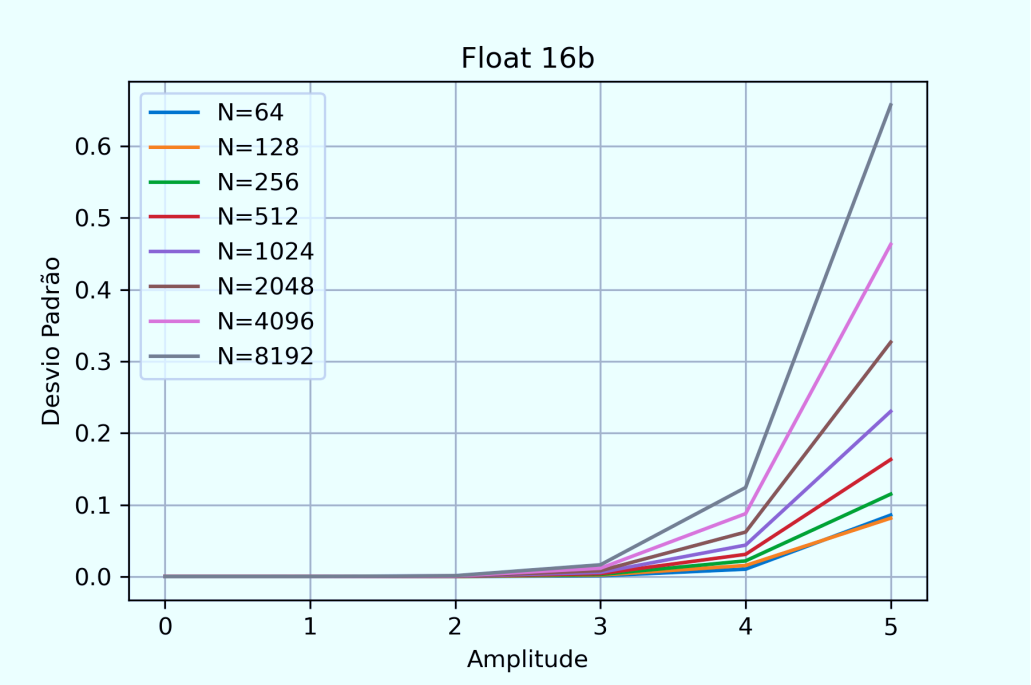
Fonte: Próprio Autor.

$$A = \{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1, 10\}$$

O desvio padrão dos resultados para posit-16b e float-16b encontram-se nas figuras 27 e 28. Como esperado, o desvio padrão dos resultados em 16 bits é maior do que para 32 bits, por causa da redução do número de bits. De modo análogo ao caso anterior, o desvio padrão aumenta com o aumento de A e com o aumento de N. Os piores casos são para $A = 10$. Observa-se uma grande diferença entre os resultados em posit-16b e float-16b, olhando apenas para o caso ($N = 8192$, $A = 10$), o desvio padrão para o posit é 42 vezes maior que o float. As diferenças são bem menores quando utiliza-se valores menores de N e A.

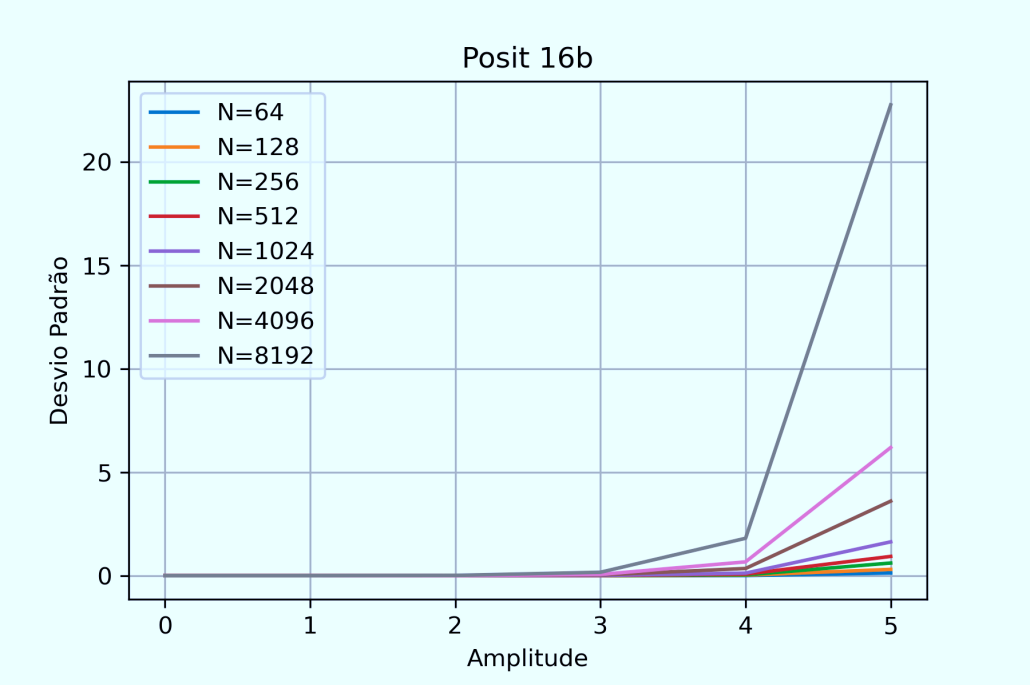
A tabela 3 mostra a comparação entre os formatos float para cada caso testado. O float-16b supera o posit-16b em quase todos casos, com exceção dos casos ($N = 64, A = 10^{-1}$) e ($N = 64, A = 1$). Em 32 bits o float conseguiu superar completamente o posit apenas para $N > 32768$, em 16 bits isso já acontece para $N > 64$. Acredita-se que isso

Figura 27 – Desvio padrão para float 16b para N até 16384.



Fonte: Próprio Autor.

Figura 28 – Desvio padrão para posit 16b para N até 16384.



Fonte: Próprio Autor.

deve acontecer devido a característica já mencionada sobre os posits, o fato deles não possuírem um diferença fixa entre dois números adjacentes. O efeito dessa característica se intensifica quando se diminui o número de bits, portanto, para 16 bits isso se torna mais evidente e afeta mais o resultado. Em 32 bits isso passa a ser problemático para $N > 32768$, já para 16 bits isso acontece muito antes $N > 64$.

Tabela 3 – Comparação posit-16b e float-16b.

N	A	**Melhor Resultado**
64	10^{-4}, 10^{-3}, 10^{-2} 10^{-1}, 1 10	float-16b posit-16b float-16b
128	-	float-16b
256	-	float-16b
512	-	float-16b
1024	-	float-16b
2048	-	float-16b
4096	-	float-16b
8192	-	float-16b

Fonte: Próprio Autor.

4 Conclusões

No presente trabalho foi realizado um estudo da representação em ponto flutuante do padrão IEEE-754 e do novo formato de representação posit. O formato posit foi proposto como alternativa ao uso do float e projetado para uma substituição *drop-in* dos floats.

No intuito de analisar a precisão dos resultados gerados pela aritmética posit foi escrita uma função para cálculo da Transformada Rápida de Fourier com decimação no tempo. O algoritmo foi feito em posit e float para dois tamanhos de palavra: 16 e 32 bits. Os resultados gerados em float-64b foram usados como valor de referência para realizar as comparações. Desse modo, para cada resultado, foi calculado um sinal de diferença em relação ao valor em 64 bits e, por sua vez, o desvio padrão deste sinal para usá-lo como critério de avaliação.

A FFT foi aplicada a diferentes versões do sinal discreto de onda quadrada. Para tanto, fez-se variar o número de amostras N do sinal e a amplitude A a fim de observar a influência destas variações nos resultados. De maneira geral, tanto para o float quanto para o posit observou-se que, fixando-se A, a precisão diminui com o aumento de N. E ainda, fixando-se N, a precisão diminui com o aumento de A. O aumento de A aumenta a magnitude dos operandos usados nos cálculos aritméticos. O aumento de N gera um número maior de multiplicações e somas, que acabam também por aumentar a magnitude dos resultados. O aumento da magnitude gera uma menor precisão tanto em float quanto em posit.

Os resultados mostraram que em 32 bits para $N \leq 32768$ o formato posit-32b apresentou melhor resultado em quase todos os casos testados, com algumas exceções para $A = 10^4$. Para $N > 32768$, o float-32b passa a apresentar melhores resultados para qualquer valor de A. Os resultados indicam, portanto, que existe um valor crítico até onde os posits conseguem gerar resultados superiores aos floats, entretanto, após esse valor, os floats passam a superá-los. No caso do posit-32b esse valor crítico foi $N = 32768$, lembrando que foram considerados apenas valores de N para o qual $log_2 N$ é um número

inteiro. Esse comportamento é explicado pelo fato dos posits possuírem uma precisão cônica, isto é, sua precisão aumenta para números próximos a 1. Em contrapartida, para números próximos aos seus valores máximo e mínimo sua precisão diminui e o espaço entre dois números adjacentes aumenta. Embora os posits consigam obter uma maior faixa de valores representável usando o mesmo número de bits, eles perdem em precisão quando se aproximam das extremidades.

Os resultados da comparação em 16b mostraram que o float-16b superou o posit-16b em quase todos casos testados. Enquanto que em 32 bits o float conseguiu superar completamente o posit para $N > 32768$, em 16 bits isso já aconteceu para $N > 64$. Percebe-se que o efeito da precisão cônica se intensifica quando se diminui o número de bits, portanto, para 16 bits isso tem um efeito maior sobre o resultado. Se em 32 bits isso passa a ser problemático para $N > 32768$, para 16 bits isso acontece muito antes $N > 64$.

Tendo em vista os resultados apresentados, percebe-se que a decisão entre usar posits e floats para uma determinada aplicação deve levar em conta, principalmente, a magnitude dos operandos ou sinais usados. Os posits apresentam melhores resultados para operações com magnitudes próximas a 1 e parece haver um limiar de magnitude, a partir do qual é mais vantajoso usar floats.

Como sugestão de trabalhos futuros, seria interessante realizar a comparação entre posits e floats com uma variedade maior de sinais para verificar se o comportamento permanece o mesmo. Além disso, poderia-se realizar a comparação para funções diferentes da FFT. Durante a realização deste trabalho percebeu-se uma limitação para se trabalhar com os posits usando as bibliotecas existentes até o presente momento. Portanto, ainda é necessário desenvolver esses recursos para que se possa explorar ao máximo o que este novo formato tem a oferecer.

Referências

Cooley, J. W.; Tukey, J. W. An algorithm for the machine calculation of complex fourier series. *Math comput.*, v. 19: 297-301, 1965. Citado na página 23.

Goldberg, D. *What Every Computer Scientist Should Know About Floating-Point Arithmetic*. 1991. Disponível em: <https://www.itu.dk/~sestoft/bachelor/IEEE754_article-.pdf>. Citado na página 21.

Gustafson, J. *The End of Error: Unum Computing*. [S.l.]: Taylor and Francis, 2015. (Chapman and Hall/CRC Computational Science). ISBN 9780198520115. Citado 2 vezes nas páginas 1 e 15.

Kahan, W. How futile are mindless assessments of roundoff in floating-point computation. p. p.37, Nov 2004. Citado na página 1.

Kulish, U. W.; Miranker, W. L. The arithmetic of the digital computer: A new approach. *SIAM Review*, v. 28, n. 1, March 1986. Citado na página 1.

L.Gustafson, J.; I.Yonemoto. *Beating Floating Point at its Own Game: Posit Arithmetic*. Disponível em: <http://www.johngustafson.net/pdfs/BeatingFloatingPoint.pdf>. Citado 6 vezes nas páginas 2, 16, 17, 18, 19 e 20.

NUMPY-POSIT. *Numpy-posit software library for Python language*. acesso: 21/04/2021. Disponível em: <https://github.com/xman/numpy-posit>. Citado na página 23.

Rump, S. M. Algebraic computation, numerical computation and verrified inclusions. In: *Trends in Computer Algebra, ser. Lecture Notes in Computer Science*. [S.l.: s.n.], 1987. v. 296, p. 177–197. Citado na página 1.

TERNOVOY, E. et al. Comparative analysis of floating-point accuracy of ieee 754 and posit standards. In: *2020 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*. [S.l.: s.n.], 2020. p. 1883–186. Citado na página 2.