



**Pós-Graduação em Ciência da Computação**

**“Implementação em FPGA de um módulo multiplicador e acumulador aritmético de alto desempenho para números em ponto flutuante de precisão dupla, padrão IEEE 754”**

**Por**

**Abner Corrêa Barros**

**Dissertação de Mestrado**



Universidade Federal de Pernambuco

[posgraduacao@cin.ufpe.br](mailto:posgraduacao@cin.ufpe.br)

[www.cin.ufpe.br/~posgraduacao](http://www.cin.ufpe.br/~posgraduacao)

**RECIFE, AGOSTO / 2008**





UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

**Abner Corrêa Barros**

**Implementação em FPGA de um módulo multiplicador e acumulador aritmético de alto desempenho para números em ponto flutuante de precisão dupla, padrão IEEE 754**

Este trabalho foi apresentado à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

ORIENTADOR: *Manoel Eusebio de Lima, PhD*

RECIFE, AGOSTO / 2008

**Barros, Abner Corrêa**

**Implementação em FPGA de um módulo multiplicador e acumulador aritmético de alto desempenho para números em ponto flutuante de precisão dupla, padrão IEEE 754 / Abner Corrêa Barros. – Recife: O Autor, 2008.**

**145 p. : il., fig., tab.**

**Dissertação (mestrado) – Universidade Federal de Pernambuco. Cln. Ciência da computação, 2008.**

**Inclui bibliografia.**

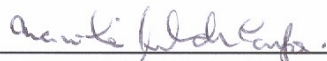
**1. Engenharia da computação. 2. Matemática da computação. I. Título.**

**621.39**

**CDD (22.ed.)**

**MEI2009-012**

Dissertação de Mestrado apresentada por **Abner Corrêa Barros** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título **“Implementação em FPGA de um Módulo Multiplicador e Acumulador Aritmético de Alto Desempenho para Números de Ponto Flutuante de Dupla Precisão, Padrão IEEE 754”**, orientada pelo **Prof. Manoel Eusébio de Lima** e aprovada pela Banca Examinadora formada pelos professores:



Profa. Marcília Andrade Campos  
Centro de Informática / UFPE

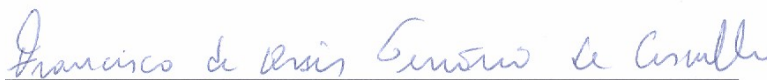


Prof. Pablo Viana da Silva  
Laboratório de Computação Científica e Visualização / UFAL  
Campus Arapiraca



Prof. Manoel Eusébio de Lima  
Centro de Informática / UFPE

Visto e permitida a impressão.  
Recife, 27 de agosto de 2008.



**Prof. FRANCISCO DE ASSIS TENÓRIO DE CARVALHO**

Coordenador da Pós-Graduação em Ciência da Computação do  
Centro de Informática da Universidade Federal de Pernambuco.



**Dedico este trabalho aos meus pais,  
João Alir S. Barros e Maria Vanda C. Barros;  
à minha querida esposa, companheira e  
incentivadora, Rosangela A. Barros;  
aos meus preciosos filhos,  
Natália, Gabriela e Yuri;  
e ao meu orientador e grande incentivador,  
Prof. Manoel Eusebio de Lima.**





## **Agradecimentos**

À Deus, primeiramente, por ter me provido todos os meios necessários à conclusão de mais esta etapa da minha formação acadêmica e profissional.

Aos meus pais, a quem devo tudo o que sou e tudo o que serei um dia.

À minha querida esposa Rosa, por ter sido mais que uma companheira em todos os momentos. Por ter sido a minha maior incentivadora, meu esteio, minha fortaleza nos momentos mais difíceis. Por ter me permitido estar tanto tempo ausente, estando sempre ao meu lado.

Aos meus filhos Natalia, Gabriela e Yuri, amo vocês demais.

Aos meus irmãos Dunga (Eder Magno), Gega (Jesimiel), Lico (Elias) e Polaca (Moreci).

Ao meu orientador e amigo, Prof. Manoel Eusebio de Lima, por tantas palavras de incentivo, por tantos exemplos inspiradores. Por ter sempre uma mão estendida e um coração aberto em todos os momentos.

Ao Professores do Centro de Informática, em especial à Professora Edna Barros, por suas palavras de incentivo e apreço.

Aos companheiros do HPCIn, em especial ao trio fantástico do MAC, JP (João Paulo), Jesus (Ângelo) e Bruno. Sem vocês eu jamais teria conseguido.

Aos amigos do projeto MECAF, Victor, Vivi, Victor Miguel e Vicente, companheiros nas lutas do dia-a-dia.

Aos ilustres professores da banca examinadora, em especial à Professora Marcília Andrade Campos, pelas valiosas informações e correções sugeridas às bases aritméticas aqui apresentadas.

Às minhas três amigas e companheiras em todos estes anos dentro do Centro de Informática da UFPE, LinaT (Aline Timóteo), Merilina (Flávia Marlin) e Amélia (Denise Narciso). Vocês estarão para sempre em meu coração.

Ao CENPES, Petrobras, e à FINEP pelo suporte financeiro e pela oportunidade de participar deste projeto de pesquisa.



## **Epígrafe**

**Ensina-nos a contar os nossos dias, para que alcancemos corações sábios.**  
**Salmos 90:12**



## **Resumo**

Os FPGAs (*Field Programmable Gate Array*) têm sido considerados como uma opção atrativa no desenvolvimento de co-processadores de aplicação específica para sistemas computacionais de alto desempenho. Tradicionalmente, entretanto, estes dispositivos vinham sendo empregados apenas para implementar sistemas que não demandassem um uso intensivo de operações aritméticas envolvendo números em ponto flutuante. Isto acontecia principalmente devido à alta complexidade e ao tamanho dos *cores de hardware* gerados e também devido a escassez de recursos lógicos adequados a este tipo de aplicação nos FPGAs disponíveis à época.

Os recentes avanços nesta tecnologia tem permitido a construção de novas famílias de FPGAs, os quais além de contar com dezenas de milhões de portas lógicas, dispõem também de recursos de hardware mais adequados à aplicações de processamento de alto desempenho, tais como: CPUs, DSPs (*Digital Signal Processor*) e grandes blocos de memória. Estes novos recursos tem permitido que projetistas e engenheiros possam implementar com maior facilidade co-processadores aritméticos mais adequados a aplicações de computação científica.

Neste trabalho, serão apresentados os detalhes de construção de uma unidade aritmética, um multiplicador e acumulador (MAC), implementado em FPGA, o qual segue o padrão IEEE 754 para números em ponto flutuante de precisão dupla. Esta unidade foi desenvolvida como parte de um co-processador aritmético de aplicação específica, dedicado a multiplicação de matrizes densas, para uso em plataformas computacionais de alto desempenho.

O padrão IEEE 754 é descrito em detalhes, bem como a arquitetura interna da unidade aritmética implementada. Serão apresentadas também as metodologia de desenvolvimento e teste empregadas na construção deste dispositivo.

Palavras chaves: FPGA, Multiplicador e acumulador, Aritmética de Ponto Flutuante, Padrão IEEE 754.



## **Abstract**

The FPGAs (Field Programmable Gate Array) have been considered as an attractive option in the development of specific application co-processors for high performance computer systems. Traditionally, however, these devices were being used only to implement systems that do not demand an intensive use of arithmetic operations with floating point. This happened, mainly because of the high complexity and size of the hardware cores generated and also because the scarcity of logic resources suitable for this type of application in previous families of FPGAs. Recent advances in this technology have allowed the development of the new family of FPGAs, with tens of millions of logic gates, which have more specific hardware resources such as CPUs, DSPs and blocks of memory. These new features have helped the designers and engineers to implement, more easily, new specific application co-processors, more suitable for applications in scientific computing.

In this work, an arithmetic unit, a multiplier and accumulator (MAC), implemented in FPGA, according to IEEE standard 754 for double-precision numbers is presented. This unit was developed as part of an application specific co-processor, dedicated to multiplication of dense matrices, for use in high performance computing platforms. The IEEE 754 standard and the internal architecture of arithmetic unit are described in detail. Will be also presented the methodology of development and testing, which includes an application in C + +, especially developed for testing and validation of the device.

**Keywords:** FPGA, multiplier and accumulator, the Floating Point Arithmetic, IEEE Standard 754.





# SUMÁRIO

<b>1</b>	<b>INTRODUÇÃO</b>	<b>15</b>
<b>2</b>	<b>ARITMÉTICA DE PONTO FLUTUANTE</b>	<b>19</b>
2.1	Um pouco de História	19
2.2	Padrões de Notação	24
2.2.1	Notação de Inteiros	25
2.2.2	Notação de Ponto Fixo	26
2.2.3	Notação de Ponto Flutuante	27
2.3	O Padrão IEEE 754	33
2.3.1	Formato de representação dos dados	34
2.3.2	Representação de Valores Especiais	37
2.3.3	Arredondamento	43
2.3.4	Modos de Arredondamento	45
2.3.5	Algoritmos de Arredondamento	49
2.3.5.1	Algoritmo do modo de arredondamento em direção ao Zero	52
2.3.5.2	Algoritmo do modo de arredondamento em direção ao + e ao - Infinito	53
2.3.5.3	Algoritmo do modo de arredondamento em direção ao mais próximo ou par	54
2.3.6	Normalização	56
2.3.6.1	Algoritmo de Normalização	58
2.3.7	Operações Aritméticas	59

2.3.7.1	Operações de Soma e Subtração.....	61
2.3.7.2	Operação de Multiplicação.....	69
<b>3</b>	<b>TRABALHOS RELACIONADOS.....</b>	<b>75</b>
3.1	Conclusões.....	86
<b>4</b>	<b>UNIDADE DE MULTIPLICAÇÃO E ACUMULAÇÃO.....</b>	<b>89</b>
4.1	Requisitos funcionais e não funcionais.....	90
4.2	Metodologia de desenvolvimento.....	92
4.3	Arquitetura Interna.....	102
4.3.1	Unidade de Aquisição.....	104
4.3.2	Unidade de Multiplicação.....	108
4.3.2.1	AddExp.....	109
4.3.2.1	MultSig.....	110
4.3.3	Unidade de Soma.....	116
4.3.3.1	ConvCmpl2 .....	118
4.3.3.2	ShiftRight .....	118
4.3.3.3	CmpExp.....	118
4.3.3.4	AddSig .....	119
4.3.3.5	ConvMagSinal.....	119
4.3.4	Unidade de Normalização e Arredondamento.....	120
4.3.4.1	NormSig.....	121
4.3.4.2	ArrdSig.....	122
4.3.4.3	AjustaExp.....	123
4.3.5	Unidade de Controle de Exceção.....	124

4.3.6	Unidade de Formatação dos Resultados.....	126
4.4	Arquitetura Final e Pipe-line do sistema.....	128
4.4.1	Descrição dos Estágios do Pipe-line.....	128
5	<b>ESTUDO DE CASO – IMPLEMENTAÇÃO NA PLACA PCI.....</b>	<b>133</b>
6	<b>CONCLUSÕES E TRABALHOS FUTUROS.....</b>	<b>139</b>
7	<b>REFERÊNCIAS BIBLIOGRÁFICAS.....</b>	<b>141</b>



# ÍNDICE DE ILUSTRAÇÕES

Figura 1.1: Projeção da demanda computacional em computação científica para os próximos anos [2].....	16
Figura 2.1: Z1, o primeiro computador digital.....	20
Figura 2.2: Distribuição dos valores representáveis em uma notação de ponto flutuante binária baseada na configuração do exemplo dado.....	31
Figura 2.3: Distribuição dos valores representáveis por números normalizados em ponto flutuante indicando a faixa de valores representado pelo zero.....	39
Figura 2.4: Distribuição dos valores representáveis por números normalizados em ponto flutuante indicando a faixa de valores representados pelo números denormalizados.....	40
Figura 2.5: Exemplos de representação dos valores especiais do padrão IEEE 754. ....	42
Figura 2.6: Diferença na representação entre um significando com 3 e um com 5 bits ....	44
Figura 2.7: Comparação entre valores representáveis com significandos com 5 e 3 bits.....	45
Figura 2.8: Direção do arredondamento em direção ao Zero.....	46
Figura 2.9: Direção do arredondamento em direção ao $+\infty$ .....	46
Figura 2.10: Direção do arredondamento em direção ao $-\infty$ .....	47
Figura 2.11: Direção do arredondamento para o mais próximo ou par.....	48
Figura 2.12: Exemplos de Arredondamento em direção ao $+\infty$ e ao $-\infty$ .....	48
Figura 2.13: Exemplo de Arredondamento para o mais próximo e em direção ao zero ....	48

Figura 2.14: Esquema para obtenção do LSB bit, do Guard bit e do Sticky bit.....	49
Figura 2.15: Nova organização dos bits para simplificar a análise do processo de arredondamento.....	50
Figura 2.16: Geração do novo significando a partir dos bits extraídos do significando original e do Round bit obtido durante o processo de arredondamento.....	52
Figura 2.17: Localização do valor relativo do número a ser arredondado a partir do Guard Bit e do Sticky Bit.....	54
Figura 2.18: Deslocamento do Significando.....	62
Figura 2.19: Representação esquemática da conversão de Op2 para o formato de Ponto Flutuante proposto.....	64
Figura 2.20: Representação esquemática da conversão de Op1 para o formato de Ponto Flutuante proposto.....	64
Figura 2.21: Representação esquemática da conversão de Op3 para o formato de Ponto Flutuante proposto.....	64
Figura 2.22: Representação esquemática da conversão de Op4 para o formato de Ponto Flutuante proposto.....	64
Figura 2.23: Diagrama esquemático da operação de soma entre Op1 e Op2.....	65
Figura 2.24: Representação do resultado da operação entre Op1 e Op2 no formato de Ponto Flutuante proposto.....	65
Figura 2.25: Representação esquemática da operação de adição entre Op3 e Op4	66
Figura 2.26: Operação de Normalização sobre resultado da adição de Op3 e Op4.	67
Figura 2.27: Distribuição dos bits durante o processo de arredondamento.....	67
Figura 2.28: Resultado do processo de arredondamento.....	68
Figura 2.29: Formatação do resultado obtido.....	68
Figura 2.30: Fluxo de execução das operações de soma e subtração.....	69

Figura 2.31: Distribuição dos bits como resultado de uma operação de multiplicação entre dois inteiros de tamanho $n$ .....	70
Figura 2.32: Fluxo de execução da operação de multiplicação.....	71
Figura 2.33: Representação esquemática da conversão de Op1 para o formato de Ponto Flutuante proposto.....	71
Figura 2.34: Representação esquemática da conversão de Op2 para o formato de Ponto Flutuante proposto.....	72
Figura 2.35: Representação esquemática da conversão do resultado para o formato de Ponto Flutuante proposto. ....	73
Figura 3.1: Relação do número de estágios do pipe-line pela potência consumida pelos módulos somadores e multiplicadores.....	76
Figura 3.2: Relação do número de estágios do pipe-line pela frequência de trabalho dos módulos somadores e multiplicadores.....	76
Figura 3.3: Diagrama interno funcional das operações de raiz quadrada e adição/subtração, respectivamente conforme propostos em [27].....	77
Figura 3.4: Diagrama interno funcional das operações de multiplicação e divisão, respectivamente conforme propostos em [27].....	78
Figura 3.5: Relação entre número de estágios do pipe-line pela frequência de trabalho e área ocupada para o módulo de multiplicação [27].....	78
Figura 3.6: Relação entre número de estágios do pipe-line pela frequência de trabalho e área ocupada para o módulo de soma/subtração [27].....	78
Figura 3.7: Relação entre número de estágios do pipe-line pela frequência de trabalho e área ocupada para o módulo de raiz quadrada [27].....	78
Figura 3.8: Relação entre número de estágios do pipe-line pela frequência de trabalho e área ocupada para o módulo de divisão [27].....	79
Figura 3.9: Esquema de controle proposto em [28] na arquitetura de multiplicação de matrizes.....	81

Figura 3.10: Arquitetura de controle e distribuição dos dados para implementações com e sem DMA conforme proposto em [28].....	82
Figura 3.11: Organização interna dos PE's e a representação funcional dos Multiplicadores e Acumuladores conforme proposto em [28].....	82
Figura 3.12: Pipe-line interno dos Multiplicadores e Acumuladores implementados em [28].....	83
Figura 4.1: Diagrama de tempo dos sinais do MAC.....	91
Figura 4.2: Diagrama esquemático funcional do MAC.....	91
Figura 4.3: Fluxo de projeto do modelo de referência.....	92
Figura 4.4: Verificação e teste do modelo de referência.....	93
Figura 4.5: Geração dos arquivos de referência.....	95
Figura 4.6: Ambiente de desenvolvimento de software utilizado - Eclipse + Mingw. .	96
Figura 4.7: Fluxo de projeto dos módulos de hardware.....	96
Figura 4.8: Processo de verificação e teste dos módulos de hardware.....	97
Figura 4.9: Ferramenta de desenvolvimento de hardware - Modelsim XE 6.2.....	98
Figura 4.10: Ferramenta de desenvolvimento de hardware - Xilinx ISE 9.2i.....	100
Figura 4.11: Visão geral do multiplicador e acumulador implementado, identificando as suas portas de entrada e de saída.....	101
Figura 4.12: Visão dos módulos internos do multiplicador a acumulador.....	101
Figura 4.13: Exemplo de operação integrada das operações de multiplicação e soma sem a normalização do resultado intermediário.....	102
Figura 4.14: Diagrama de distribuição dos módulos internos do MAC.....	104
Figura 4.15: Diagrama interno da Unidade de Aquisição.....	104
Figura 4.16: Diagrama esquemático interno do blocos funcionais que formam a Unidade de Aquisição .....	105



Figura 4.17: Remapeamento dos valores do expoente de 11 para 12 bits.....	106
Figura 4.18: Diagrama do módulo que estende o expoente.....	107
Figura 4.19: Diagrama interno da Unidade de Multiplicação.....	108
Figura 4.20: Diagrama esquemático do bloco AddExp.....	110
Figura 4.21: Operação de multiplicação entre os números X e Y em representação binária.....	111
Figura 4.22: Algoritmo Dadda aplicado a multiplicação dos significandos.....	114
Figura 4.23: Esquema funcional da multiplicação dos operandos D e H, aqui representados pelo operando B.....	115
Figura 4.24: Arvore de somas parciais.....	116
Figura 4.25: Diagrama esquemático funcional da Unidade de Soma.....	117
Figura 4.26: Diagrama funcional interno do bloco CmpExp.....	120
Figura 4.27: Diagrama interno da Unidade de Normalização e Arredondamento....	120
Figura 4.28: Diagrama interno do bloco funcional NormSig.....	121
Figura 4.29: Diagrama interno do ArrdSig .....	122
Figura 4.30: Esquema para conversão do expoente de 12 para 11 bits.....	124
Figura 4.31: Diagrama interno da Unidade de Controle de Exceções .....	125
Figura 4.32: Diagrama interno da Unidade de Formatação dos Resultados.....	127
Figura 4.33: Arquitetura final e Pipe-line do Sistema.....	132
Figura 5.1: Placa de prototipação Virtex-II Development Board da AVNET.....	133
Figura 5.2: Características dos componentes da família Xilinx Virtex II [35].....	134
Figura 5.3: Diagrama sucinto da arquitetura interna da Virtex II [35].....	135
Figura 5.4: Resultado da síntese conjunta.....	135
Figura 5.5: Camadas de abstração de software.....	136



# 1 INTRODUÇÃO

O emprego de co-processadores aritméticos de aplicações específicas a fim de melhorar de desempenho de sistemas computacionais não é algo novo em computação de alto desempenho. Desde o início de sua história esta classe de computadores têm contado com diversos dispositivos de hardware que visam melhorar seu desempenho em aplicações que façam uso intensivo de operações aritméticas.

Um dos principais dispositivos desenvolvido nesta área foi o co-processador aritmético i8087, apresentado ao mercado pela Intel em 1980 [1], o qual era oferecido como uma solução completa, de alto desempenho e independente de plataforma. Este componente implementava as quatro operações aritméticas básicas e ainda as operações de raiz quadrada, tangente, arco tangente, exponencial e logaritmos para números em notação de ponto flutuante. Atualmente, soluções semelhantes encontram-se integradas diretamente nas arquiteturas de praticamente todos os processadores através das unidades de ponto flutuantes ou FPU (*Floating Point Unit*).

Entretanto, ainda que adequada à grande maioria das aplicações, este tipo de arquitetura não tem sido suficiente para suprir a necessidade de processamento de uma parte significativa das aplicações atuais, devido não apenas ao aumento da complexidade das operações mas também ao aumento no volume de dados envolvidos [2][3].

Aplicações em áreas como genômica, proteômica, sísmica, realidade virtual e simuladores do clima, apenas para citar algumas, têm demandado recursos computacionais que já ultrapassam os Tera-flops, ou  $10^{12}$  operações de ponto flutuante por segundo [2][3]. A Figura 1.1 traz uma projeção da demanda computacional esperada para algumas destas aplicações para os próximos anos [2].

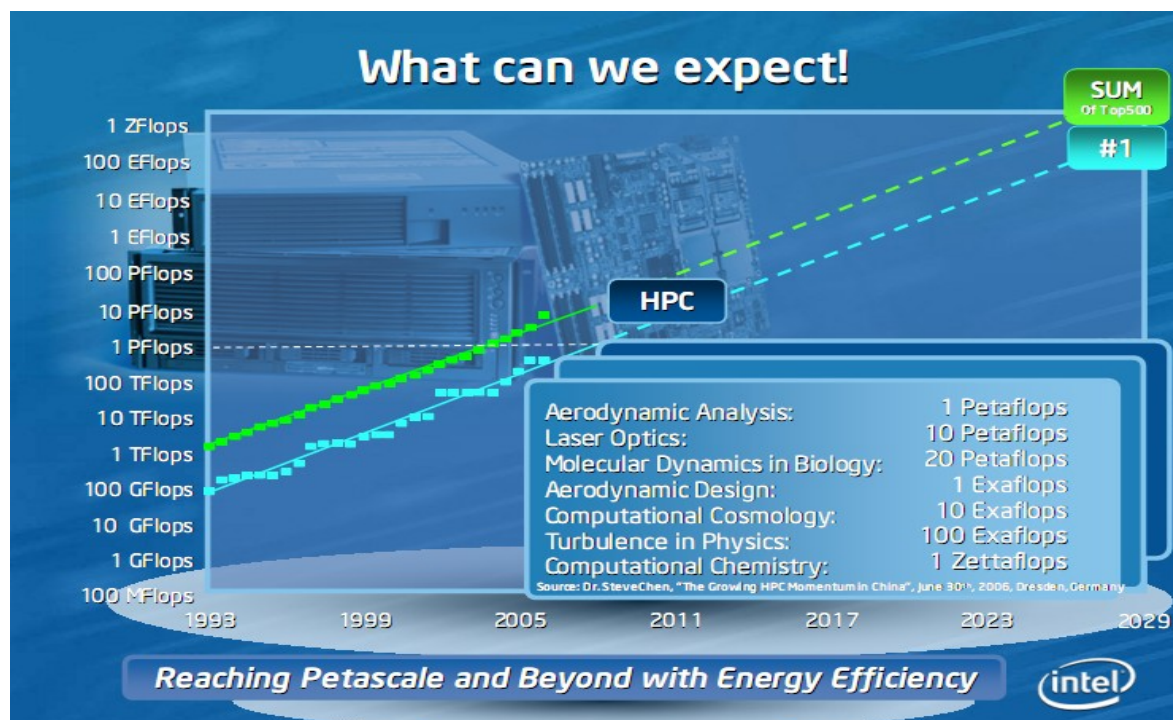


Figura 1.1: Projeção da demanda computacional em computação científica para os próximos anos [2]

Neste cenário, o uso de dispositivos lógicos programáveis, como os FPGAs (*Field Programmable Gate Arrays*), e os CPLDs (*Complex Programmable Logic Devices*), como elementos aceleradores de processamento, tem despertado grande interesse, tanto da comunidade científica quanto dos fabricantes de computadores [4][5][6][7]. Este interesse se deve principalmente a possibilidade do emprego destes dispositivos para desenvolvimento de co-processadores aritméticos de aplicação específica que melhor se adaptem às características de cada problema a ser tratado [8].

O emprego destes dispositivos têm aberto caminho para o desenvolvimento de um novo paradigma no tratamento da computação científica, passando do conceito tradicional de implementação de algoritmos em software, para a implementação em hardware. Entretanto, este novo paradigma tem exigido dos profissionais envolvidos um conhecimento profundo, tanto do algoritmo a ser implementado quanto da arquitetura do hardware a ser projetado. Neste contexto, novas linguagens de programação e especificação têm surgido, na tentativa de

melhor adequar o mundo do hardware ao tradicional mundo de programação em software[9][10].

Embora com um grande poder de representação, *devido aos milhões de portas lógicas que dispõe na forma de elementos reconfiguráveis, os FPGAs não podem ser empregados como solução para todos os tipos de algoritmos. Por operar a frequências de trabalho relativamente baixas [11][12], se comparadas a frequência de trabalho dos processadores, esta tecnologia não tem apresentado bom resultado em algumas classes de algoritmos*, principalmente aqueles com forte características de controle e fluxo seqüencial. Por outro lado, estudos tem demonstrado que classes de algoritmos que apresentem um alto grau de paralelismo intrínseco e nos quais estejam envolvidos grandes volumes de dados, apresentam não apenas um ganho substancial em desempenho mas também em consumo de energia [13].

Em [13], é demonstrando que os excelentes resultados apresentados por algoritmos quando implementados em FPGA, se devem principalmente à natureza do problema a ser tratado e à arquitetura de hardware disponível. Entretanto, outros fatores ainda podem ser destacados:

- Paralelismo intrínseco existente entre as operações internas do algoritmo implementado;
- Paralelismo de acesso aos dados;
- Reuso dos dados;
- Possibilidade de uma maior integração e sincronismo entre as diversas fases do algoritmo implementado, o que permite a redução de operações redundantes.

Principalmente em se tratando de operações com números em ponto flutuante, o desenvolvimento de operações integradas, diretamente em hardware, tem permitido eliminar diversas fases de pré e pós processamento, com significativa redução na complexidade dos algoritmos implementados.

Em [13] o autor conclui que, apesar da aparente desvantagem apresentada pela baixa frequência de trabalho dos *FPGAs*, a relação operações/ciclos de trabalho para os algoritmos implementados diretamente em hardware é extremamente favorável.

O trabalho objeto desta dissertação trata da implementação de um co-processador aritmético de aplicação específica, dedicado a operações integradas de multiplicação e soma de números em ponto flutuante de precisão dupla, o qual atende ao padrão IEEE 754. Este co-processador foi prototipado em um FPGA Xilinx XCV6000, controlado através de um computador tipo PC através de um barramento PCI. O núcleo de hardware foi totalmente testado e validado, e um estudo de caso foi implementado para sua demonstração.

Esta dissertação está dividida da seguinte forma:

No Capítulo 2 será apresentada uma breve revisão sobre representação e operação de números em ponto flutuante de acordo com o padrão IEEE 754.

No Capítulo 3 será apresentado um resumo sucinto dos principais trabalhos relacionados e sua contribuição e/ou relação com o este trabalho.

No Capítulo 4 serão apresentados os detalhes de implementação do co-processador.

Um estudo de caso com o co-processador implementado é apresentado no Capítulo 5.

Finalmente, o Capítulo 6 apresenta as conclusões do projeto e sugestões para trabalhos futuros.

## 2 ARITMÉTICA DE PONTO FLUTUANTE

“Para muitas pessoas a aritmética de ponto flutuante ainda é considerada algo esotérico.”[14].

Esta afirmação não nos surpreenderia se não dissesse respeito à comunidade científica em geral e, mais especificamente, aos profissionais da área da computação. Em parte isto se deve ao grande poder de representação das linguagens de programação, as quais dispõem de bibliotecas com praticamente todas as funções e operações que cientistas e engenheiros normalmente necessitam em seu dia a dia, de forma que a maioria destes simplesmente abstraem os detalhes de implementação dos programas e linguagens que utilizam. Por este motivo, o tipo ponto flutuante passa a ser tratado apenas como mais uma entidade abstrata, capaz de, como por um passe de mágica, representar em apenas 32 ou 64 bits todo o conjunto dos números reais.

Diferentemente do conjunto dos números inteiros, o qual encontra representação direta no hardware dos computadores através das operações com números em base binária, o conjunto dos números reais deve ser representado de maneira indireta, através dos números em ponto flutuante. Isto faz com que os seus detalhes de manipulação e representação fiquem encobertos dos programadores.

### 2.1 UM POUCO DE HISTÓRIA

Desde o primeiro computador digital, o Z1 (Figura 2.1), implementado por Konrad Zuse em 1938 [15], verificou-se a necessidade de se buscar uma representação conveniente para os números reais nos sistemas computacionais .

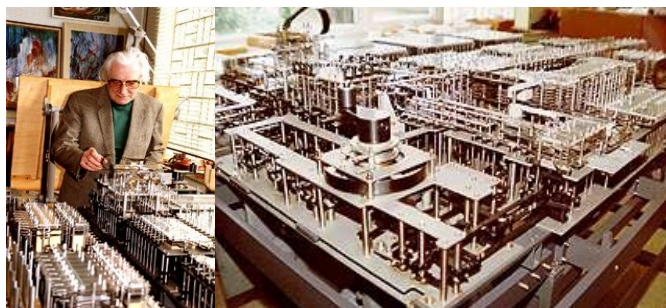


Figura 2.1: Z1, o primeiro computador digital

O próprio Konrad Zuse propôs e implementou uma representação, a qual ele deu o nome de “*semi-logaritmos*”, muito próxima da representação atualmente adotada para números em ponto flutuante de precisão simples [15].

No início da sua história, os computadores eram empregados basicamente para a execução de projetos científicos e de engenharia, os quais se utilizam largamente de cálculos com números reais.

Entre as décadas de 60 e 70, com o avanço da micro-eletrônica, e conseqüentemente dos computadores, surgiram diversas propostas e soluções tanto em hardware quanto em software para a representação dos números reais. Mas, a falta de uma padronização, tanto na representação do número em si, como em outros detalhes de implementação, tornavam a tarefa de operar com os números reais algo extremamente penoso para os programadores. Em alguns casos era necessário lançar mão de recursos no mínimo mirabolantes, para não dizer bizarros, para garantir um resultado confiável em uma determinada operação aritmética. Atribuições do tipo  $x = (x+x) - x$ , ou multiplicar todo e qualquer número por 1.0 antes de qualquer outra operação, eram os artifícios utilizados a fim de garantir a representação adequada dos números reais [16].

Por construção, os computadores digitais trabalham apenas com números inteiros em representação binária, os quais, apesar do seu grande poder de representação, não são suficientes para representar diretamente o conjunto dos números reais.



De um modo geral, as soluções mais promissoras convergiam em torno de uma representação semelhante à notação científica já empregada na aritmética clássica, tal como a adotada por Zuse no Z1. As soluções diferiam porém no número de bits empregados, na base numérica adotada e em outros detalhes das operações aritméticas em si e do processo de arredondamento dos resultados obtidos.

O modo de representação dos números reais em sistemas computacionais começou a mudar por volta de 1976. Ano em que a Intel iniciou o projeto de um co-processador aritmético para os processadores das famílias i8088/6 e i432. Este co-processador deveria representar uma solução única, em hardware, para os problemas da representação e operação dos números em ponto flutuante para estas duas famílias de processadores.

Esta decisão, foi tomada pelo então pesquisador da Intel, Dr. Palmer, a partir de uma exposição feita pelo Dr. William Kahan, dez anos antes, quando este era professor visitante na Universidade de Stanford. Na ocasião, Dr. William Kahan discursou sobre as anomalias decorrentes das soluções existentes na época, as quais além de causar transtornos para os programadores, inflacionavam os custos dos programas desenvolvidos para às plataformas existentes. O próprio Dr. William Kahan já havia, ele mesmo, promovido melhorias significativas nas unidades de cálculo das conhecidas calculadoras Hewlett-Packard [16].

O Dr. William Kahan, foi então convidado a participar como consultor deste novo projeto e, de imediato, sugere a adoção do padrão desenvolvido e implementado nos computadores da Digital Equipment Corporation(DEC), segundo ele a melhor solução até então apresentada. A Intel rejeita a sua proposta alegando que não desejava apenas uma boa solução, senão a melhor solução para o problema de um mercado que, segundo esperavam, se tornaria muito, muito grande [16].

O modelo então proposto e implementado pela equipe dos Drs. John Palmer e William Kahan, veio a se tornar o conhecidíssimo co-processador aritmético i8087.

Com uma lógica simples e eficiente, implementada com apenas alguns milhares de transistores, o i8087 atendia plenamente a todos os requisitos de confiabilidade e portabilidade exigidos pelos programas da época. Estavam assim lançadas as bases do que viria a se tornar o padrão adotado em praticamente todas as soluções, tanto de hardware, como de software, para aritmética de ponto flutuante.

Na mesma época, em meados da década de 70, outros grandes fabricantes de processadores começaram também a desenvolver as suas próprias soluções em hardware, entre eles a Zilog, a National e a Motorola. Além destes, existiam também as soluções já adotadas por grandes fabricantes de supercomputadores tais como a CRAY, a IBM e a Digital Equipment Corporation. Entretanto, cada um com o seu próprio padrão de representação e detalhes de implementação.

Foi em meio a esta babel de soluções sobre a representação dos reais tanto em hardware quanto em software, cada uma com suas vantagens e desvantagens, que o IEEE interveio, através do Dr. Robert Stewart, convocando a todos para a obtenção de um consenso sobre o assunto. Nasce assim, em 1977 a proposta para a criação do padrão IEEE 754, o qual seria concluído e lançado quase 10 anos depois, em 1985. Posteriormente, em 1987, este padrão seria ainda ampliado e revisado dando origem ao padrão IEEE 854.

Mesmo sob protesto dos demais fabricantes participantes, esta norma seguiu quase que na íntegra o projeto proposto e implementado pela Intel no i8087.

Em 2001 foi apresentada uma proposta de revisão para o padrão IEEE 754, denominada DRAFT Standard for Floating-Point Arithmetic P754. Alguns documentos referenciam a este padrão como IEEE 754r, em virtude do nome do grupo de trabalho que o elaborou dentro do IEEE.[17][18]

De um modo geral, o padrão IEEE 754 trata da implementação da aritmética de ponto flutuante em base binária, ou radix 2, enquanto o padrão IEEE 854 trata da implementação da aritmética de ponto flutuante para uma base, ou radix, qualquer, sem limitação no tamanho em bits da representação a ser adotada.

Desta forma, pode-se afirmar que o padrão IEEE 854 realmente estende o IEEE 754, o qual está integralmente contemplada em suas recomendações.

Por outro lado, o IEEE P754 propõe a adoção direta da base decimal, no lugar da base binária até então adotada, o que tornaria a representação muito mais inteligível ao ser humano e mais adequada ao uso nos sistemas financeiros, uma vez que estes invariavelmente adotam o sistema decimal em suas operações [19] [20][17].

Atualmente a aritmética de ponto flutuante é algo ubíquo em sistemas computacionais. Praticamente todas as linguagens possuem uma biblioteca que dá suporte as suas operações. Desde um simples computador pessoal, ao mais possante supercomputador, todos possuem algum tipo de hardware acelerador para melhorar o seu desempenho em operações com ponto flutuante [14].

Mesmo após o lançamento das revisões posteriores, o padrão IEEE para aritmética ponto flutuante, ou simplesmente **IEEE-754**, ainda é o padrão de notação mais utilizado pelos fabricantes de hardware e software de computadores. Suas normas são empregadas tanto na construção de compiladores como na implementação de unidades de processamento aritmético, também conhecidas como FPUs (*Floating Point Unit*), as quais são implementadas diretamente em hardware.

Neste trabalho será dado ênfase apenas ao estudo e a implementação seguindo o padrão IEEE 754 [19].

## 2.2 PADRÕES DE NOTAÇÃO

O problema de definir padrões de representação passa obrigatoriamente pela natureza da informação a ser representada e pelo poder de representação do sistema onde esta deva ser representada.

Este problema se torna ainda mais grave em se tratando da representação de conjuntos infinitos não numeráveis, como no caso de números reais, uma vez que não só todos os elementos do referido conjunto devem ser representáveis, mas também todas as operações e relações envolvendo estes elementos devem ser preservadas.

No caso específico da representação dos números reais em sistemas computacionais, a solução adotada foi proposta na forma de uma expansão numérica denominada expansão “*b-ádica*” [21].

Pela expansão “*b-ádica*” todo e qualquer numero real,  $x$ , pode ser **univocamente** representado da seguinte forma:

$$x = \delta d_n d_{n-1} \cdots d_1 d_0 . d_{-1} d_{-2} \cdots \quad (1)$$

$$x = \delta \sum_{k=n}^{-\infty} d_k b^k \quad (2)$$

onde,  $\delta \in \{+, -\}$ , ( $\delta = \text{Sinal Algébrico}$ )

$b \in \mathbb{N}$ ,  $b > 1$ , ( $b = \text{Base numérica}$ )

$0 \leq d_k \leq b - 1$ ,  $k = n(-1) - \infty$ , ( $d = \text{Dígitos}$ )

$d_k \leq b - 2$ , para infinitamente muitos  $k$

Ou seja, através desta expansão, dada uma base numérica qualquer, todo e qualquer número real pode ser expresso como um somatório infinito de termos ponderados da base adotada. Em sistemas computacionais, devido a restrições de ordem tecnológica, adota-se uma representação finita da expansão *b-ádica* para representar todos os números reais, o que inclui os números naturais, inteiros, racionais e irracionais.

### 2.2.1 Notação de Inteiros

Este é o padrão de notação numérica normalmente suportada por todos os sistemas computacionais. Ele apresenta a vantagem de ter suporte direto no hardware de todos os processadores, o que torna a sua utilização extremamente barata tanto do ponto de vista do tamanho do código gerado, quanto pelo número de ciclos de relógio necessários à execução das suas operações aritméticas básicas.

Sua definição depende exclusivamente da quantidade de dígitos utilizado para representar o número. Nesta notação, um número inteiro,  $z$  qualquer, pode ser representado em uma base numérica  $\beta$  na forma de uma sequência de  $n$  dígitos ordenados, mais uma representação de sinal, através da expressão abaixo:

$$z = \pm \sum_{i=1}^n d_i * \beta^{i-1}, \quad (3)$$

Ou seja, o valor numérico de  $z$  na base  $\beta$  é obtido pelo somatório do valor atribuído aos dígitos, multiplicados pela base numérica adotada ponderada pela posição do dígito, conforme se pode ver a seguir.

$$z = \pm d_n * \beta^{n-1} + d_{n-1} * \beta^{n-2} + \dots + d_2 * \beta^1 + d_1 * \beta^0 \quad (4)$$

Esta notação, apesar da sua simplicidade, tem um razoável poder de representação, pois os números são formados por arranjo, com repetição dos

valores numéricos contidos na base adotada. Desta forma, em uma representação com  $n$  dígitos mais uma indicação de sinal, é possível formar  $2 \times \beta^n$  números ou valores distintos. Em especial na base binária, onde é comum adotar-se o dígito mais significativo exclusivamente para a indicação do sinal do número representado, considerando uma representação com  $n$  dígitos, teremos:

$$\pm(d_{n-1} * 2^{n-1} + d_{n-2} * 2^{n-2} + \dots + d_2 * 2^1 + d_1 * 2^0) = 2^n \text{ números representáveis} \quad (5)$$

### 2.2.2 Notação de Ponto Fixo

Na notação de ponto fixo busca-se uma maneira de representar não apenas a parte inteira dos números mas também a sua parte fracionária. Para tanto, divide-se os  $n$  dígitos da representação em dois grupos, separados pelo “ponto separador”, também conhecido como “*radix point*”. Os dígitos do grupo à esquerda do ponto separador são utilizados para representar a parte inteira do número, enquanto que, os dígitos da direita são utilizados para representar a sua parte fracionária.

Desta forma, a definição de uma notação de ponto fixo depende tanto da quantidade de dígitos utilizados na representação, quanto de como estes dígitos estão distribuídos para representação da parte inteira e da parte fracionária.

De maneira similar à notação de inteiros, nesta notação, o valor numérico representado também é obtido a partir do somatório do valor de cada um dos seus dígitos ponderados pela base, de acordo com a posição relativa que o dígito ocupa na *string* numérica representada. A única diferença neste caso é que a posição relativa do dígito em relação ao ponto separador também é levado em conta.

Desta forma, considerando a representação de um número  $z$  qualquer, em uma notação de ponto fixo com  $n$  dígitos, sendo  $k$  a quantidade de dígitos reservados para a representação da parte fracionária, ou seja, sendo  $k$  a posição do ponto separador, teremos o valor de  $z$  definido conforme indicado na expressão a seguir.

$$z = \pm \sum_{i=1}^n di * \beta^{i-1-k}, \quad (6)$$

Nesta notação, o acréscimo de expressividade obtido com a inclusão da parte fracionária vem com o prejuízo da quantidade de dígitos disponíveis para a representação da parte inteira e, conseqüentemente, da quantidade de números inteiros representáveis, que passam dos  $\beta^n$  possíveis com a notação de números inteiros, para  $\beta^{n-k}$ . Entretanto, como a cada valor inteiro representável foram acrescentados  $\beta^k$  partições decimais, temos a expressão:

$$\beta^{n-k} * \beta^k = \beta^n \quad (7)$$

Ou seja, o total de números representáveis ainda continua  $\beta^n$ .

### 2.2.3 Notação de Ponto Flutuante

A notação de ponto flutuante estende a notação de ponto fixo com o acréscimo de mais um grupo de dígitos denominado expoente. Com o acréscimo deste grupo de dígitos o número passa a ser representado pelo valor contido no conjunto de dígitos da parte inteira e da parte fracionária, os quais recebem juntos o nome de significando, multiplicado pela base elevada ao valor do campo expoente, conforme demonstrado pela expressão a seguir.

$$\pm \text{Significando} * \beta^{\text{expoente}} \quad (8)$$

Por definição, na notação de ponto flutuante a parte inteira do significando é representada por apenas um dígito, ficando os restantes dos dígitos para a representação da parte fracionária do mesmo.

Desta forma, o valor expresso para a notação de ponto flutuante passa a ser calculado a partir da expressão

$$\pm \sum_{i=0}^n di * \beta^{i-k} * \beta^{\text{expoente}} \quad (9)$$

Com uma pequena manipulação desta expressão, conforme disposto na expressão (10), pode-se facilmente perceber que o acréscimo do campo expoente tornou possível mover virtualmente o ponto separador da sua posição original para uma outra que permita representar o número com maior precisão.

$$\pm \sum_{i=0}^n di * \beta^{i-k} * \beta^{\text{expoente}} \rightarrow \pm \sum_{i=0}^n di * \beta^{i-k+\text{expoente}} \quad (10)$$

Desta forma, dependendo do conteúdo do campo expoente, pode-se *virtualmente* deslocar o "ponto separador" para qualquer posição da *string* numérica que representa o número. Em alguns casos, isso pode levar o ponto separador a se posicionar além do número de dígitos originalmente contidos no significando, aumentando assim virtualmente a precisão com que o valor é expresso. Os exemplos a seguir compara valores representados em uma notação de ponto flutuante de base decimal, com um significando com 3 dígitos, com a sua representação em uma notação de ponto fixo.

- a)  $1.23 * 10^1 = 12.3$
- b)  $1.23 * 10^4 = 12300.0$
- c)  $1.23 * 10^{-1} = 0.123$
- d)  $1.23 * 10^{-3} = 0.00123$

Apesar de conferir uma maior expressividade à notação, diferentemente do que muitos imaginam, a inclusão do campo expoente por si só, não aumenta a quantidade de números representáveis em ponto flutuante para além do que seria possível representar com o mesmo número de dígitos nas notações de Inteiros ou de Ponto Fixo. Mesmo na notação de ponto flutuante a quantidade de números representáveis continua um arranjo com repetição do número total de dígitos



utilizados na representação. Por exemplo, em uma representação de base binária com  $n$  dígitos, divididos 1 bit para sinal,  $k$  bits para o expoente e o restante dos  $n-k$  bits para o significando, teríamos:

$$2 * 2^{n-1-k} * 2^k = 2^n \text{ números representáveis} \quad (11)$$

Um detalhe importante na notação de ponto flutuante, é que o campo expoente pode assumir tanto valores positivos quanto valores negativos. No entanto, diferentemente do que normalmente se adota para a representação de números negativos na formatação binária, onde o padrão é utilizar a representação em complemento a dois, no padrão IEEE 754, a fim de facilitar as operações de comparação entre expoentes, adotou-se uma representação diferenciada, onde o zero é expresso por um valor de referência denominado de *Bias*. Assim, valores maiores que o *Bias* representam expoentes positivos, ao passo que valores menores que o *Bias* representam expoentes negativos. Desta forma, o valor real expresso pelo campo expoente deve ser sempre calculado pela diferença entre o valor contido no campo e o valor do *Bias*.

Assim sendo, uma notação de ponto flutuante é totalmente definida a partir da quantidade de dígitos no significando, da base numérica adotada, do valor de referência do expoente, ou *Bias*, e dos valores mínimo e máximo representáveis no campo expoente. A Tabela 3 mais a frente, traz a definição destes valores para as quatro notações de ponto flutuante definidas no padrão IEEE 754.

A inclusão do campo expoente e com ele a possibilidade de manipular virtualmente a posição do "ponto separador", tornou a notação de ponto flutuante muito mais expressiva que as notações de inteiro e ponto fixo.

A Tabela 1 e o esquema gráfico da Figura 2.2 a seguir demonstram o poder de representação da notação de ponto flutuante, em comparação com às notações de ponto fixo e de inteiros.

Na Tabela 1, apenas para fins didáticos, são comparadas representações binárias hipotéticas com 8 bits nas três notações. Em todas elas o bit mais

significativo está reservado para a representação do sinal do número e os demais bits para a representação o seu módulo.

Na notação de ponto fixo, os bits reservados à representação do módulo do número foram distribuídos em dois grupos, um com 4 bits para a parte inteira e outro com 3 bits para a parte fracionária. Na notação de ponto flutuante estes mesmos bits foram separados, 1 bit para a parte inteira, 3 bits para a parte fracionária e 3 bits para o expoente.

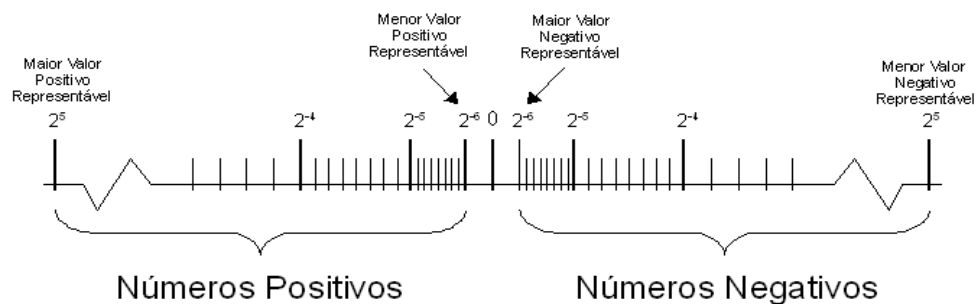
Pela análise desta tabela pode-se facilmente perceber que, independente da notação adotada e da configuração dos bits nestas notações, em todos os casos pode-se representar no máximo  $2^8$  ou 256 valores ou números diferentes.

**Tabela 1: Comparação entre a quantidade de valores representáveis nas notações de Inteiros, Ponto Fixo e Ponto Flutuante**

Notação	Composição	Valor. Min. representável (em Módulo)	Valor Max. representável(em Módulo)	Intervalo entre os valores	N. Valores
Inteiro	<b>S XXXXXX</b>	1	127	1	256
Ponto Fixo	<b>S XXXX.XXX</b>	0,125	15,875	0,125	256
Ponto Flut.	<b>S X.XXXEEE</b>	0,015625	30,000	Conforme o expoente	256

Na coluna Composição, S=Sinal, X= bits que compõem a parte inteira e a parte fracionária do número e E=bits que compõem o expoente do número.

Diferentemente das notações de inteiros e ponto fixo, na notação de ponto flutuante o intervalo entre dois valores consecutivos depende do valor do campo expoente, como se pode observar na Tabela 2 a seguir.



**Figura 2.2:** Distribuição dos valores representáveis em uma notação de ponto flutuante binária baseada na configuração do exemplo dado

**Tabela 2:** Exemplo de formatação dos valores gerados em ponto flutuante no formato dado como exemplo

Ponto Flutuante (S X.XXX EEE)				
Expoente	Significando	Vi. Max. Em Módulo	Intervalo entre Valores	Qtd. Valores
4	XXXX0,000	30,000	2,00	32
3	XXXX,000	15,000	1,00	32
2	XXX.X00	7,500	0,500	32
1	XX.XX0	3,750	0,250	32
0	X.XXX	1,875	0,125	32
-1	0,XXXX	0,9375	0,0625	32
-2	0,0XXXX	0,46875	0,03125	32
-3	0,00XXXX	0,234375	0,015625	32

Como se pode perceber, a grande vantagem do ponto flutuante é a sua capacidade de representar em um mesmo formato tanto valores muito pequenos quanto valores muito grandes com a precisão máxima permitida pelo número de

dígitos utilizados na notação adotada. No exemplo dado, com apenas 8 bits foi possível representar tanto valores tão pequenos como  $2^{-5}$  como valores tão grandes como  $2^5$ , o que naturalmente exigiria pelo menos 10 bits para ser representado.

## 2.3 O PADRÃO IEEE 754

O padrão IEEE 754 define os requisitos mínimos a serem seguidos na implementação da aritmética de ponto flutuante para a base numérica binária. Entre estes requisitos estão:

- **Formato de representação dos dados** – estão definidos quatro formatos de representação para os dados, denominados *Float*, *Double*, *Float Extended* e *Double Extended*. Todas as implementações do padrão IEEE 754 devem dispor de suporte para os tipos *Float* e *Double*, sem, no entanto, precisar obrigatoriamente dar suporte às formas estendidas destes formatos.
- **Operações aritméticas** - uma implementação completa do IEEE 754 deve conter as seguintes operações aritméticas: soma, subtração, multiplicação, divisão e raiz quadrada para todos os formatos de dados suportados.
- **Operações não aritméticas** – estão definidas também as operações de comparação, arredondamento para inteiro no formato de ponto flutuante, conversão de inteiro para ponto flutuante e conversão da representação decimal padrão para ponto flutuante e vice-versa.
- **Arredondamento** – para as operações aritméticas e de conversão estão definidos quatro modos de arredondamento denominados arredondamento para o mais próximo, ou *round to nearest*, arredondamento em direção ao zero, ou *round to zero*, arredondamento em direção ao +infinito, ou *round to +infinity*, e arredondamento em direção ao -infinito, ou *round to -infinity*.

- **Normalização** – com o intuito de garantir a unicidade na formatação dos dados, está definido que todo dado gerado como resultado de uma operação aritmética deve passar por um processo de normalização. O processo de normalização garante que o dígito mais significativo da representação será sempre diferente de zero.

### 2.3.1 Formato de representação dos dados

No padrão IEEE 754 os formatos de representação de ponto flutuante estão divididos pelo tipo de notação adotada, que pode ser básica ou estendida, e pela precisão, podendo ser de precisão simples ou dupla.

Desta forma, temos:

- Ponto flutuante de notação básica e precisão simples, também conhecido como ***float***, com 32 bits;
- Ponto flutuante de notação básica e precisão dupla, ou ***double***, com 64 bits;
- Ponto flutuante de *notação* estendida e precisão simples, ou *float extended* estendido, com 43 bits ou mais;
- Ponto flutuante de notação estendida e precisão dupla, ou *double extended* estendido, com 79 bits ou mais.

De um modo geral, segundo o padrão IEEE 754, um formato de ponto flutuante pode ser totalmente definido através da especificação dos seguintes parâmetros:

- $p$  = números de bits do significando (precisão)
- $E_{max}$  = expoente máximo

- $E_{min}$  = expoente mínimo
- **Bias** = valor de referência para o campo expoente

Definidos estes parâmetros, o valor de um número expresso em ponto flutuante qualquer pode ser definido pela seguinte expressão:

$$-1^s * d.ddd \dots d * \beta^e \quad (12)$$

Nesta expressão  $s$  representa o bit de sinal,  $d.ddd\dots d$  representa os  $p$  dígitos do significando, o qual está expresso na notação de ponto fixo com apenas um dígito à esquerda do ponto separador,  $\beta$  representa a base numérica adotada e, uma vez que o padrão IEEE 754 adota a base binária,  $\beta$  será sempre igual a 2, e por último,  $e$  representa o expoente do número na base adotada e tem seu valor  $\geq E_{min}$  e  $\leq E_{max}$ .

A mesma expressão também pode ser vista da seguinte forma:

$$-1^s * (d_0 + d_1 \beta^{-1} + \dots + d_{p-1} \beta^{-(p-1+e)}) , (0 \leq d_i < \beta) \quad (13)$$

A Tabela 3 a seguir demonstra os valores padrão para os parâmetros de configuração dos quatro formatos de ponto flutuante definidos pelo padrão IEEE 754.

Tabela 3: Distribuição dos bits e valores de referência para os formatos de ponto flutuante definidos no padrão IEEE 754

Campos	Formato			
	Simplex Float	Simplex Estendido Extended Float	Duplo Double	Duplo Estendido Extended Double
Sinal ( <b>S</b> )	1 bit	1 bit	1 bit	1 bit
Significando ( <b>P</b> )	24 bits	$\geq 32$ bits	53 bits	$\geq 64$ bits
Expoente ( <b>E</b> )	8 bits	$\geq 11$ bits	11 bits	$\geq 15$ bits
Expoente $E_{\max}$	+127	$\geq +1023$	+1023	$\geq +16383$
Expoente $E_{\min}$	-126	$\leq -1022$	-1022	$\leq -16382$
Expoente Bias	+127	Não definido	1023	Não definido

A fim de garantir a unicidade da representação, evitando com isto representações numéricas redundantes, o padrão IEEE 754 estabeleceu que o significando de todos os números seja expresso com a sua parte inteira representada por um único dígito da base numérica adotada, sendo este obrigatoriamente diferente de zero. Esta medida impede que números como o 1,0, tenham diversas representações redundantes, tais como  $1,0 \times 2^0$  ou  $0,1 \times 2^1$  ou ainda como  $0,01 \times 2^2$ . Isto, além de garantir a unicidade de representação, permite também para o caso de representações que utilizem a base binária, seja possível a dispensa do armazenamento do dígito que representa a parte inteira do significando, uma vez que o seu conteúdo é conhecido a priori.

Assim, de fato, no IEEE 754 armazenam-se apenas os bits que formam a parte fracionária do significando, os quais no padrão IEEE 754 recebem o nome de fração. Ao bit que não é armazenado dá-se o nome de bit escondido.

Esta é a notação considerada padrão para todas as notações de ponto flutuante do padrão IEEE 754. Assim, todo e qualquer número que atende a esta notação é tido como estando **“normalizado”**. Qualquer representação diferente desta será considerada **“desnormalizada”** e não poderá ser utilizada como entrada para nenhuma das operações definidas para este padrão.



### 2.3.2 Representação de Valores Especiais

A fim de corrigir incongruências de representação entre a notação adotada no padrão IEEE 754 e a aritmética clássica, foram definidas algumas representações especiais para valores que de outra forma não seriam representáveis.

Esta abordagem adotada pelo IEEE 754 não encontra equivalência em todos os sistemas que implementam algum tipo de representação de números reais. No padrão adotado no sistema IBM 370 por exemplo, todos os padrões de bits possíveis são utilizados para representar valores numéricos válidos. Neste padrão todas as operações que possam dar origem a valores não representáveis são encerradas como uma exceção acompanhada de uma mensagem de texto indicando a condição de falha [14].

No IEEE 754, foram definidas representações especiais para os seguintes casos:

- **Zero** – Devido ao padrão IEEE 754 não permitir a representação de números desnormalizados, foi necessário definir uma representação especial para o zero. Normalmente o zero só pode ser representado com o significando desnormalizado. A solução encontrada foi representá-lo como sendo o menor número representável na forma normalizada, ou seja, como  $-1^s \times 1,0 \times 2^{e_{\min}-1}$ . Desta forma, o zero é reconhecido por convenção e não diretamente por seu valor representado. Um outro detalhe importante na representação do zero no padrão IEEE 754, é que este dispõe de sinal, ou seja, existem  $+0$  e  $-0$ . Entretanto, a operação de comparação entre dois zeros retorna sempre verdadeira independente do sinal que estes assumam. A representação do zero com sinal é útil para definir o sinal do resultado de operações envolvendo o zero, as quais em alguns casos podem dar origem a  $+\infty$  e  $-\infty$ . A Tabela 4 e a Figura 2.5 a seguir trazem exemplos práticos de representação do +zero e do -zero.

- **NaN** – Esta sinalização, que literalmente informa que o valor armazenado não é um número, é reservada para ser utilizada como resultado de diversas operações aritméticas tais como:  $0/0$ ,  $0 \times \infty$ ,  $\sqrt{-1}$  e  $\infty + (-\infty)$ , que por definição, devem ser sinalizadas como inconsistentes. Qualquer operação aritmética onde pelo menos um dos operandos seja NaN deve retornar um NaN como resultado. Todas as operações aritméticas de comparação ( $=$ ,  $<$ ,  $\leq$ ,  $>$ ,  $\geq$ ) exceto ( $\neq$ ), onde pelo menos um dos operandos seja NaN devem retornar falso. O bit de sinal não tem função associada na representação do NaN. A Tabela 4 e a Figura 2.5 a seguir, trazem exemplos práticos de representação do NaN.
- **Infinito** – A representação de Infinito está reservada para demonstrar que um determinado número é, em módulo, maior que o maior número representável. Uma sinalização de Infinito tanto pode ser gerada em resposta a operações do tipo  $x/0$ , onde  $x \neq 0$ , como durante o processo de normalização e arredondamento. A semelhança do zero, o Infinito também tem sinal associado, o qual é útil para preservar o sinal das operações aritméticas. A Tabela 4 e a Figura 2.5 a seguir trazem exemplos práticos de representação do +Infinito e do -Infinito.
- **Denormal** – Esta representação foi adotada para tentar suprir a necessidade de sinalizar corretamente uma faixa de valores de ponto flutuante, comumente resultante de operações aritméticas, que apesar de serem corretamente representáveis no momento da operação em que são geradas, têm de ser convertidos para zero durante o processo de normalização e arredondamento por não poderem ser corretamente representados no modo normalizado. A Figura 2.3 a seguir indica a localização desta faixa de valores.

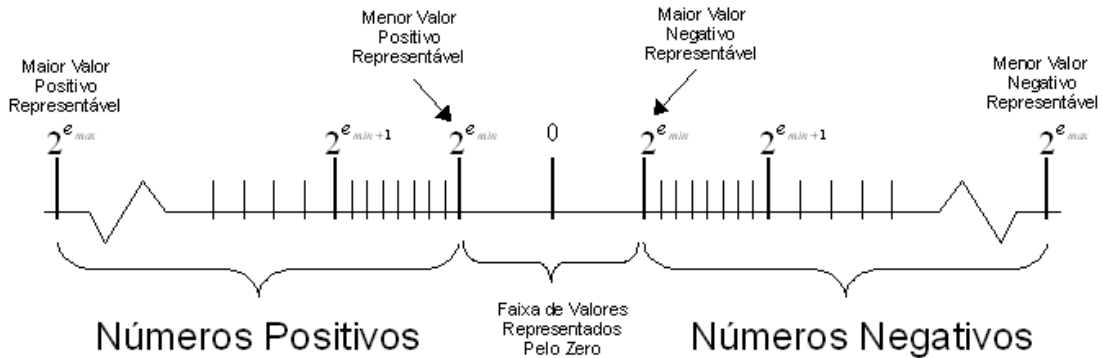


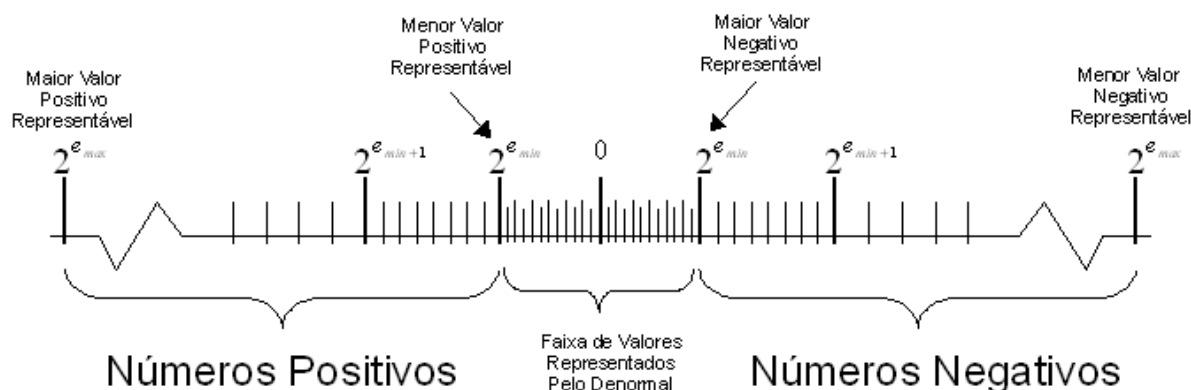
Figura 2.3: Distribuição dos valores representáveis por números normalizados em ponto flutuante indicando a faixa de valores representado pelo zero

Como um exemplo para esta situação, podemos considerar a operação de subtração entre os números  $n_1 = 6.87 \times 10^{-97}$  e  $n_2 = 6.81 \times 10^{-97}$ , ambos em uma notação de base decimal com expoente mínimo igual a  $10^{-98}$ . O resultado exato desta operação é  $0,06 \times 10^{-97}$ , o qual após a normalização é representado como  $6,0 \times 10^{-99}$ . Entretanto, como este não é um valor representável no formato de ponto flutuante adotado, conforme pode-se ver na Figura 2.3, porque causa um underflow no expoente, o mesmo deve ser arredondado para zero.

O problema é que situações como estas podem criar inconsistências em trechos de código do tipo:

```
if ( $n_1 \neq n_2$ )
    then  $z = 1/(n_1 - n_2);$ 
```

Desta forma, a adoção de uma representação para números **denormalizados** não apenas resolve problemas desta natureza como aumenta a exatidão dos resultados das operações aritméticas, uma vez que define um novo conjunto de números representáveis, localizados entre o menor número positivo e o maior número negativo representáveis e o zero, conforme apresentado na Figura 2.4 a seguir:



**Figura 2.4:** Distribuição dos valores representáveis por números normalizados em ponto flutuante indicando a faixa de valores representados pelo números denormalizados

A fim de poder sinalizar esta nova faixa de valores de maneira diferenciada, uma vez que estes representam números obtidos a partir de um processo de normalização que não pode ser incluído, no padrão IEEE 754, a semelhança do que já ocorre com o número zero, estes números são representados, por convenção, associados ao expoente  $e_{\min}-1$ , e não em relação a  $e_{\min}$  como seria de se esperar. Na prática, ao ser reconhecido como um número denormal, este deve ter seu expoente primeiramente convertido antes que possa ser operado.

Desta forma, adotando-se a representação dos números denormalizados, o resultado do exemplo anteriormente apresentado  $6,0 \times 10^{-99}$ , seria agora representado como  $0,6 \times 10^{-98}$ . Isto aumentaria a precisão de sua representação e impediria a ocorrência de inconsistências como a apresentada no exemplo dado.

Deve-se ter uma especial atenção na implementação de operações com números denormalizados, uma vez que estes apesar de serem armazenados como números normalizados, devem ser operados como números desnormalizados, ou seja, com o bit escondido igual a zero.

A Figura 2.5 e a Tabela 4 a seguir trazem exemplos práticos da representação dos números especiais no padrão IEEE 754.

Tabela 4: Valores Especiais e suas representações segundo o padrão IEEE 754

	Sinal	Significando	Expoente
Zero	X	1.000...0	0000...00
+ Infinito	0	1.000...0	1111...11
- Infinito	1	1.000...0	1111...11
NaN	X	1.NNN...N	1111...11
Denormal	X	1.NNN...N	0000...00

Obs.: Na Tabela 4 e na Figura 2.5, X= bit com valor '0' ou '1' e NNN...N = *string* de bits, onde pelo menos uma das posições é diferente de zero.

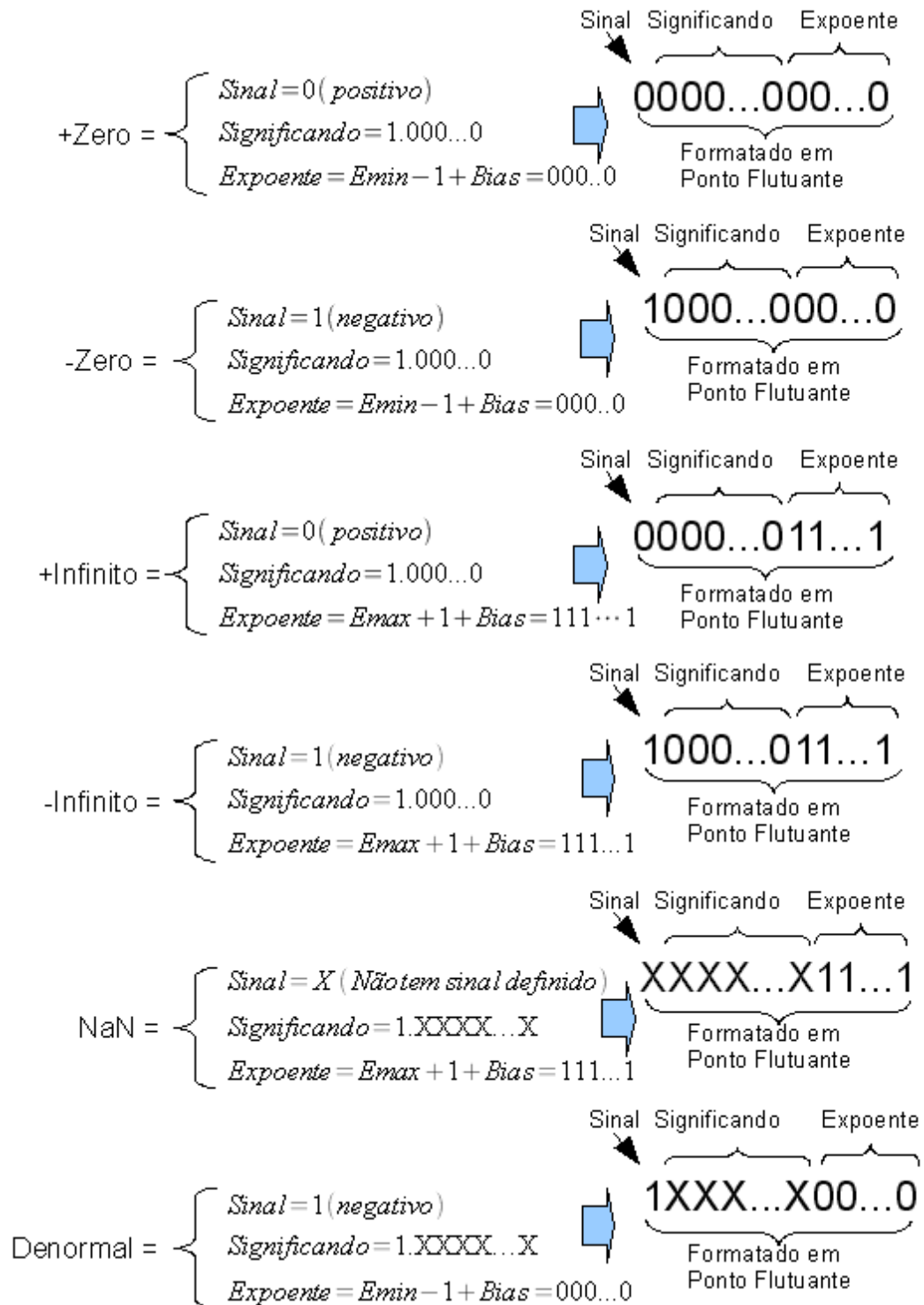


Figura 2.5: Exemplos de representação dos valores especiais do padrão IEEE 754.

### 2.3.3 Arredondamento

O termo arredondamento segundo o padrão IEEE 754 é o processo de ajustar ou encaixar um número tido como “*infinitamente preciso*” para um formato de menor precisão, que não dispõe de todos os dígitos necessários a sua representação com a precisão original [19].

Ao trabalhar-se com números em ponto flutuante, deve-se ter em mente que o processo de arredondamento introduzirá um erro no número arredondado, uma vez que o número arredondado é na verdade uma aproximação do valor original.

Diversos são os relatos de falhas em sistemas ocasionadas por estes erros de arredondamento [22]. Padrões mais recentes de representação tais como o IEEE 854 [20] e o IEEE P754 [17] buscam minimizar estes erros adotando outras bases numéricas e/ou uma representação com um maior número de dígitos.

Existem a princípio duas situações nas quais um número real não pode ser representado de forma exata em uma notação de ponto flutuante. A primeira é quando o número não encontra representação finita na base adotada, ou quando, ainda que encontrando, sua representação exige mais dígitos que o disponível no formato de ponto flutuante adotado. Esta situação pode ser ilustrada pelo decimal 0.1, o qual, apesar de poder ser expresso com uma representação finita quando representado na base decimal, torna-se uma dízima quando representado na base binária [14], conforme se pode constatar a seguir:

$$0.1_{10} = 1.0 * 10^{-1}$$



$$0.1_2 = 1.10011001100110011001101 \dots * 2^{-4}$$

A outra situação ocorre quando o valor a ser representado é menor que o menor ou maior que o maior número representável no formato de ponto flutuante adotado, nestes casos o número é arredondado para + ou - infinito ou para zero.

Do ponto de vista da implementação, a função arredondamento nada mais é que um processo de mapeamento, onde busca-se identificar dentre todos os números representáveis na notação destino aquele que melhor represente o número a ser arredondado.

Considere  $F_1$  e  $F_2$ , dois sistemas de ponto flutuante.  $F_1$  representado com um significando com  $n_1$  dígitos e  $F_2$  representado com  $n_2$  dígitos,  $n_1 > n_2$ . Verifica-se que todo o número representável em  $F_1$ , a exceção dos números que forem maior que o maior ou menor que o menor número representável em  $F_2$ , ao ser arredondado para  $F_2$ , ou será precisamente representável ou sua representação estará entre dois outros números representáveis na notação destino, aos quais denominamos de adjacente superior e adjacente inferior. Esta característica pode ser explicada pelo fato de o significando ser representado em uma notação de ponto fixo, tendo, nas duas notações o mesmo número de dígitos para representar a sua parte inteira. Desta forma, a diferença no poder de representação dos significandos se dará sempre pela sua capacidade de expressar a parte fracionária do número representado. A Figura 2.6 a seguir demonstra a diferença entre o poder de representação de um significando com 5 bits quando comparado com um de 3 bits.

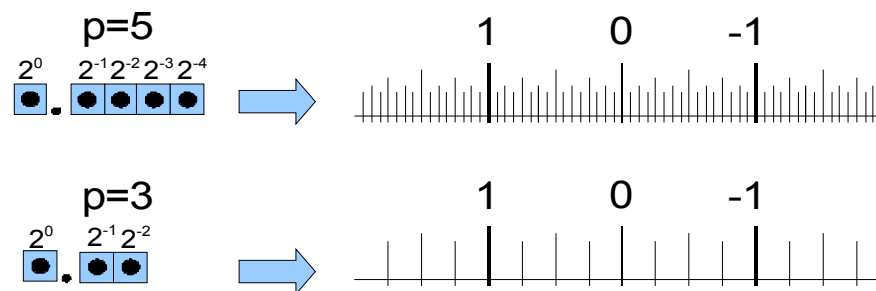


Figura 2.6: Diferença na representação entre um significando com 3 e um com 5 bits



Como se pode perceber, valores múltiplos de  $2^{-2}$  como 0, 1,  $1/2$  ou  $3/4$ , podem ser representados precisamente tanto na primeira quanto na segunda notação. Já valores múltiplos de  $2^{-3}$  e  $2^{-4}$  que não forem múltiplo de  $2^{-2}$ , tais como  $1/8$  ou  $3/16$  só podem ser representados precisamente na primeira notação, tendo que ser aproximados, ou arredondados, na segunda.

Na Figura 2.7 a seguir, vemos em destaque o exemplo do conjunto de valores existentes no intervalo entre 0 e  $1/4$  para a notação com o significando com 5 bits. Apesar de representáveis com o significando de 5 bits, estes valores precisam ser arredondados para serem representados com o significando com apenas 3 bits. Neste exemplo, cada um dos valores representados terá que ser arredondado ou para 0 ou para  $1/4$ , uma vez que estes são, respectivamente o adjacente inferior e o adjacente superior dos valores originais na notação que adota o significando com apenas 3 bits.

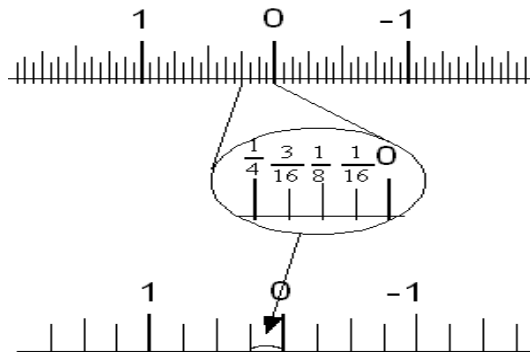


Figura 2.7: Comparação entre valores representáveis com significandos com 5 e 3 bits.

### 2.3.4 Modos de Arredondamento

No padrão IEEE 754 estão definidos os seguintes modos de arredondamento:

- **Arredondamento em direção ao zero (*Round to Zero*)** - Neste modo, ilustrado na Figura 2.5, o número a ser operado é arredondado para o primeiro valor representável mais próximo em direção ao zero. Esta é

com certeza a forma mais simples e direta de arredondamento. Na prática apenas eliminam-se os *bits* além do último *bit* representável fazendo com que o número resultante esteja assim mais próximo do zero que o número original.

### Round to Zero

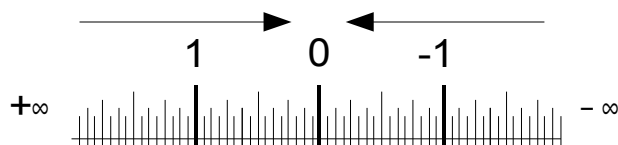


Figura 2.8: Direção do arredondamento em direção ao Zero

- **Arredondamento em direção ao  $+\infty$  (*Round to  $+\infty$* )** – Neste modo, arredonda-se o número para o número representável mais próximo em direção ao infinito positivo. Desta forma, números negativos devem que ser arredondados para o seu adjacente inferior, enquanto números positivos devem ser arredondados para o seu adjacente superior.

### Round to $+\infty$

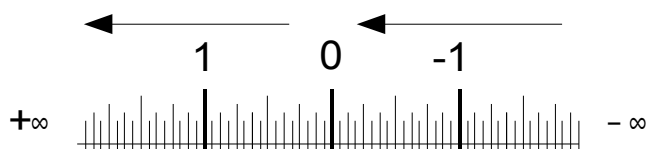


Figura 2.9: Direção do arredondamento em direção ao  $+\infty$

- **Arredondamento em direção ao  $-\infty$  (*Round to  $-\infty$* )** – Este formato é semelhante ao anterior, diferindo apenas pelo fato que o arredondamento se dará sempre para o número representável mais próximo em direção ao infinito negativo. Desta forma, números negativos devem que ser arredondados para o seu adjacente superior,

enquanto números positivos devem ser arredondados para o seu adjacente inferior.

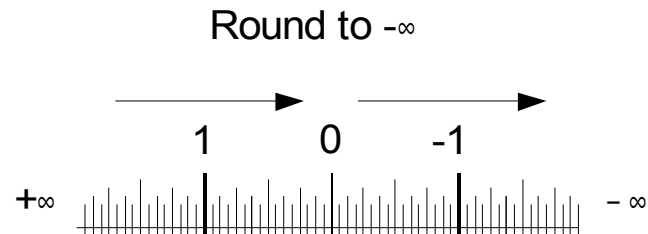


Figura 2.10: Direção do arredondamento em direção ao  $-\infty$

- **Arredondamento para o mais próximo ou par (Round to Nearest Even)** - Neste modo busca-se o arredondamento para o número representável mais próximo do valor originalmente calculado e, em havendo empate entre dois números igualmente distantes, decide-se sempre pelo adjacente que for par.

Este é o formato mais utilizado nas implementações do padrão IEEE 754, tanto em software quanto em hardware. Também é aquele que apresenta os resultados mais exatos. Em contrapartida, é também aquele que exige mais recurso computacional para ser implementado.

Este modo de arredondamento só não apresenta os melhores resultados quando a sua aplicação resulta em um número que, após a normalização, se transforme em um número maior que o maior ou menor que o menor número representável, o qual acaba sendo convertido ou para infinito ou para zero.

Existem na literatura diversas propostas de implementação para este modo de arredondamento, incluindo variações, que de um modo geral ou buscam melhorar o seu desempenho ou reduzir o hardware e/ou o tamanho de código necessário a sua implementação [23][24] [25]

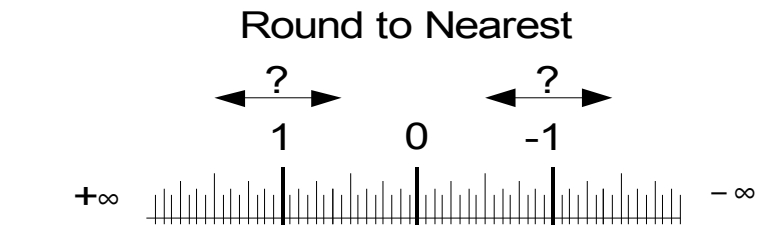


Figura 2.11: Direção do arredondamento para o mais próximo ou par

As Figuras 2.13 e 2.14 a seguir trazem dois exemplos de números gerados originalmente com um significando com 5 bits e suas representações após terem sido arredondados, seguindo os métodos apresentados, para um significando com apenas 3 bits.

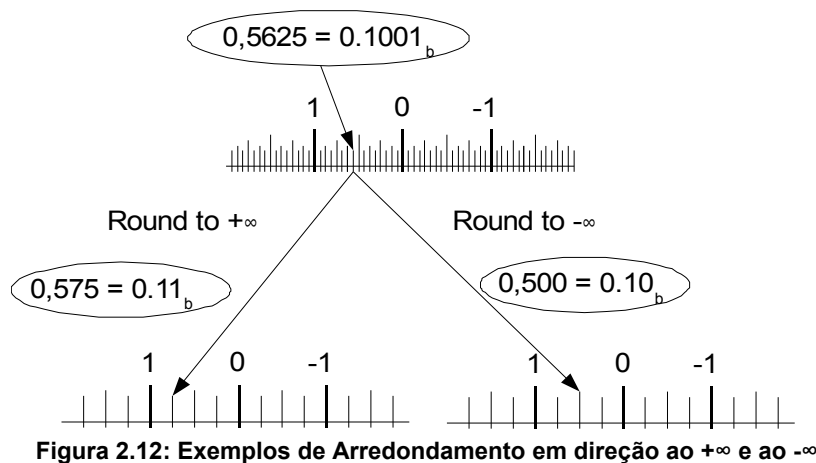


Figura 2.12: Exemplos de Arredondamento em direção ao  $+\infty$  e ao  $-\infty$

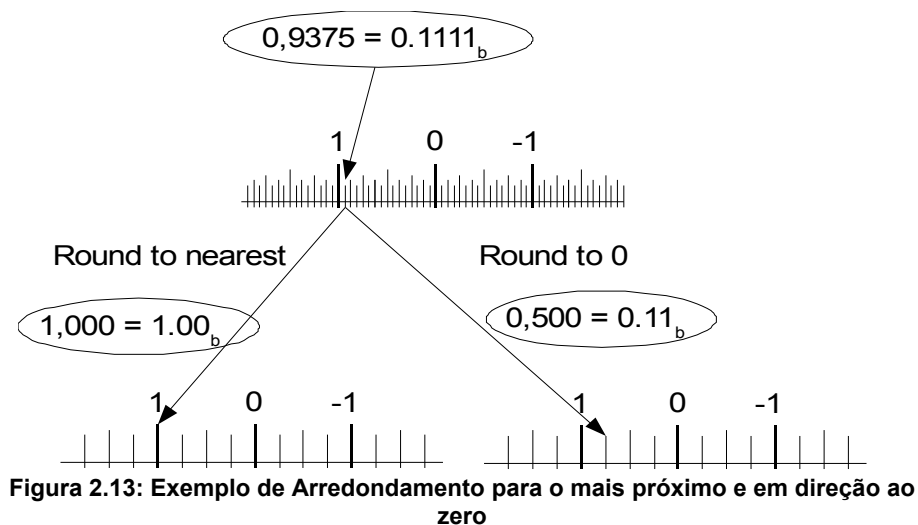


Figura 2.13: Exemplo de Arredondamento para o mais próximo e em direção ao zero

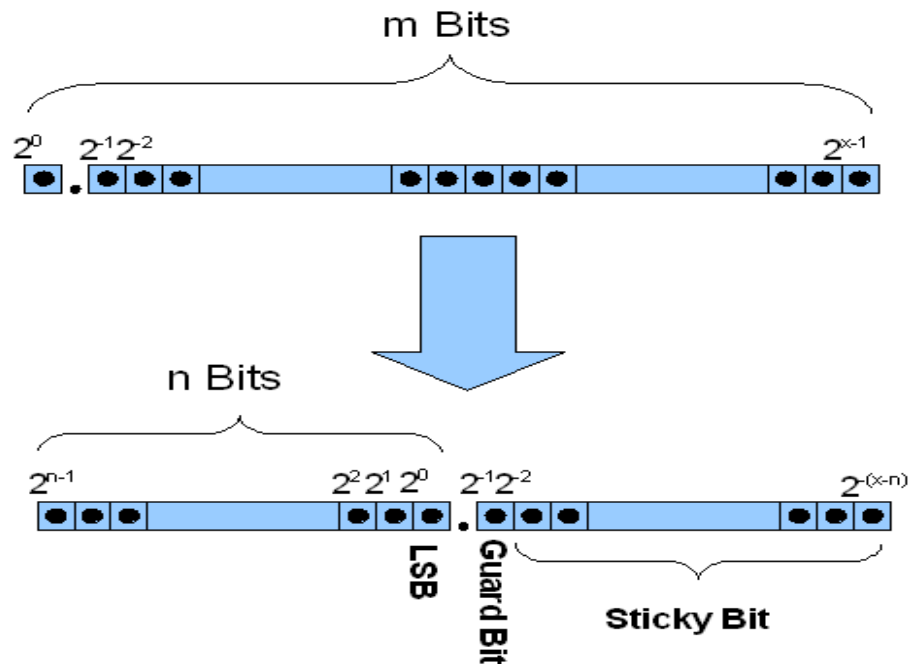


Figura 2.14: Esquema para obtenção do LSB bit, do Guard bit e do Sticky bit.

### 2.3.5 Algoritmos de Arredondamento

Os algoritmo de arredondamento aqui utilizado basearam-se em grande parte nos trabalhos apresentados em [24] e [25].

Por estes algoritmos, todos os quatro modos de arredondamento definidos no padrão IEEE 754 compartilham de um mesmo processo inicial de pré-processamento, o qual tem como objetivo a geração de um significando pré-arredondado e a obtenção de algumas outras informações que orientarão o restante do processo de arredondamento.

Em linhas gerais, dado um número em ponto flutuante com um significando com  $m$  bits, o qual deseja-se arredondar para uma representação com  $n$  bits, primeiramente separam-se os  $n$  bits mais significativos do significando original, a fim de formar o significando pré-arredondado. Em seguida, identificam-se três bits especiais denominados *LSB* (*Least Significant Bit*), *Guard bit* e *Sticky bit*, os quais

são obtidos do significando original conforme indicado no esquema da Figura 4.2. Por fim, durante o processo de arredondamento, a partir da análise do modo de arredondamento, do sinal do número a ser arredondado e dos três bits especiais, defini-se o valor a ser atribuído a um quarto bit, denominado *Round Bit*, o qual deve ser adicionado ao significando pré-arredondado a fim de obter-se o significando final arredondado, conforme mostra a 2.15 a seguir.

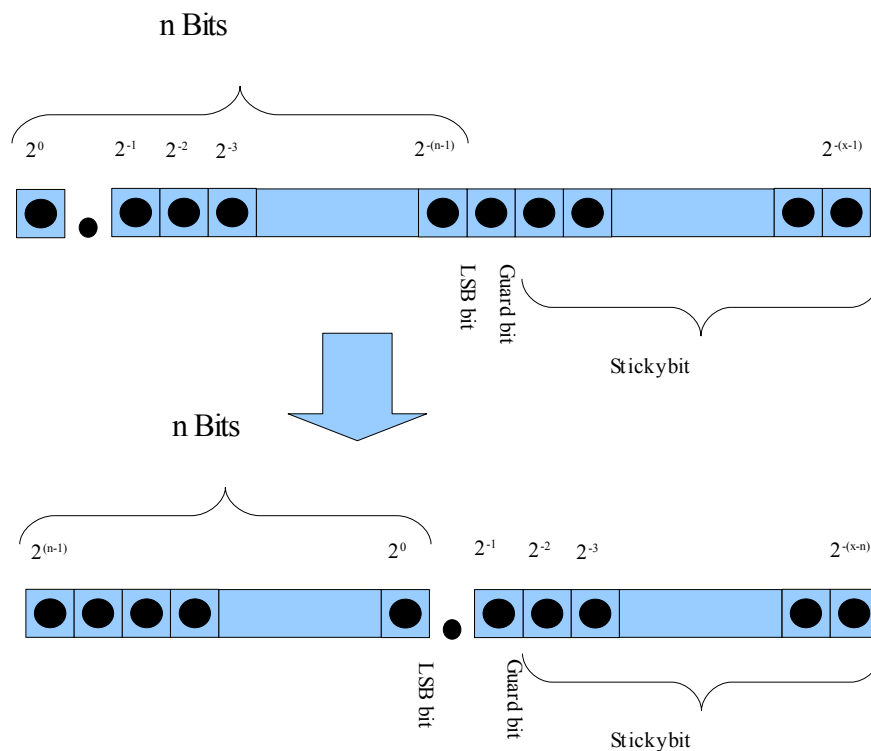


Figura 2.15: Nova organização dos bits para simplificar a análise do processo de arredondamento

Com relação aos bits especiais, o *bit Guard Bit*, conforme se pode observar na Figura 4.2, dividirá os bits do significando original em dois grupos, um à esquerda, com os  $n$  bits mais significativos, os quais formarão o significando pré-arredondado, e um à direita, com o restante dos bits que serão descartados. O *Sticky Bit* será obtido do resultado da aplicação da função lógica “OU” envolvendo todos os bits do grupo de bits à direita do *Guard bit*. O *LSB Bit* é o bit menos significativo do significando pré-arredondado.

Neste ponto, apenas para simplificar a análise do processo de arredondamento, reorganizaremos os *bits* do significando deslocando o ponto separador da sua posição original para a posição entre o *LSB Bit* e o *Guard Bit*, conforme disposto na Figura 2.15.

Esta nova organização dos *bits* torna-se interessante porque nos permite visualizar o processo de arredondamento de um número qualquer como sendo o processo de arredondamento de um número fracionário para um número inteiro. E, dado que todo número fracionário situa-se no intervalo definido por seus dois inteiros adjacentes, como podemos ver no esquema dado a seguir, podemos resumir o nosso processo de arredondamento como sendo o processo de escolha de para qual destes dois inteiros adjacentes o número fracionário deverá ser arredondado.

$$(n + 1) \leftarrow n.zzz \cdots zz \rightarrow (n)$$

Desta forma, considerando o esquema proposto, temos os seguintes exemplos:

- a)  $10 \leftarrow 9.21 \cdots \rightarrow 9$  (9.21 pode ser arredondado para 9 ou para 10)
- b)  $3 \leftarrow 2.73 \cdots \rightarrow 2$  (2.73 pode ser arredondado para 2 ou para 3)
- c)  $236 \leftarrow 235.21 \cdots \rightarrow 235$  (235.21 pode ser arredondado para 235 ou para 236)

Com isto podemos perceber que, o que está envolvido no processo de arredondamento de um número Real qualquer, não representável em uma determinada notação de ponto flutuante, é o processo de mapeamento ou escolha, entre qual dos seus adjacentes representáveis nesta notação o substituirá. Nesta escolha, o modo de arredondamento adotado é que norteará, em última instância, a escolha sobre qual dos dois adjacentes deverá ser utilizado.

Assim sendo, após a execução dos passos iniciais, aplica-se o algoritmo de arredondamento específico associado ao modo de arredondamento escolhido, de forma a poder valorar adequadamente o *Round bit*.

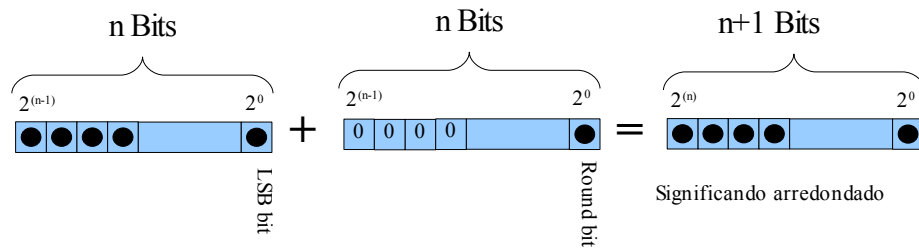


Figura 2.16: Geração do novo significando a partir dos bits extraídos do significando original e do Round bit obtido durante o processo de arredondamento.

O valor atribuído ao *Round bit* definirá, em última instância, para qual dos números adjacentes representáveis o número original será arredondado, de tal forma que:

- Quando o *Round Bit* for zero, arredondar-se-á o número para o valor adjacente inferior, ou seja, para aquele com o menor valor em módulo.
- Por outro lado, quando o *Round Bit* for um, arredondar-se-á o número para o adjacente superior, ou seja, para aquele que tem o maior valor em módulo.

### 2.3.5.1 Algoritmo do modo de arredondamento em direção ao Zero

Neste modo de arredondamento, o número será sempre arredondado para o seu adjacente inferior, ou para aquele adjacente que esteja mais próximo do zero, conforme indica a Figura 2.8. Para tanto, simplesmente atribui-se zero ao *Round Bit*. Em síntese, simplesmente eliminam-se os bits excedentes do significando a ser



arredondado, preservando os  $n$  bits mais significativos como sendo o significando arredondado.

### 2.3.5.2 Algoritmo do modo de arredondamento em direção ao + e ao - Infinito

Uma vez que o algoritmo utilizado no modo de arredondamento em direção ao +Infinito é muito semelhante ao aplicado no modo de arredondamento em direção ao -Infinito, *apresentaremos primeiramente* os detalhes do algoritmo para o modo de arredondamento em Direção ao -Infinito, em seguida apenas indicaremos os passos que diferenciam este procedimento do adotado *no modo de arredondamento em direção ao +Infinito*.

No modo de Arredondamento em Direção ao -Infinito, conforme se pode observar da Figura 2.10, os números positivos devem ser arredondados para o seu adjacente inferior, enquanto os números negativos devem ser arredondados para o seu adjacente superior. Para atingir tal objetivo, aproveita-se o fato de o *bit* de sinal ter valor zero para números positivos e um para números negativos, e apenas atribui-se o bit de sinal diretamente ao *Round bit*, de tal forma que o *Round bit* seja igual a zero para os números positivos e um para os números negativos. Figura 2.9

No modo de Arredondamento em Direção ao +Infinito, aplica-se a mesma lógica, com a única diferença que o *Round bit* é obtido a partir do inverso do *bit* de Sinal. Assim este bit será zero quando se estiver arredondando um número negativo e um quando o número for positivo.

### 2.3.5.3 Algoritmo do modo de arredondamento em direção ao mais próximo ou par

Como o próprio nome diz, neste modo busca-se arredondar o número para o adjacente representável que estiver mais próximo do número original, ou, em caso de empate, para o adjacente que for par.

Desta forma, conforme indicado na Figura 2.11, não se pode definir para qual dos adjacentes um determinado número será arredondado sem antes proceder uma análise do número a ser arredondado, o que é feito através do *Guard bit*, do *Sticky bit* e do *LSB*.

Segundo o esquema proposto na Figura 2.15, em nossa análise o *Guard Bit* estará diretamente associado ao valor atribuído ao *bit* ponderado com  $2^{-1}$ , ou 0,5, no somatório das partes que compõe esta nova parte fracionária do número. O *Sticky Bit* por sua vez está associado à ocorrência de um valor diferente de zero no somatório dos *bits* que tenham peso associado menor que  $2^{-1}$ . O esquema proposto na Figura 2.17 demonstra como se procede esta análise.

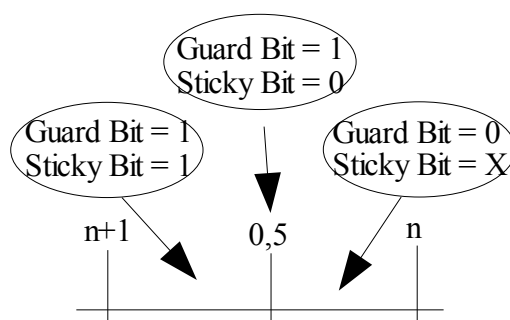


Figura 2.17: Localização do valor relativo do número a ser arredondado a partir do *Guard Bit* e do *Sticky Bit*

Primeiramente, observa-se a parte fracionária do número através dos valores atribuídos ao *Guard-bit* e ao *Sticky-bit*. Nesta observação podemos chegar as seguintes situações:

- a) O *Guard-bit* e o *Sticky-bit* são ambos iguais a '1', o que indica que a parte fracionária do número é maior que 0,5, e que portanto o número a ser arredondado está mais próximo do seu adjacente superior do que do seu adjacente inferior. Neste caso o *Round-bit* deve ser feito igual a '1'.
- b) O *Guard-bit* e o *Sticky-bit* são ambos iguais a '0', o que indica que a parte fracionária do número é também igual a '0', e que portanto o número é coincidente com o seu adjacente inferior. Neste caso o *Round-bit* deve ser feito igual a '0'.
- c) O *Guard-bit* é '0' e o *Sticky-bit* é '1', o que indica que a parte fracionária do número é menor que 0,5, mas maior que '0', estando desta forma o número mais próximo do seu adjacente inferior do que do adjacente superior. Neste caso o *Round-bit* deve ser feito igual a '0'.
- d) O *Guard-bit* é '1' e o *Sticky-bit* é '0', o que indica que a parte fracionária do número é exatamente igual a 0,5, estando o número desta forma a igual distância dos seus dois adjacentes representáveis. Neste caso o número deve ser arredondado para o adjacente par. Uma forma prática de identificar o adjacente par é observando o *LSB*. Uma vez que, por nossa análise, o adjacente inferior é o próprio número representado sem os bits que serão eliminados, quando o *LSB for igual a zero, ou seja, quando o bit menos significativo da parte que será preservada* for igual a zero, o número e, conseqüentemente, o adjacente inferior, será par. De maneira análoga, quando o *LSB for igual a '1'* o adjacente inferior será ímpar e conseqüentemente o adjacente superior será par. Desta forma, pode-se simplesmente atribuir o valor do *LSB* ao *Round-bit* a fim de concluir o processo de arredondamento.

A Tabela 5 a seguir demonstra de maneira sucinta os possíveis valores para os bits *Guard Bit*, *Sticky Bit* e *LSB* e os valor a ser associado ao *Round Bit* para cada uma das combinações.

**Tabela 5: Definição do valor do Round Bit no modo Round to Nearest**

Guard Bit	Sticky Bit	LSB Bit	Round Bit
0	X	X	0
1	0	0	0
1	0	1	1
1	1	X	0

Neste trabalho será implementado apenas o modo de arredondamento para o mais próximo ou par, por ser este o que apresenta os resultados mais exatos e por ser este também o modo padrão de arredondamento do IEEE 754.

### 2.3.6 Normalização

Conforme descrito na Seção 2.3.1 desta dissertação, a fim de garantir a unicidade de representação dos números em ponto flutuante, o padrão IEEE 754 adota o padrão de representação normalizada para todos os valores representáveis. Como o IEEE 754 adota a base binária, isto significa dizer que todo número deve ser representado com a parte inteira do significando igual a 1. Toda a representação que não atende a este padrão é considerada desnormalizada e não pode ser corretamente operada.

Operações aritméticas, mesmo que operando apenas com números normalizados, podem tanto gerar resultados normalizados quanto resultados desnormalizados. A seguir temos alguns exemplos de operações entre operandos normalizados que apresentam resultados desnormalizados.

$$a) 3,0 * 10^0 * 3,0 * 10^0 = 9,0 * 10^0$$



$$(1.100...00b * 2^1) * (1.100...00b * 2^1) = (10.010...00b * 2^2)$$

$$b) 1,0 * 10^0 + 1,0 * 10^0 = 2,0 * 10^0$$



$$(1.000...00b * 2^0) + (1.000...00b * 2^0) = (10.000...00b * 2^0)$$

$$c) 1,0 * 10^0 - 5,0 * 10^{-1} = 0,5 * 10^0$$



$$(1.000...00b * 2^0) - (1.000...00b * 2^{-1}) = (0.100...00b * 2^0)$$

Como se pode perceber, quando implementados na forma binária, apesar de expressarem corretamente os resultados das operações, todos as três operações apresentam resultados em desacordo com o padrão de representação adotado no IEEE 754, sendo portanto considerados desnormalizados.

Nos dois primeiro exemplos os resultados são considerados desnormalizados por apresentarem a parte inteira do significando maior que 1. No terceiro exemplo o resultado é considerado desnormalizado por apresentar a parte inteira igual a zero.

A seguir é apresentado o algoritmo de normalização adotado neste projeto.

### 2.3.6.1 Algoritmo de Normalização

O algoritmo aqui apresentado é destinado a implementações de ponto flutuante em representação binária, entretanto, com os devidos ajustes, este algoritmo pode também ser aplicado a implementações de ponto flutuante em representação decimal:

1. Identifica-se a posição do primeiro bit igual a um no significando.
2. Calcula-se a distância da posição do primeiro *bit* em '1' encontrado, para a posição padrão em que este deveria se encontrar para números normalizados. Ou seja, calcula-se a distância deste para a primeira posição a esquerda do ponto separador. Estando este à esquerda da posição esperada a distância é considerada positiva, estando este à direita a distância é considerada negativa.
3. Efetua-se o deslocamento dos *bits* do significando de forma a colocar o primeiro *bit* em na posição correta, ou seja, na primeira posição à esquerda do ponto separador.
4. Uma vez que o procedimento de deslocar um número para a direita equivale a dividi-lo pela base adotada, ao mesmo tempo que desloca para a esquerda equivale a multiplicá-lo pela mesma base, a cada deslocamento do significando, tanto para a direita como para a esquerda, deve-se proceder a correção do expoente, de forma a manter o valor originalmente expresso pelo número representado.
5. Deve-se ter em mente que o procedimento de normalização não deve modificar o valor originalmente expresso pelo número representado. Desta forma, não deve haver descarte dos dígitos que forem sendo deslocados.

6. Por fim, havendo necessidade, deve-se proceder o arredondamento do significando após o processo de normalização.

Desta forma, tomando como exemplo os resultados obtidos nos exemplos anteriores teríamos:

- Para os resultados obtidos no exemplos (a) e (b), apresentados no início desta seção, respectivamente  $10.010...00b * 2^2$  e  $10.000...00b * 2^0$ , temos que: o primeiro bit em '1' está deslocado uma posição à esquerda da posição onde deveria se encontrar. Portanto, deve-se ao mesmo tempo proceder o deslocamento do significando em uma posição à direita e efetuar o incremento proporcional do expoente. Desta forma teríamos respectivamente:  

$$10.010...00b * 2^2 \rightarrow 1.0010...00b * 2^3$$

$$10.000...00b * 2^0 \rightarrow 1.0000...00b * 2^1$$
- Para o resultado obtido no exemplo (c),  $0.100...00b * 2^0$  temos: o primeiro bit em '1' está deslocado uma posição à direita da posição onde deveria se encontrar. Neste caso deve-se ao mesmo tempo proceder o deslocamento do significando em uma posição à esquerda e efetuar o decremento proporcional do expoente. Como resultado teríamos:  $0.100...00b * 2^0 \rightarrow 1.000...00b * 2^{-1}$

### 2.3.7 Operações Aritméticas

O IEEE 754 determina que todas as suas implementações dêem suporte às operações aritméticas de soma, subtração, multiplicação, divisão e resto da divisão inteira, definida pelo operador *REM*.

Apesar de não fazer restrições no tocante aos algoritmos a serem adotados, o IEEE 754 define que todas as operações aritméticas devam primeiramente

produzir um resultado definido como infinitamente preciso, com tantos dígitos quantos forem necessários à sua representação mais precisa, os quais devem então ser arredondados e normalizados para o formato de ponto flutuante adotado.

Também estão definidos os resultados esperados para as operações em que um ou os dois operandos envolvidos não estejam normalizados. Nestas situações, deve-se observar um fluxo alternativo de execução, de modo a obter os resultados identificados nas Tabelas 6,7,8 e 9.

Tabela 6: Resultados esperados para operações de multiplicação  $a * b$ .

<b>b \ a</b>	<b>Zero</b>	<b>Denorm *</b>	<b>OK</b>	<b>NaN</b>	<b>Infinito</b>
<b>Zero</b>	Zero	Zero	Zero	NaN	Zero
<b>Denorm *</b>	Zero	Zero	Zero	NaN	Zero
<b>OK</b>	Zero	Zero	OK	NaN	Infinito
<b>NaN</b>	NaN	NaN	NaN	NaN	NaN
<b>Infinito</b>	NaN	NaN	Infinito	NaN	Infinito

Tabela 7: Resultados esperados para operações de divisão  $a/b$ .

<b>b \ a</b>	<b>Zero</b>	<b>Denorm *</b>	<b>OK</b>	<b>NaN</b>	<b>Infinito</b>
<b>Zero</b>	NaN	NaN	NaN	NaN	NaN
<b>Denorm *</b>	NaN	NaN	NaN	NaN	NaN
<b>OK</b>	Zero	Zero	OK	NaN	Infinito
<b>NaN</b>	NaN	NaN	NaN	NaN	NaN
<b>Infinito</b>	Zero	Zero	Zero	NaN	NaN

Tabela 8: Resultados esperados para operações de soma  $a+b$ .

<b>b \ a</b>	<b>Zero</b>	<b>Denorm *</b>	<b>OK</b>	<b>NaN</b>	<b>Infinito</b>
<b>Zero</b>	Zero	Zero	Zero	NaN	Infinito
<b>Denorm *</b>	Zero	Zero	OK	NaN	Infinito
<b>OK</b>	OK	OK	OK	NaN	Infinito
<b>NaN</b>	NaN	NaN	NaN	NaN	NaN
<b>Infinito</b>	Infinito	Infinito	Infinito	NaN	**



Tabela 9: Resultados esperados para operações de subtração a-b.

<b>b \ a</b>	<b>Zero</b>	<b>Denorm *</b>	<b>OK</b>	<b>NaN</b>	<b>Infinito</b>
<b>Zero</b>	Zero	Zero	OK	NaN	Infinito
<b>Denorm *</b>	Zero	Zero	OK	NaN	Infinito
<b>OK</b>	OK	OK	OK	NaN	Infinito
<b>NaN</b>	NaN	NaN	NaN	NaN	NaN
<b>Infinito</b>	Infinito	Infinito	Infinito	NaN	NaN

Obs.: \*Em nossa implementação, operandos sinalizados como Denorm, são automaticamente convertidos para zero. \*\*O resultado de operações de subtração entre dois operandos sinalizados como infinito podem resultar infinito, quando os sinais dos operandos coincidem, ou NaN, quando os sinais não são coincidentes.

Neste trabalho, uma vez que o objetivo final é implementar uma unidade de multiplicação e acumulação, serão implementadas apenas as operações de soma, subtração e multiplicação de ponto flutuante, descritas em detalhes a seguir.

### 2.3.7.1 Operações de Soma e Subtração

De um modo geral, pode-se dizer que as operações de soma e subtração em ponto flutuante compartilham de um mesmo algoritmo de execução. Em ambos os casos deve-se, primeiramente preparar os operandos, em seguida efetuar a operação desejada e, por fim, normalizar e arredondar o resultado para o formato de ponto flutuante adotado.

O procedimento de preparação dos operandos tem como objetivo alinhar os bits dos significandos entre si, de tal forma que a operação de soma ou subtração ocorra somente entre *bits* que estejam sendo associados à pesos equivalentes.

Este procedimento se faz necessário uma vez que, como foi demonstrado na Secção 1.2.2, o valor do campo expoente irá alterar o peso atribuído aos *bits* do significando, fazendo com que o ponto separador seja virtualmente deslocado da sua posição original.

Considere, para uma melhor compreensão, os operandos Op1 e Op2, os quais, poderiam ser expressos da seguinte forma:

$$Op1 \rightarrow -1^{sinal} * (b_n \beta^{e_1} + b_{n-1} \beta^{e_1-1} \dots + b_1 \beta^{e_1-(n+1)} + b_0 \beta^{e_1-n})$$

$$Op2 \rightarrow -1^{sinal} * (b_n \beta^{e_2} + b_{n-1} \beta^{e_2-1} \dots + b_2 \beta^{e_2-(n+1)} + b_0 \beta^{e_2-n})$$

Sendo  $e_1$  o expoente de Op1 e  $e_2$  o expoente de Op2, verifica-se que só se pode proceder uma operação de soma ou subtração entre Op1 e Op2 se  $e_1=e_2$ , uma vez que só nesta condição os seus *bits* estarão perfeitamente alinhados entre si.

É fácil perceber que, se  $e_1 > e_2$  teremos os pesos associados aos bits de Op1 maior que os seus correspondentes em Op2, exigindo desta forma que se proceda um realinhamento nos *bits* de um dos significandos antes de poder operá-los. De maneira análoga, o mesmo ocorre se  $e_2 > e_1$ .

A fim de garantir maior precisão nos resultados, deve-se sempre realinhar o significando do operando com o menor expoente em função do significando do operando com maior expente, de tal forma a poder expressar ambos em função do maior expoente.

O realinhamento dos bits se faz através de deslocamentos sucessivos à direita dos bits do significando a ser realinhado, conforme demonstrado na Figura 2.18. Desloca-se desta forma o significando que está sendo realinhado até que a posição do seu ponto separador coincida com a posição do ponto separador do significando que não será alterado.

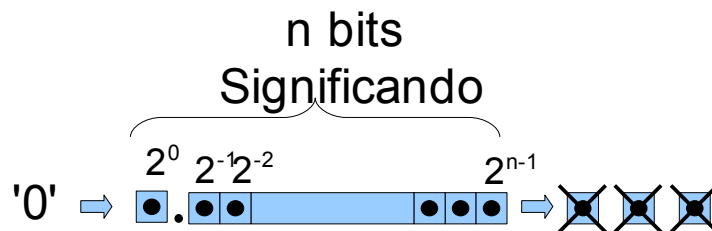


Figura 2.18: Deslocamento do Significando

A fim de manter o valor originalmente representado pelo significando que está sendo deslocado, uma vez que a operação de deslocamento de um número à direita equivale a dividi-lo pela base numérica adotada, a cada deslocamento deve-se incrementar o valor do expoente associado.

Após este procedimento, teremos os dois operandos expressos em função de um mesmo expoente, e por conseguinte, os pontos separadores e todos os bits realinhados, podendo-se operá-los normalmente.

Apenas como um exemplo prático, consideremos duas operações de soma entre os operandos Op1, Op2, Op3 e Op4, respectivamente 12,5 , 4,5 , 14,5 e 4,75 , representados no formato de ponto flutuante hipotético dado na Tabela 10 a seguir:

**Tabela 10: Formato de Ponto Flutuante com 9 bits, formulado apenas para fins didáticos.**

<b>Formato de Ponto Flutuante Binário (<math>-1^s \cdot p.ppppp \cdot 2^e</math>)</b>		
<b>Campo</b>	<b>Valor</b>	<b>Observações</b>
Sinal (s)	1 bit	Tamanho em bits
Significando (p)	6 bits	5 bits armazenados + 1 bit implícito
Expoente (e)	3 bits	Tamanho em bits
Expoente Máximo ( $e_{\max}$ )	3	Representado pelo binário 110
Expoente Mínimo ( $e_{\min}$ )	-2	Representado pelo binário 001
Zero de Referência (Bias)	3	Representado pelo binário 011

As Figuras 2.20, 2.19, 2.21 e 2.22 a seguir, demonstram o processo de conversão e a representação dos operandos Op1, Op2, Op3 e Op4 para o formato de Ponto Flutuante proposto.

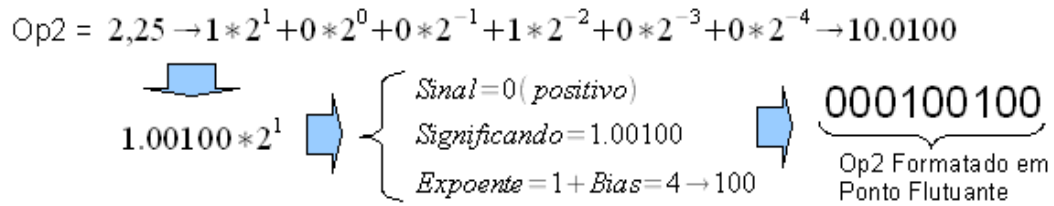


Figura 2.19: Representação esquemática da conversão de Op2 para o formato de Ponto Flutuante proposto

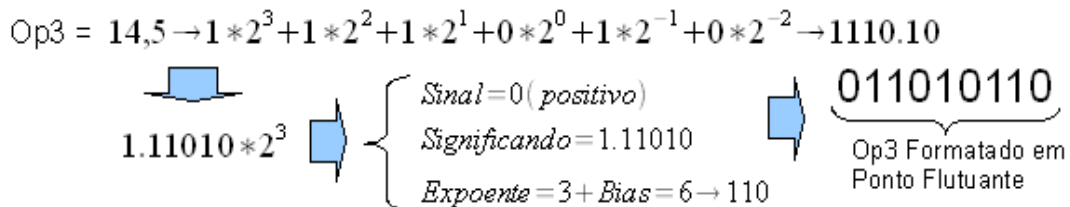


Figura 2.21: Representação esquemática da conversão de Op3 para o formato de Ponto Flutuante proposto

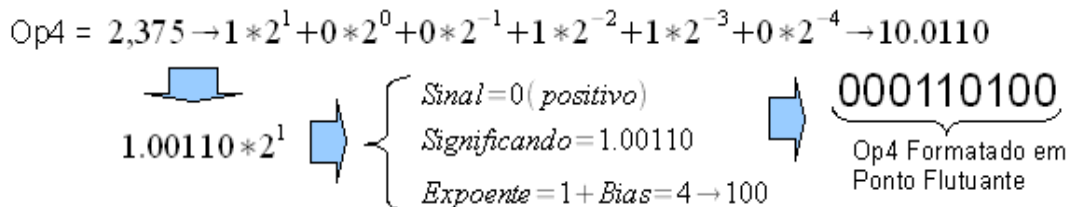


Figura 2.22: Representação esquemática da conversão de Op4 para o formato de Ponto Flutuante proposto

Considere primeiramente a operação de soma entre os operandos Op1 e Op2. Esta operação pode ser feita de duas maneiras, com e sem o realinhamento do significando, conforme a Figura 2.23 a seguir.

Na Figura 2.23(a) temos a operação sendo efetuada sem o realinhamento do significando. Como se pode observar, ao tentar-se efetuar a operação sem proceder o realinhamento, acaba-se por se obter não apenas um significando resultante incorreto, mas também o expoente associado fica indefinido.

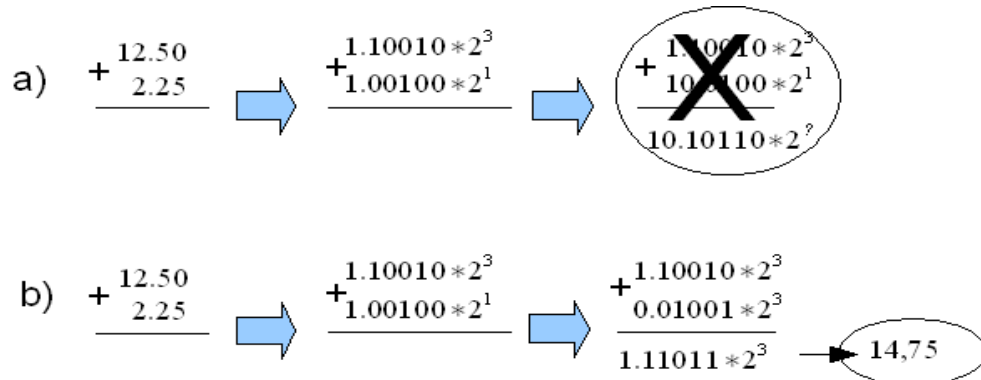


Figura 2.23: Diagrama esquemático da operação de soma entre Op1 e Op2

Na Figura 2.23(b) temos a operação sendo efetuada da maneira correta. Pelo diagrama apresentado, primeiramente os operandos são convertidos do formato de ponto flutuante adotado para a representação binária equivalente. Em seguida, efetua-se o realinhamento dos significandos com o devido reajuste do expoente do operando que teve o significando alterado. Em nosso exemplo, uma vez que Op1 possui o maior expoente, procede-se o realinhamento do significando de Op2 seguindo o esquema apresentado na Figura 2.18.

Após haver concluído o realinhamento, efetua-se a soma direta, bit a bit dos dois significandos.

O resultado obtido é considerado no padrão IEEE 754 como infinitamente preciso, ou seja, está expresso na maior precisão alcançável para o formato de ponto flutuante adotado para os operandos.

Como o resultado obtido já está normalizado, e não necessita ser

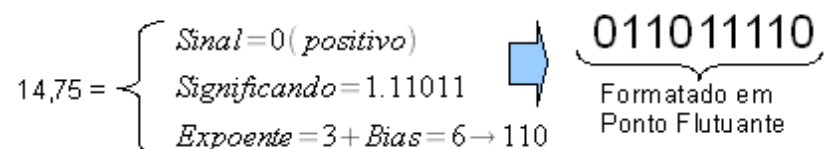


Figura 2.24: Representação do resultado da operação entre Op1 e Op2 no formato de Ponto Flutuante proposto

Considere agora a operação de soma entre Op3 e Op4, a qual está demonstrada esquematicamente na Figura 2.25.

$$\begin{array}{c}
 +14,500 \\
 \hline
 2,375
 \end{array}
 \rightarrow
 \begin{array}{c}
 +1.11010 * 2^3 \\
 \hline
 1.00110 * 2^1
 \end{array}
 \rightarrow
 \begin{array}{c}
 +1.11010 * 2^3 \\
 +0.01001 * 2^3 \\
 \hline
 10.00011 * 2^3
 \end{array}
 \rightarrow 16,75$$

Figura 2.25: Representação esquemática da operação de adição entre Op3 e Op4

De maneira semelhante ao adotado na operação anterior, antes de serem realinhados, os operandos são primeiramente convertidos do formato de ponto flutuante para o formato binário.

Como se pode observar, devido ao descarte dos bits menos significativos, o que normalmente ocorre durante o processo de realinhamento do significando, após a operação de realinhamento o significando do operando Op4 passou de 2,375 para 2,25. A expressão a seguir demonstra o que ocorreu com a representação de Op4.

$$(2,375_{10})10.01110_2 \rightarrow (2,25_{10})00010.01_2$$

Por fim, estando os bits dos significando já realinhados, pode-se efetuar a operação de soma dos significandos, obtendo como resultado um significando dito infinitamente preciso.

Nesta operação, diferentemente da anterior, obteve-se um resultado desnormalizado, o qual precisará ser normalizado e o arredondado antes que se possa obter o resultado final da operação.

O procedimento de normalização se dará conforme descrito na Sessão 2.3.6 desta dissertação. Na Figura 2.26 a seguir temos o resultado obtido após a normalização.

$$\begin{array}{c}
 10.00011 * 2^3 \\
 \text{Normalização} \rightarrow \boxed{\phantom{00010.01}} \\
 1.000011 * 2^4
 \end{array}$$

Figura 2.26: Operação de Normalização sobre resultado da adição de Op3 e Op4

Por fim, aplica-se o algoritmo de arredondamento para o mais próximo ou par ao significando já normalizado, conforme descrito na Seção 2.3.3 desta dissertação, obtendo o significando final da operação. Para o formato de ponto flutuante que estamos adotando neste exemplo, primeiramente reserva-se os 6 *bits* mais significativos para formar o significando pré-arredondado, ficando o *bit* menos significativo do significando original exercendo a função de *Guard bit*. Neste caso o *Guard bit* será igual a um, e, como não restam mais *bits* para formarem o *Sticky bit*, assume-se que o *Sticky bit* é igual a zero. O LSB neste exemplo é igual a um. A Figura 2.27 a seguir demonstra como ficam distribuídos os bits do significando original durante o processo de arredondamento.



Figura 2.27: Distribuição dos bits durante o processo de arredondamento

Por fim, aplicando-se os valores obtidos à Tabela 5, verifica-se que o *Round bit* deve ser feito igual a um. O resultado final do procedimento de arredondamento pode ser visto na Figura 2.28 a seguir.

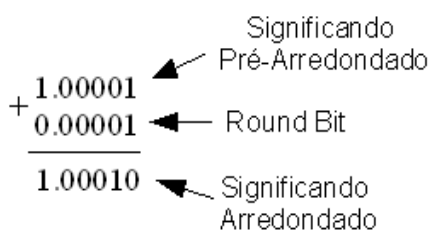


Figura 2.28: Resultado do processo de arredondamento

A princípio, poderíamos considerar as operações de normalização e arredondamento concluídas, tendo como resultado o que consta na Figura 2.29 a seguir:

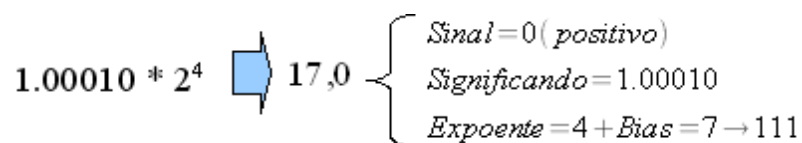


Figura 2.29: Formatação do resultado obtido

Entretanto, ao compararmos o valor do expoente obtido com o formato de ponto flutuante proposto, conforme indicado na Tabela 10, percebe-se que o mesmo é maior que o valor máximo permitido. Assim, sendo o resultado obtido maior que o maior número representável, o mesmo deve ser convertido para +Infinito. Desta forma o resultado final da operação de soma entre os operando Op3 e Op4, no formato de ponto flutuante adotado para este exemplo é igual a +Infinito, ou seja:

$$14.5 + 2.375 = +\infty$$

A fim de economizar recursos de hardware e ao mesmo tempo simplificar a implementação da operação de subtração, na implementação desta operação nesta dissertação, optou-se por converter os operandos do formato padrão de magnitude e sinal para o formato de complemento a dois. Desta forma podemos tratar as operações de soma e subtração indistintamente como operações de soma. Ao fim da operação, o resultado obtido é novamente convertido para o formato de magnitude e sinal.

A Figura 2.30 traz o fluxo de execução das operações de soma e subtração.



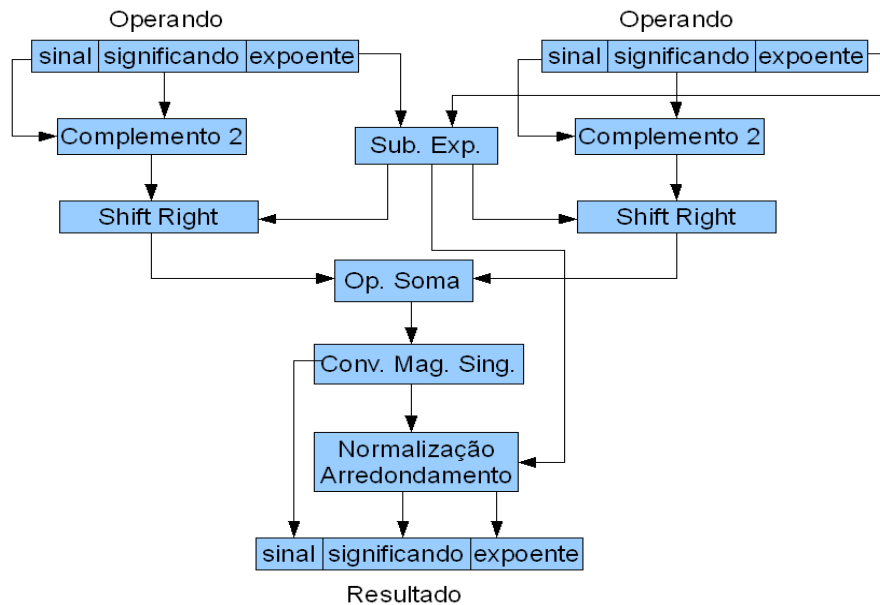


Figura 2.30: Fluxo de execução das operações de soma e subtração

### 2.3.7.2 Operação de Multiplicação

Diferentemente das operações de soma e subtração, a operação de multiplicação dispensa qualquer preparação prévia dos operandos.

O algoritmo de multiplicação pode ser resumido em:

1. somar os expoentes;
2. multiplicar os significandos;
3. normalizar e arredondar o resultado obtido, e, por fim;
4. calcular o sinal do resultado através da aplicação da função lógica “OU EXCLUSIVO” com os sinais dos operandos.

Dois detalhes importantes devem ser observados durante a execução das operações de multiplicação dos significandos e de soma dos expoentes.

1. Com relação à multiplicação dos significandos, deve-se observar que uma vez que a multiplicação de dois inteiros com  $n$  bits resulta em um inteiro  $2n$  bits e, considerando que o ponto separador divide o significando em duas partes distintas: uma representando a parte inteira, com 1 bit, e outra representando a parte fracionária, com  $n-1$  bits restantes, temos que o resultado da operação de multiplicação entre os significandos resultará em um significando com  $2n$  bits, com 2 bits representando a parte inteira e  $2n-2$  bits para a parte fracionária, conforme pode ser visto na Figura 2.31.

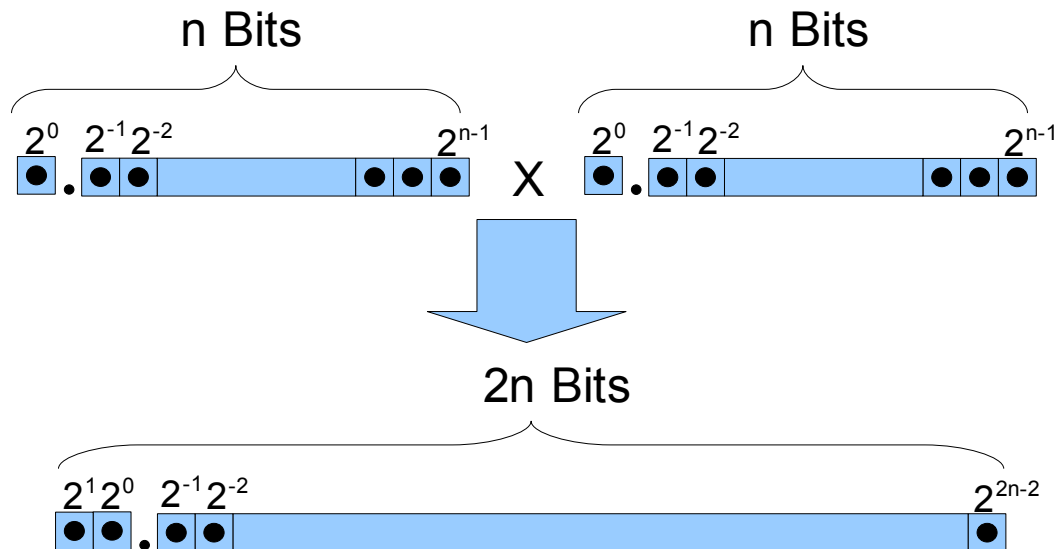


Figura 2.31: Distribuição dos bits como resultado de uma operação de multiplicação entre dois inteiros de tamanho  $n$

2. O outro detalhe diz respeito a necessidade do cancelamento do Bias, ou zero de referência, durante a operação de soma dos expoentes. Uma vez que o valor do expoente é formado a partir da soma do expoente desejado com o valor do Bias, deve-se evitar a soma duplicada do Bias durante a soma dos expoentes, subtraindo o seu valor do resultado final obtido.

A Figura 2.32, a seguir, demonstra o fluxo de execução das operações deste algoritmo.

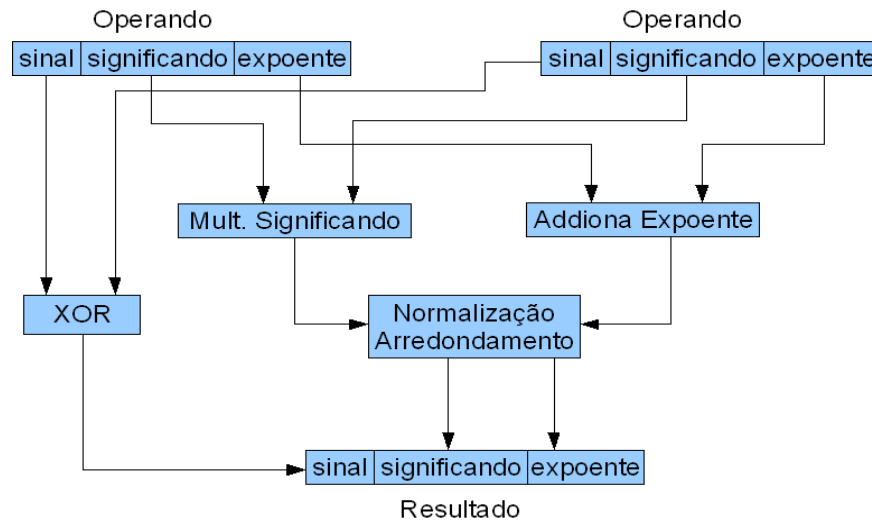


Figura 2.32: Fluxo de execução da operação de multiplicação

Apenas como um exemplo, considere a operação de multiplicação entre os operandos Op1 e Op2, respectivamente 1,65625 e 6,125, representados no formato de ponto flutuante apresentado na Tabela 10.

Neste formato, Op1 e Op2 seriam representados da seguinte forma:

$$\text{Op1} = 1,65625 \rightarrow 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 0 \cdot 2^{-4} + 1 \cdot 2^{-5}$$

$$\text{Op1} = \begin{cases} \text{Sinal} = 0 \text{ (positivo)} \\ \text{Significando} = 1.10101 \\ \text{Expoente} = 0 + \text{Bias} = 3 \rightarrow 011 \end{cases} \quad \Rightarrow \quad \underbrace{010101011}_{\text{Formatado em Ponto Flutuante}}$$

Figura 2.33: Representação esquemática da conversão de Op1 para o formato de Ponto Flutuante proposto.

$$\text{Op2} = 6,125 \rightarrow 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + 0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3}$$

$$\text{Op2} = \begin{cases} \text{Sinal} = 0 \text{ (positivo)} \\ \text{Significando} = 1.10001 \\ \text{Expoente} = 2 + \text{Bias} = 5 \rightarrow 101 \end{cases} \quad \Rightarrow \quad \underbrace{010001101}_{\text{Formatado em Ponto Flutuante}}$$

Figura 2.34: Representação esquemática da conversão de Op2 para o formato de Ponto Flutuante proposto.

Aplicando o algoritmo de multiplicação proposto tem-se:

$$\text{Multiplicando os Significandos} = 1.10101 * 1.10001 = 10.1000100101$$

$$\text{Somando os Expoentes} = e_1 + e_2 - \text{Bias} = 3 + 5 - 3 = 2$$

$$\text{Operando os sinais} = (0 = \text{Positivo}) \text{ XOR } (0 = \text{Positivo}) = (0 = \text{Positivo})$$

Aplicando os resultados obtidos à expressão 10 da Seção 2.2.4.1, obtem-se:

$$2^3 + 2^1 + 2^{-3} + 2^{-6} + 2^{-8} = 10,14453125$$

Pelo padrão IEEE 754, este seria considerado o resultado infinitamente preciso para esta operação. Entretanto, este resultado não pode ser diretamente representado no formato de ponto flutuante adotado, devendo ser primeiramente normalizado e arredondado seguindo o que foi apresentado nas seções 2.3.6 e 2.3.5. Neste exemplo, aplicando-se o modo de arredondamento para o mais próximo ou par, após a normalização tem-se:

$$\text{Significando} = 10.1000100101 \rightarrow 1.01000100101$$

$$\text{Expoente} = 2 \rightarrow 3$$

Após a aplicação do algoritmo de arredondamento tem-se:

$$\text{Significando} = 1.01000100101 \rightarrow 1.01001$$

$$\text{Expoente} = 3 \rightarrow 3$$

Por fim, aplicando novamente os resultados obtidos à Expressão (13) da Seção 2.2.4.1, obtém-se:

$$2^3 + 2^1 + 2^{-2} = 10,25$$

Este é considerado o resultado mais preciso para esta operação nesta notação de ponto flutuante.

O resultado obtido pode então ser convertido para o formato de ponto flutuante adotado conforme disposto na Figura 2.35 a seguir:

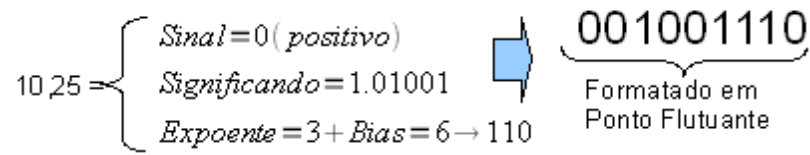


Figura 2.35: Representação esquemática da conversão do resultado para o formato de Ponto Flutuante proposto.



### 3 TRABALHOS RELACIONADOS

Existe uma vasta quantidade de trabalhos científicos associados ao tema dos co-processadores aritméticos e suas implementações. De um modo geral, estes trabalhos se dividem entre aqueles que descrevem o padrão IEEE 754, apresentando de forma sucinta o embasamento teórico necessário à boa compreensão do seu conteúdo, dos quais destacamos [14] e [26], e aqueles que descrevem os detalhes de implementação destes co-processadores como [27], [25], [24] e [28].

Devido ao foco do nosso trabalho, em nossa pesquisa bibliográfica demos maior atenção aos trabalhos relacionadas à implementações e a melhorias propostas para implementações segundo o padrão IEEE 754 em *FPGA*. Mais especificamente aos trabalhos relacionados às implementações das operações de soma, subtração e multiplicação e às técnicas de normalização e arredondamento.

Dentre os trabalhos pesquisados destacamos [27], [29], [25], [24] e [28], os quais são discutidos a seguir:

Em [29] é apresentada uma análise de diversas configurações de multiplicadores e somadores/subtratores com relação ao número de estágios de pipe-line, a área ocupada e o consumo de energia quando implementadas em *FPGA*.

Os autores relatam haver atingido frequências de trabalho em torno de 200 Mhz para implementações de operações de precisão dupla e 240 Mhz para operações de precisão simples com um *FPGA* Xilinx Virtex2Pro XC2VP125-7f1696.

Também é relatada a implementação de um multiplicador de matrizes a partir dos núcleos (cores) desenvolvidos, o qual atingiu um desempenho da ordem de 8 Gflops para operações de precisão dupla e 15 Gflops para operações de precisão simples.

Um dos diferenciais do trabalho apresentado em [29], é a análise apresentada da relação do número de estágios de pipe-line das arquiteturas do somador e do multiplicador implementados em função da frequência de trabalho, e também da métrica utilizada para avaliar o poder computacional dos *FPGAs*. Neste trabalho, a fim de enfatizar a grande vantagem que os *FPGAs* apresentam em relação ao consumo de energia, os autores propõem que o desempenho computacional de sistemas baseados em *FPGAs* devam ser medidos em função do número de operações de ponto flutuante executadas em relação ao volume de energia elétrica consumida, em Gflops/Watts. Nesta análise os autores concluem que as implementações em *FPGA* apresentaram um ganho de 6 vezes na relação Gflops/Watts quando comparadas ao consumo normalmente observado para a execução das mesmas operações em processadores de uso geral.

As Figuras 3.2 e 3.1 demonstram os resultados obtidos.

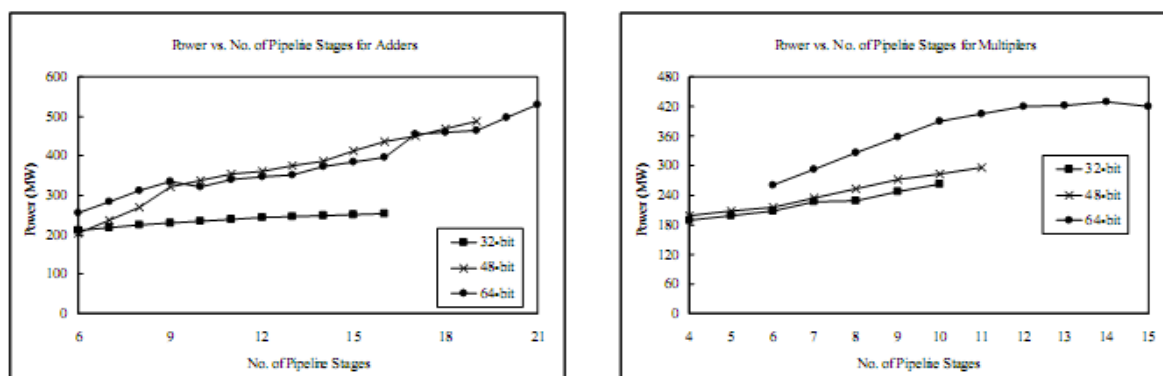


Figura 3.1: Relação do número de estágios do pipe-line pela potência consumida pelos módulos somadores e multiplicadores

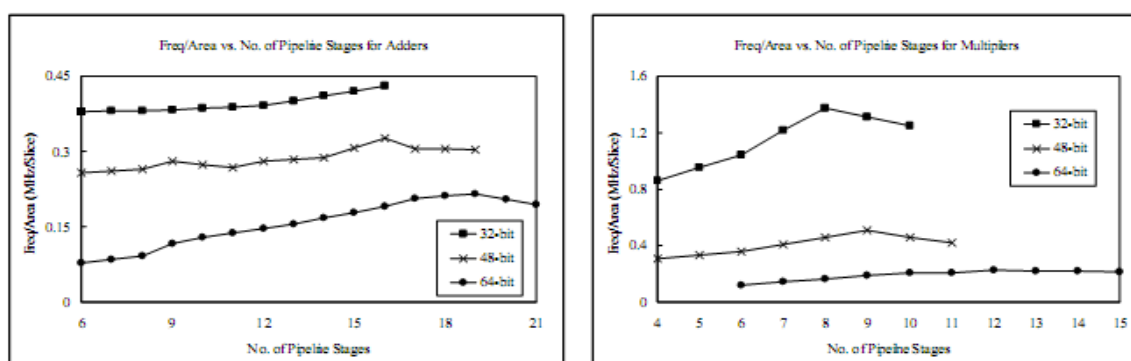


Figura 3.2: Relação do número de estágios do pipe-line pela frequência de trabalho dos módulos somadores e multiplicadores



Os autores em [29] concluem que arquiteturas de ponto flutuante baseadas em *FPGAs*, como a que foi utilizada por eles para efetuar a multiplicação de matrizes, além de poderem apresentar um melhor desempenho do ponto de vista computacional, quando comparadas aos processadores, podem ainda propiciar uma significativa economia de energia.

Em [27], é apresentada uma biblioteca completa contendo as cinco operações básicas para ponto flutuante determinadas pelo padrão IEEE 754 para números de precisão dupla, 64 bits, ou seja: soma, subtração, multiplicação, divisão e raiz quadrada. Os algoritmos utilizados são apresentados na forma de diagramas funcionais indicando os blocos internos implementados, acompanhados de uma descrição textual das suas funcionalidades. As Figuras 3.3 e 3.4 mais a frente trazem estes diagramas funcionais conforme apresentados pelos autores.

É apresentada também uma análise da relação entre o número de estágios de pipe-line pela frequência de trabalho e pela área ocupada para algumas versões dos módulos aritméticos quando implementadas em um *FPGA* Xilinx Virtex II Pro XC2VP100. As Figuras 3.5, 5.4, 3.7 e 3.8 a seguir trazem os resultados obtidos.

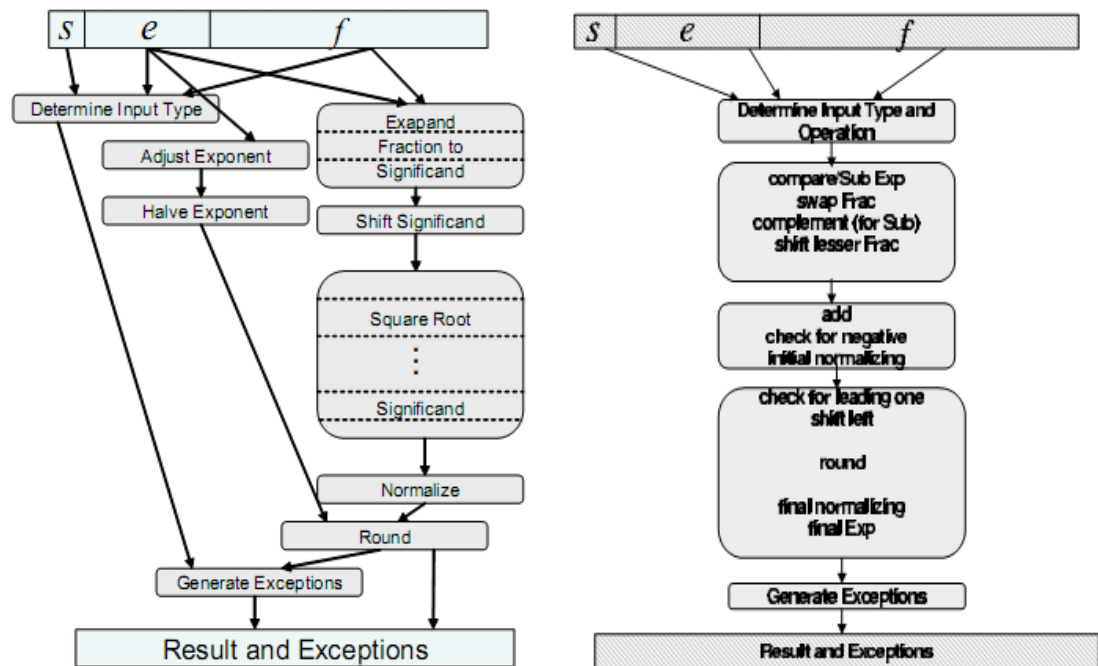


Figura 3.3: Diagrama interno funcional das operações de raiz quadrada e adição/subtração, respectivamente conforme propostos em [27]

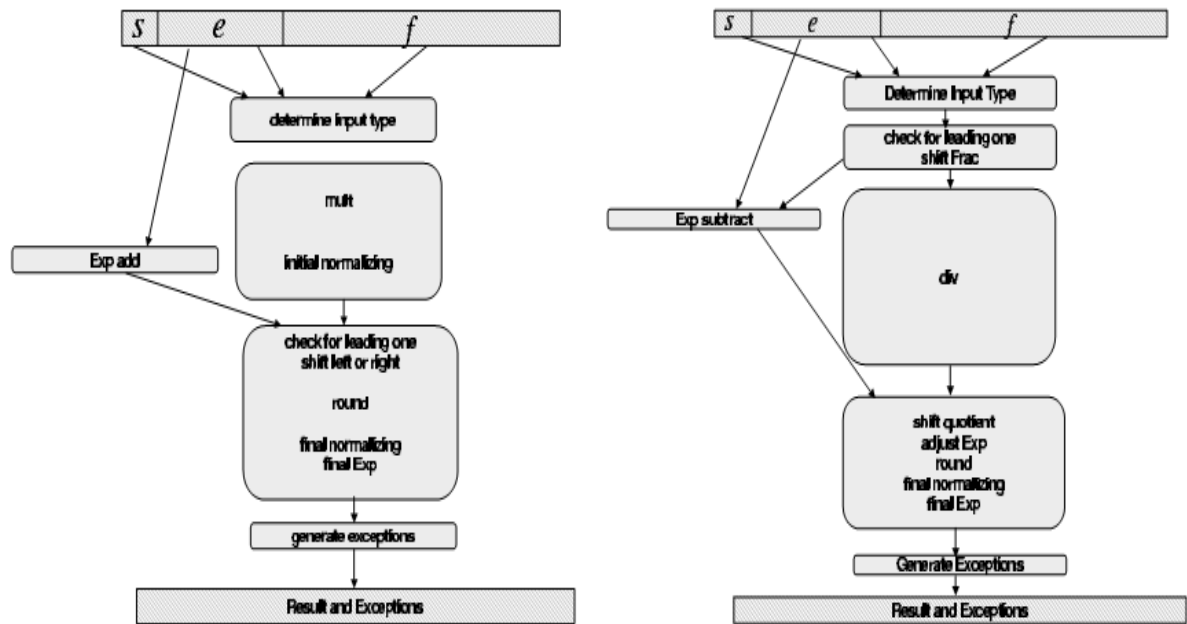


Figura 3.4: Diagrama interno funcional das operações de multiplicação e divisão, respectivamente conforme propostos em [27]

Configuration	No. of Stages	Maximum Frequency (MHz)	Area (slices)	% of FPGA's Total Area	Frequency/Area (MHz/1000 slices)
Most Compliant	19	170	2085	42	81
Common Features	12	170	1165	23	150
Lowest Overhead	11	170	935	19	181

Figura 3.5: Relação entre número de estágios do pipe-line pela frequência de trabalho e área ocupada para o módulo de multiplicação [27]

Configuration	No. of Stages	Maximum Frequency (MHz)	Area (slices)	% of FPGA's Total Area	Frequency/Area (MHz/1000 slices)
Most Compliant	17	170	1640	33	103
Common Features	17	170	1580	31	107
Lowest Overhead	14	170	1078	21	157

Figura 3.6: Relação entre número de estágios do pipe-line pela frequência de trabalho e área ocupada para o módulo de soma/subtração [27]

Configuration	No. of Stages	Maximum Frequency (MHz)	Area (slices)	% of FPGA's Total Area	Frequency/Area (MHz/1000 slices)
Most Compliant	60	164	2332	47	70.3
Common Features	57	164	1826	37	89.8
Lowest Overhead	55	169	1666	34	101.4

Figura 3.7: Relação entre número de estágios do pipe-line pela frequência de trabalho e área ocupada para o módulo de raiz quadrada [27]

Configuration	No. of Stages	Maximum Frequency (MHz)	Area (slices)	% of FPGA's Total Area	Frequency/Area (MHz/1000 slices)
Most Compliant (NRD)	68	140	4243	86	33
Common Features (NRD)	60	140	3625	73	38
Lowest Overhead (NRD)	58	140	3213	66	44
Lowest Overhead (SRT4)	32	90	3713	75	24

Figura 3.8: Relação entre número de estágios do pipe-line pela frequência de trabalho e área ocupada para o módulo de divisão [27]

Apesar de descrever os algoritmos de forma detalhada, os autores não apresentam detalhes internos de implementação tais como descrição dos estágios do pipe-line de cada módulo e nem apresentam nenhuma referência sobre tratamento de exceção.

Em [25] e [24] são apresentados dois extensos e detalhados trabalhos sobre os algoritmos de arredondamento para números em ponto flutuante segundo o padrão IEEE 754. Pode-se considerar [24] uma extensão do trabalho apresentado em [25], na qual os autores não só exploram mais detalhadamente o que foi proposto anteriormente como propõem melhorias aos algoritmos anteriormente propostos. As observações e modelos de implementações apresentados em ambos os trabalhos foram em grande parte aproveitados neste trabalho.

Em ambos os trabalhos os autores propõem a implementação do algoritmo denominado **Round to Nearest Up** como uma alternativa de baixo custo ao algoritmo de arredondamento para o mais próximo ou par definido no padrão IEEE 754, mais apropriada às implementações em software ou para quando os recursos de hardware forem escassos.

A implementação do algoritmo denominado **Round to Nearest Up**, conforme proposto por ambos os autores pretende substituir todo o algoritmo de arredondamento utilizado no modo de arredondamento para o mais próximo ou par, conforme proposto no padrão IEEE 754, pela adição de '1' ao valor armazenado no *Guard bit*, ou seja, caso o *Guard bit* já tenha sido valorado em '1' o *Overflow* desta soma resultaria na adição de '1' ao significando pré-arredondado.

Pela análise dos autores, ainda que podendo apresentar um erro de  $2^{\text{expoente} - 52}$  para alguns casos, esta ainda seria uma boa opção para sistemas não críticos, que suportassem erros desta magnitude, e onde os recursos de hardware fossem escassos.

Em [24] ainda são propostas melhorias nesta arquitetura, de tal forma a corrigir estas situações de falha.

Nesta dissertação foi dada preferência pela implementação completa do modo de arredondamento para o mais próximo ou par, conforme definido no padrão IEEE 754, a fim de garantir uma total conformidade com o que foi estabelecido pelo IEEE.

O trabalho apresentado em [28] é, no presente momento, até aonde pudemos verificar, o único que descreve em detalhes a implementação de um módulo multiplicador e somador integrado para ponto flutuante de precisão dupla, descrevendo inclusive os detalhes dos seus estágios de pipe-line. O foco principal desse trabalho é a implementação de uma arquitetura de multiplicação de matrizes densas otimizado para implementação em *FPGA*. Segundo os autores, o algoritmo proposto permite, potencialmente, um desempenho ótimo, por explorar a localidade e a reusabilidade dos dados. Foram implementados 39 unidades integradas de multiplicação e adição em um *FPGA* Xilinx Virtex II Pro XCV2P125, atingindo um desempenho de 15,6 GFlops com 1600 KB de memória local e 400 MB/s de velocidade de acesso à memória externa.

Um dos grandes diferenciais desse trabalho foi a descrição detalhada de cada um dos seus 12 estágios de pipe-line, bem como de toda a arquitetura de controle e acesso aos dados.

Os autores propõem a implementação de uma arquitetura mista de multiprocessamento formada por um processador de propósito geral associado à 39 processadores de propósito específico, formados a partir dos multiplicadores e acumuladores implementados, organizados em uma arquitetura tipo *Single Program Multiple Data (SPMD)*, de forma que o processador de propósito geral fornece os

dados e o controle necessários ao funcionamento dos processadores de propósito específico.

A Figura 3.9 a seguir traz um diagrama funcional do fluxo de dados e controle adotado na multiplicação de matrizes proposta por [28]. Segundo o esquema proposto cada um dos 39 processadores de propósito específico aqui referidos como PEs, fica responsável por efetuar o produto interno de uma linha e uma coluna das matrizes que estão sendo multiplicadas. Esta arquitetura tem a vantagem de não apenas explorar o paralelismo na execução das operações mas também de permitir o reuso eficiente dos dados, reduzindo desta forma o *throughput* de dados da memória.

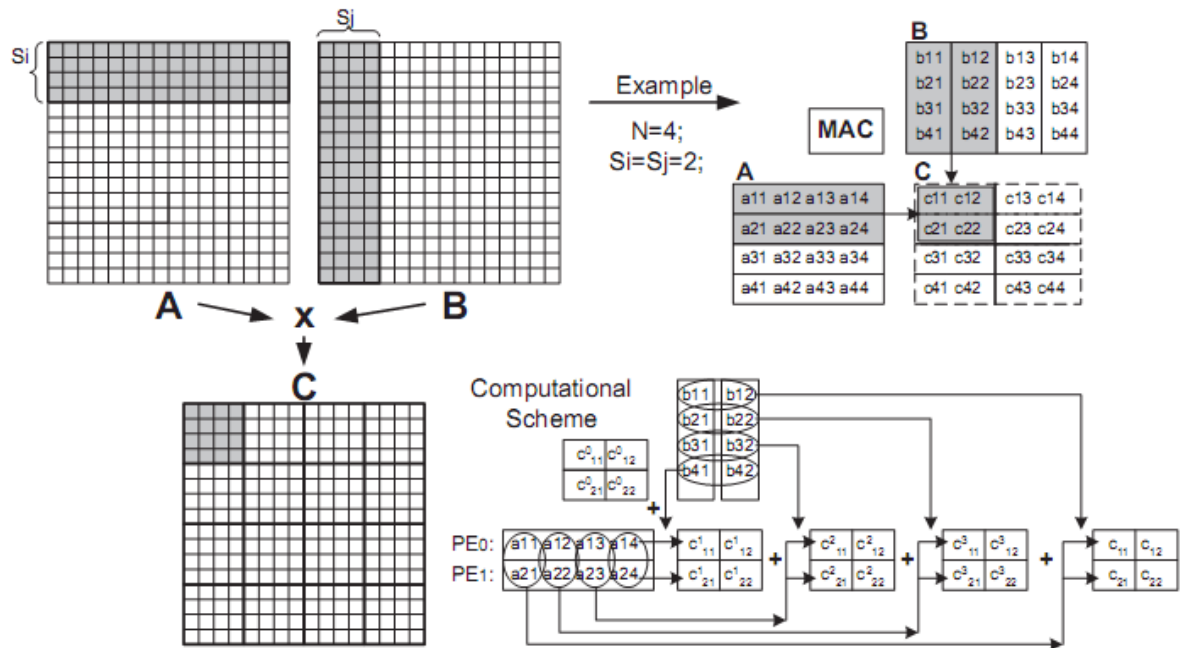


Figura 3.9: Esquema de controle proposto em [28] na arquitetura de multiplicação de matrizes

As Figuras 3.10 e 3.11 a seguir trazem maiores detalhes do esquema de controle e distribuição dos dados, bem como o detalhe da construção interna dos PE's.

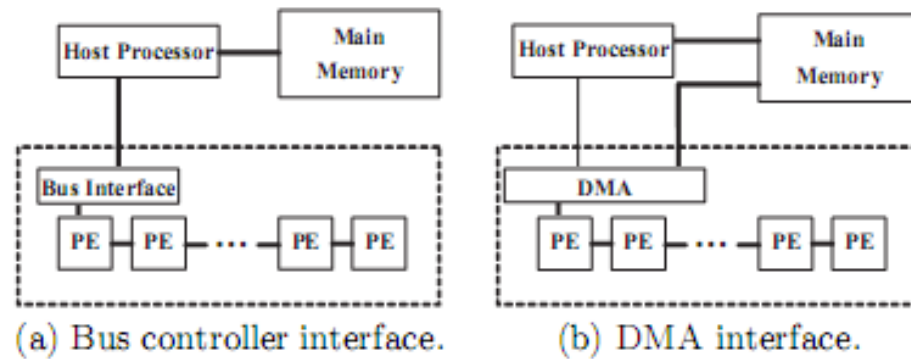


Figura 3.10: Arquitetura de controle e distribuição dos dados para implementações com e sem DMA conforme proposto em [28]

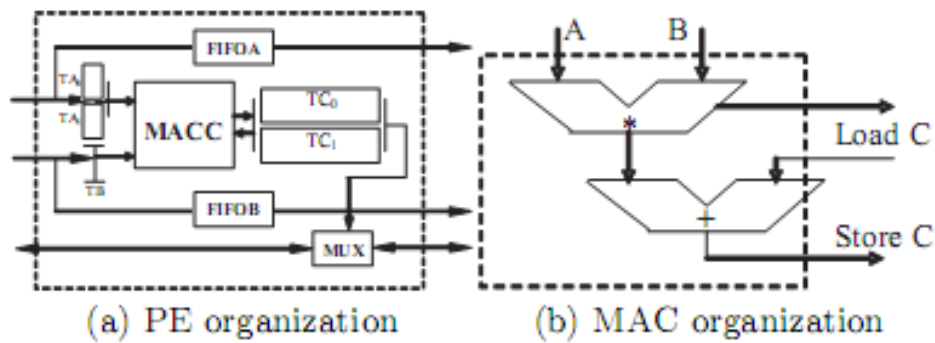


Figura 3.11: Organização interna dos PE's e a representação funcional dos Multiplicadores e Acumuladores conforme proposto em [28]

A Figura 3.12 a seguir traz o diagrama do pipe-line interno dos multiplicadores e acumuladores implementados.

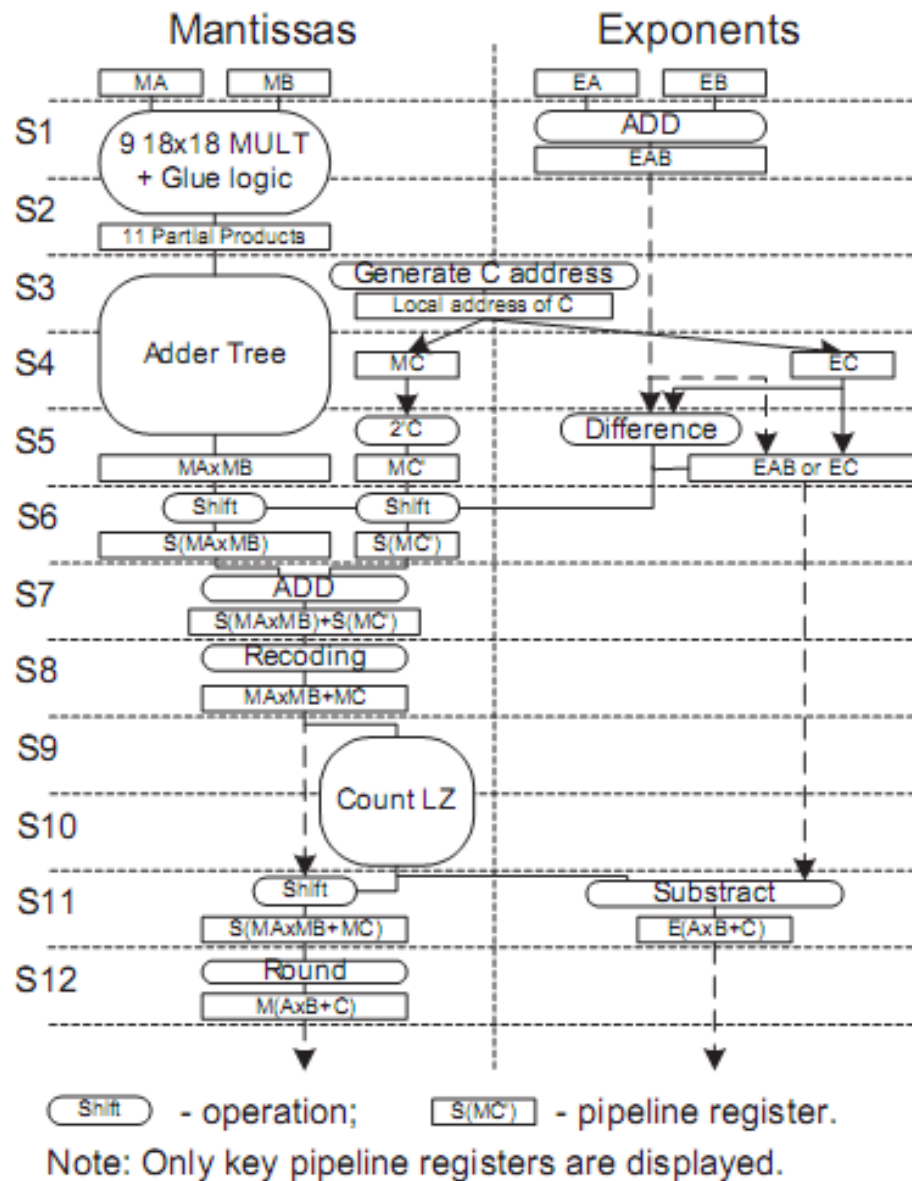


Figura 3.12: Pipe-line interno dos Multiplicadores e Acumuladores implementados em [28]

O pipe-line implementado obedece a seguinte ordem de operações:

- Entre o 1º e o 5º estágios do pipe-line processa-se a operação de multiplicação entre os operandos A e B, representados na Figura 3.12 pelas mantissas MA e MB e pelos expoentes EA e EB. Esta operação é feita multiplicando-se os significandos, identificados na Figura 3.12 como mantissas MA e MB, ao mesmo tempo em que se somam os expoentes. Um

detalhe importante no esquema adotado é que, a fim de reduzir o uso dos blocos multiplicadores existentes no *FPGA*, os *bits* de cada um dos significandos são divididos em 4 grupo de *bits*, sendo 3 com 17 bits e 1 com 2 bits, os quais são multiplicados através de 9 blocos multiplicadores e alguma lógica auxiliar, resultando em 11 produtos parciais. Caso não fosse adotada tal estratégia seria necessário utilizar 16 blocos multiplicadores para efetuar a multiplicação dos significandos.

- No terceiro estágio do pipe-line é gerado o endereço de busca do operando C.
- No quarto estágio do pipe-line o operando C é carregado. Da mesma forma que os operandos A e B, o operando C também é carregado através do seu significando, referido na Figura 3.12 como MC, e pelo seu expoente, referido na Figura 3.12 como EC.
- No quinto estágio do pipe-line é efetuado o complemento a dois do significando do operando C ao mesmo tempo em que é calculada a diferença entre o expoente do resultado da operação de multiplicação entre os operando A e B e o expoente do operando C. Com base na diferença calculada, identifica-se o maior entre os dois expoentes, o qual será propagado como expoente resultante da operação de soma entre o resultado da multiplicação entre os operandos A e B e o operando C. Também com base nesta diferença, é definido a quantidade de deslocamentos que serão necessários para o realinhamento dos significandos antes de proceder a operação de soma entre o operando resultante da multiplicação entre os significandos dos operandos A e B e o complemento a dois do significando do operando C.
- Entre o sexto e sétimo estágios do pipe-line são efetuados os deslocamentos necessários ao realinhamento dos significandos e a operação de soma dos significandos já realinhados.



- No oitavo estágio o significando resultante da operação de soma é novamente convertido para o formato de magnitude e sinal.
- Entre o nono e o décimo primeiro estágios do pipe-line é efetuada a normalização do significando já convertido para o formato de magnitude e sinal.
- No décimo segundo estágio é efetuado o arredondamento do significando já normalizado.

### 3.1 CONCLUSÕES

Embora os demais trabalhos científicos apresentados neste capítulo [25][24] [29][27] apresentem soluções, em todo ou em parte, para a implementação em hardware das operações de soma e multiplicação de números em ponto flutuante, verifica-se que apenas o trabalho apresentado em [28] traz uma solução completa, à semelhança do que se propõe nessa dissertação. Entretanto, verifica-se também que existem alguns pontos da arquitetura proposta em [28] que podem e necessitam ser melhorados. A seguir são apresentadas algumas sugestões de solução para os referidos pontos. Soluções estas que foram integralmente implementadas no trabalho objeto desta dissertação.

- A fim de garantir a compatibilidade com o padrão IEEE 754 no tocante à manipulação dos tipos especiais de dados definidos neste padrão, conforme descritos na Seção 3.3.1, à proposta original apresentada em [25] foi acrescentado mais um estágio de pipe-line, no qual identificam-se e classificam-se os tipos dos operandos. Desta forma, diferentemente do trabalho proposto em [28], o qual apresentava restrições com relação ao conteúdo dos operandos, na arquitetura proposta nesta dissertação pode-se manipular todos os tipos de dados definido no IEEE 754, incluindo NaN e Denormal.
- Ainda com relação a proposta original apresentada em [28], foi incluído um módulo responsável por identificar e controlar a ocorrência de exceções que possam ser causadas exclusivamente devido ao tipos de dados dos operandos. Este módulo trabalha em paralelo com o restante da arquitetura, analisando os tipos dos operandos que estão sendo manipulados por cada uma das operações. Com isto busca-se

identificar exceções tais como operações de soma entre  $+\infty$  e  $-\infty$ , as quais devem ser sinalizadas como NaN.

- A fim de identificar exceções causadas durante as operações internas, foi adicionada à arquitetura proposta por [28], no módulo responsável pela normalização e arredondamento, uma lógica auxiliar, responsável por monitorar e sinalizar a ocorrência de *underflow* ou *overflow* no expoente final obtido após os procedimentos de normalização e arredondamento. Esta lógica auxiliar faz com que tais ocorrências sejam reportadas convertendo-se o resultado final obtido para zero ou para infinito.
- Foi efetuada uma correção na arquitetura proposta em [28], a fim de eliminar uma falha grave ali identificada. Verificou-se que na arquitetura proposta não foi prevista a necessidade de uma operação de complemento a dois no significando resultante da operação de multiplicação entre os operandos A e B, que se encontra no formato de magnitude e sinal, o qual deverá ainda ser somado com o operando C que se encontra no formato de complemento a dois. Conforme pode-se verificar, trata-se de uma falha grave na referida arquitetura, uma vez que não se pode operar diretamente valores expressos no formato de magnitude e sinal com valores expressos no formato de complemento a dois. Como solução, foi incluída a lógica necessária para corrigir o referido problema.



## 4 UNIDADE DE MULTIPLICAÇÃO E ACUMULAÇÃO

Este capítulo descreve os requisitos funcionais e não funcionais, a arquitetura interna, os detalhes de implementação e a metodologia de desenvolvimento empregada na implementação da Unidade de Multiplicação e Acumulação.

A Unidade de Multiplicação e Acumulação, ou simplesmente *MAC*, é um co-processador aritmético de aplicação específica, implementado em linguagem de descrição de hardware, VHDL, capaz efetuar operações integradas de multiplicação e adição de números em ponto flutuante de precisão dupla formatados segundo o padrão IEEE 754.

Esta unidade foi concebida de forma a poder ser utilizada como base à construção de um co-processador aritmético, específico para multiplicação de matrizes densas.

#### 4.1 REQUISITOS FUNCIONAIS E NÃO FUNCIONAIS

**REQ1.** O *MAC* deverá dispor de 6 portas de entrada, 3 das quais com 64 bits, denominadas portas *Op.A*, *Op.B* e *Op.C*, através das quais se procede a carga dos operandos. As 3 portas restantes, com um bit cada, ficarão distribuídas uma porta para o sinal de *Reset*, uma para o sinal de *Clock* e a última para o sinal *Valid*, o qual será utilizado para indicar que conteúdo das portas *Op.A*, *Op.B* e *Op.C* são válidos.

**REQ2.** O *MAC* deverá dispor de 2 portas de saída, uma com 64 bits, denominada *Result*, a qual conterá o resultado da operação  $(A*B)+C$ , onde *A*, *B* e *C* são os valores carregados nas portas *Op.A*, *Op.B* e *Op.C*, e uma com um *bit*, denominada *Done*, através da qual o *MAC* indicará que o conteúdo da porta *Result* é válido, ou seja, foi operado a partir de operandos válidos.

**REQ3.** Tanto os operandos de entrada quanto o resultado fornecido pelo *MAC* deverão estar formatados em ponto flutuante de precisão dupla – 64 bits, formatados segundo o padrão IEEE 754

**REQ4.** O *MAC* deverá ser capaz de operar com os números *Zero*, *+Infinito*, *-Infinito* e *NaN*, conforme definidos pelo padrão IEEE 754.

**REQ5.** O *MAC* deverá aceitar números no formato *Denormal*, ainda que podendo convertê-lo para zero antes de operá-lo.

**REQ6.** O *MAC* deverá converter automaticamente qualquer dos valores de entrada que estiver formatado como *Denormal* para zero antes de operá-lo.

**REQ7.** O *MAC* deverá sinalizar corretamente a ocorrência de *+Infinito*, *-Infinito* e *NaN* durante qualquer das suas operações internas, utilizando para tanto os valores definidos no padrão IEEE 754.

**REQ8.** O *MAC* deverá ser capaz de executar uma operação integrada de multiplicação e soma sobre seus operandos por ciclo de relógio;

**REQ9.** É desejável que o *MAC* possa operar até à uma frequência de 200 Mhz;

**REQ10.** O *MAC* deverá atender em suas portas de entrada e saída a sinalização proposta na Figura 4.1 a seguir. Esta figura mostra a relação entre sinais de dados e controle do *MAC*.

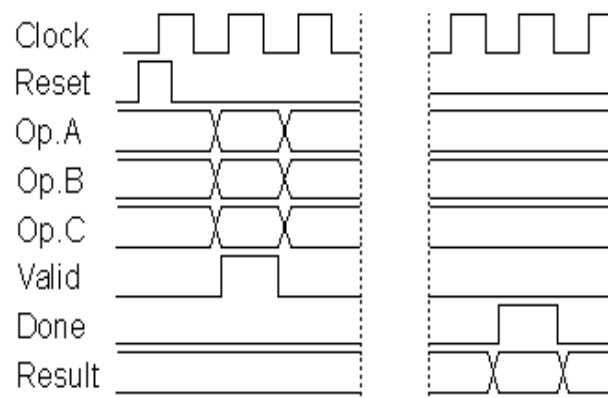


Figura 4.1: Diagrama de tempo dos sinais do MAC

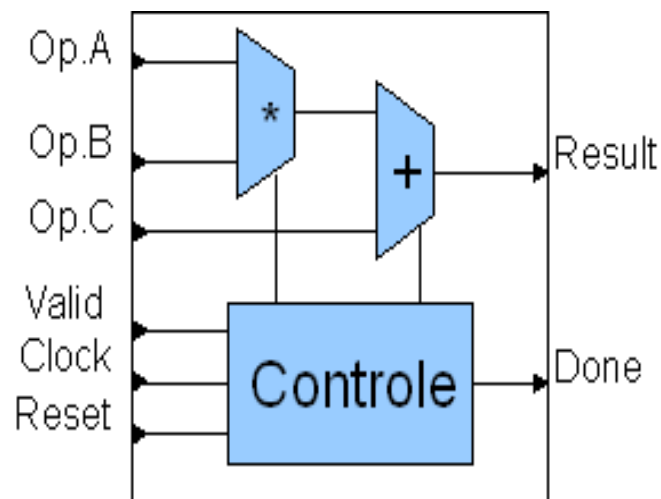


Figura 4.2: Diagrama esquemático funcional do MAC

## 4.2 METODOLOGIA DE DESENVOLVIMENTO

Neste projeto, devido à sua complexidade, foi adotado a metodologia de desenvolvimento incremental. A partir dos requisitos inicialmente elicitados, foi implementado um modelo de referência em linguagem de alto nível, linguagem C, o qual serviu de base ao desenvolvimento dos módulos de hardware.

Durante o processo de construção do modelo de referência, primeiramente foi implementado um modelo baseado nas operações de multiplicação e soma da biblioteca padrão da linguagem C, ao qual denominamos de Modelo Canônico. Este modelo inicial foi então refinado, tendo suas operações aritméticas gradativamente substituídas por funções de mesma funcionalidade, mas que efetuavam a manipulação direta dos vetores binários obtidos a partir do modelo canônico, dando assim origem ao modelo de referência de alto nível.

A partir do modelo de referência foram gerados diversos arquivos texto contendo vetores binários, os quais serviram tanto como entrada, quanto como elementos de verificação durante o processo desenvolvimento dos módulos de hardware. A Figura 4.3 a seguir ilustra o fluxo de projeto adotado.

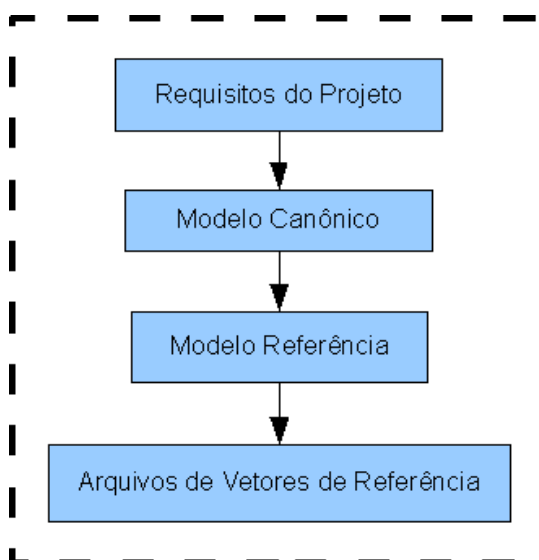


Figura 4.3: Fluxo de projeto do modelo de referência



A construção do modelo canônico teve como objetivo simplificar o processo de desenvolvimento e garantir a total confiabilidade dos resultados obtidos com o modelo de referência.

Os resultados obtidos com o modelo canônico, bem como, os valores de cobertura de teste dos módulos de hardware, foram convertidos das representações de ponto flutuante para representações de vetores binários, servindo posteriormente como entrada e padrão de comparação para o modelo de referência.

A partir dos vetores binários obtidos, pode-se modelar no modelo de referência todas as operações internas que deveriam ser efetuadas nas diversas fases de cada operação aritmética. Os resultados obtidos por estas operações foram então confrontados com os resultados já obtidos a partir do modelo canônico.

Para a verificação final deste segundo modelo, agora totalmente implementado na forma de funções discretas, foi implementado um aplicativo de teste, através do qual foram efetuados testes com valores randômicos, valores de uso e de pior caso. Os resultados obtidos foram verificados automaticamente, tendo sido atingido uma taxa de 100% de acertos em um universo de mais de 100.000 operações efetuadas. A Figura 2.12 mostra o fluxo utilizado para a verificação e testes do modelo de referência.

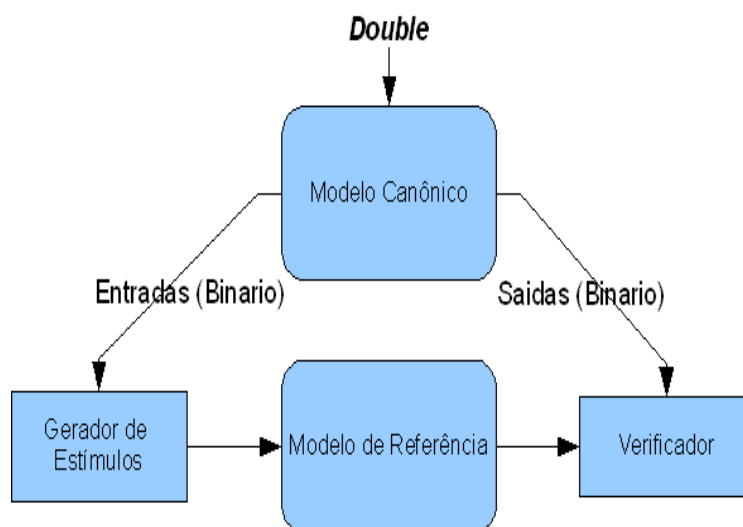


Figura 4.4: Verificação e teste do modelo de referência

O projeto do modelo de referência como um todo foi dividido em 5 grandes módulos, os quais foram implementados na forma de funções. Estas funções receberam suas entradas ou diretamente dos vetores binários gerados pelo modelo de referência ou dos vetores binários gerados internamente por outros módulos/funções. Foram implementados os seguintes módulos:

- Multiplicador parcial
- Árvore de soma
- Somador
- Normalização
- Arredondamento
- Devido à limitações da biblioteca da linguagem C utilizada, foi necessário implementar também diversas funções auxiliares para se poder manipular com números com mais de 64 bits.

Após a validação completa do modelo de referência, passou-se a geração dos vetores binários que serviram de referência para a construção dos módulos a serem implementados em hardware. A Figura 4.5, mostra a hierarquia de conexão dos módulos implementados e os pontos a partir dos quais foram gerados os vetores de referência. Estes vetores foram armazenados na forma de arquivos texto, para uso posterior.

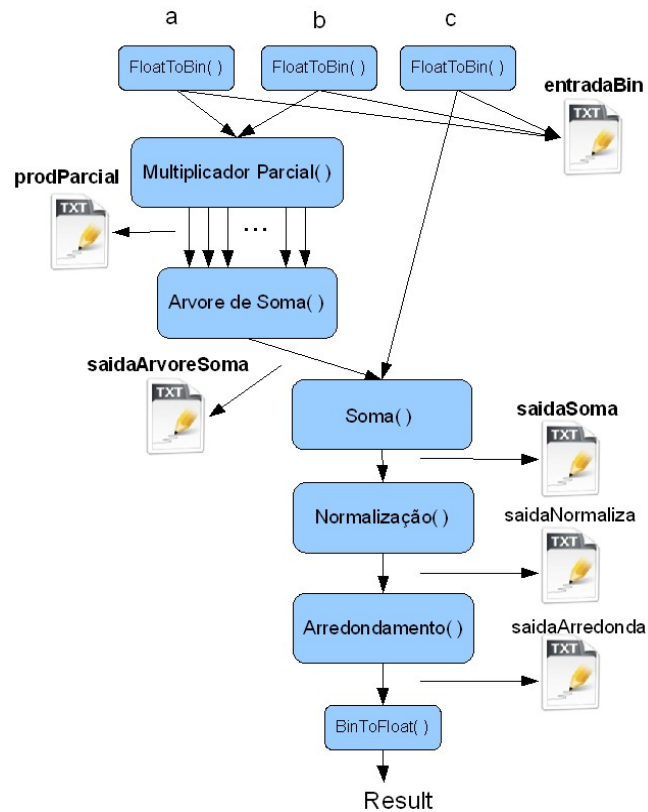


Figura 4.5: Geração dos arquivos de referência

Tanto o modelo de referência quanto o modelo canônico foram inteiramente desenvolvidos em linguagem C, utilizando a ferramenta *Eclipse Europa* [30] conjuntamente com o compilador *Mingw* [31] versão 5.1.3. A Figura 4.6 traz uma visão da tela da ferramenta Eclipse.

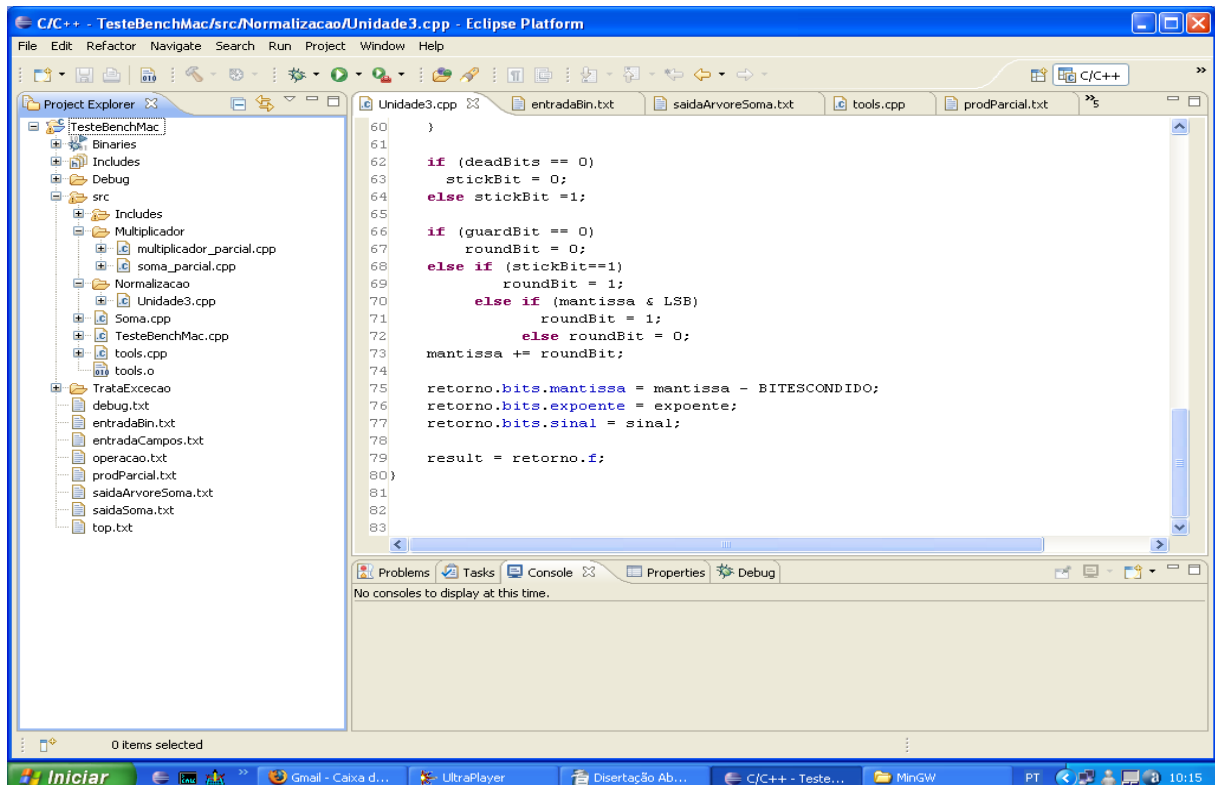


Figura 4.6: Ambiente de desenvolvimento de software utilizado - Eclipse + Mingw

A partir do modelo de referência testado e validado foi definida a arquitetura dos módulos de hardware a serem implementados no sistema.

O desenvolvimento dos módulos de hardware seguiu o fluxo de projeto demonstrado na Figura 4.7 a seguir:

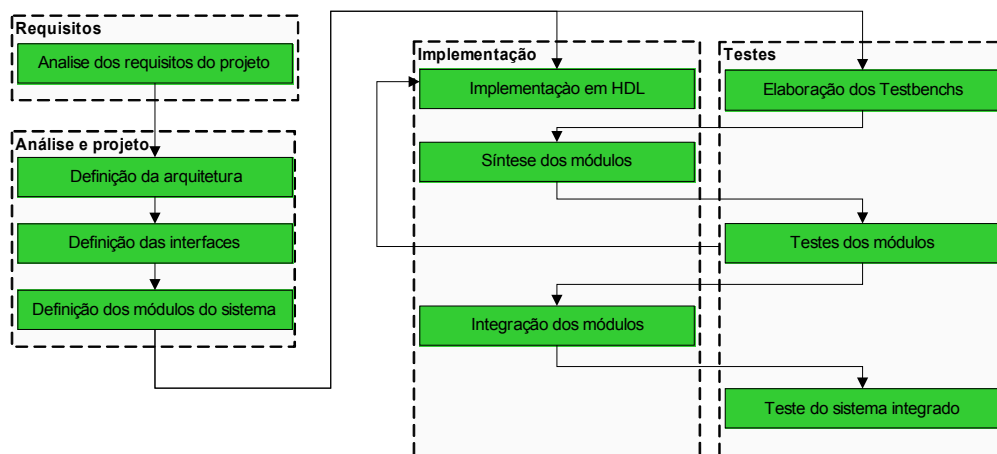


Figura 4.7: Fluxo de projeto dos módulos de hardware

Primeiramente foram elicitados os requisitos funcionais e não funcionais do multiplicador e acumulador. Em seguida, baseado nos resultados obtidos durante a implementação do modelo de referência, procedeu-se a definição da arquitetura, dos módulos e suas interfaces. Por fim, foram implementados e verificados os diversos módulos do projeto. Todos os módulos foram implementados na linguagem de descrição de hardware VHDL.

Durante o processo de desenvolvimento e validação dos módulos de hardware foram utilizadas as ferramentas *Modelsim XE III 6.2* [32] e *Xilinx ISE 9.2i* [33].

A ferramenta *Modelsim* dispõe de um ambiente completo para o desenvolvimento, testes e validação dos módulos de hardware. A mesma pode ser utilizada inclusive para a construção de *testbenches* baseados em modelos descritos através de vetores armazenados em arquivos tipo texto. Esta característica foi aproveitada para a construção dos *testbenches* de cada módulo construído. A Figura 4.8 a seguir ilustra a estratégia utilizada neste projeto.

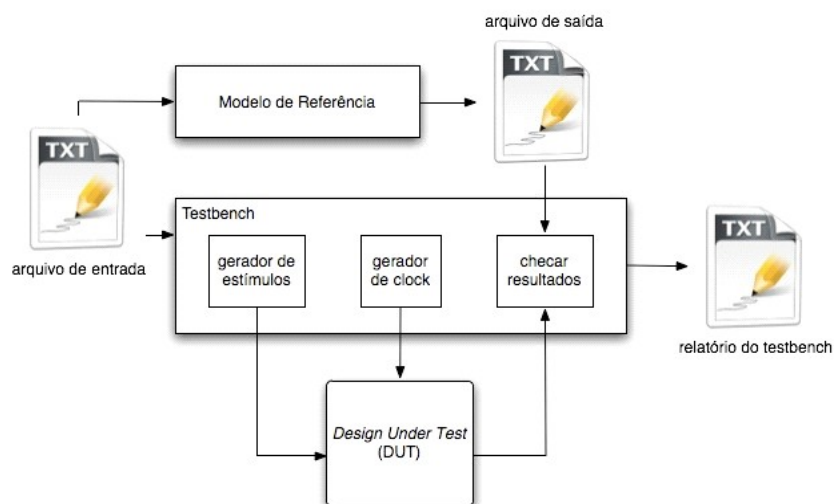


Figura 4.8: Processo de verificação e teste dos módulos de hardware

Neste projeto, a ferramenta *Modelsim* foi utilizada tanto como suporte para o desenvolvimento dos modelos funcionais de hardware quanto para a execução dos testes funcionais e de validação de cada módulo e do projeto como um todo. A Figura 4.9 mais a frente traz uma imagem da tela da ferramenta Modelsim.

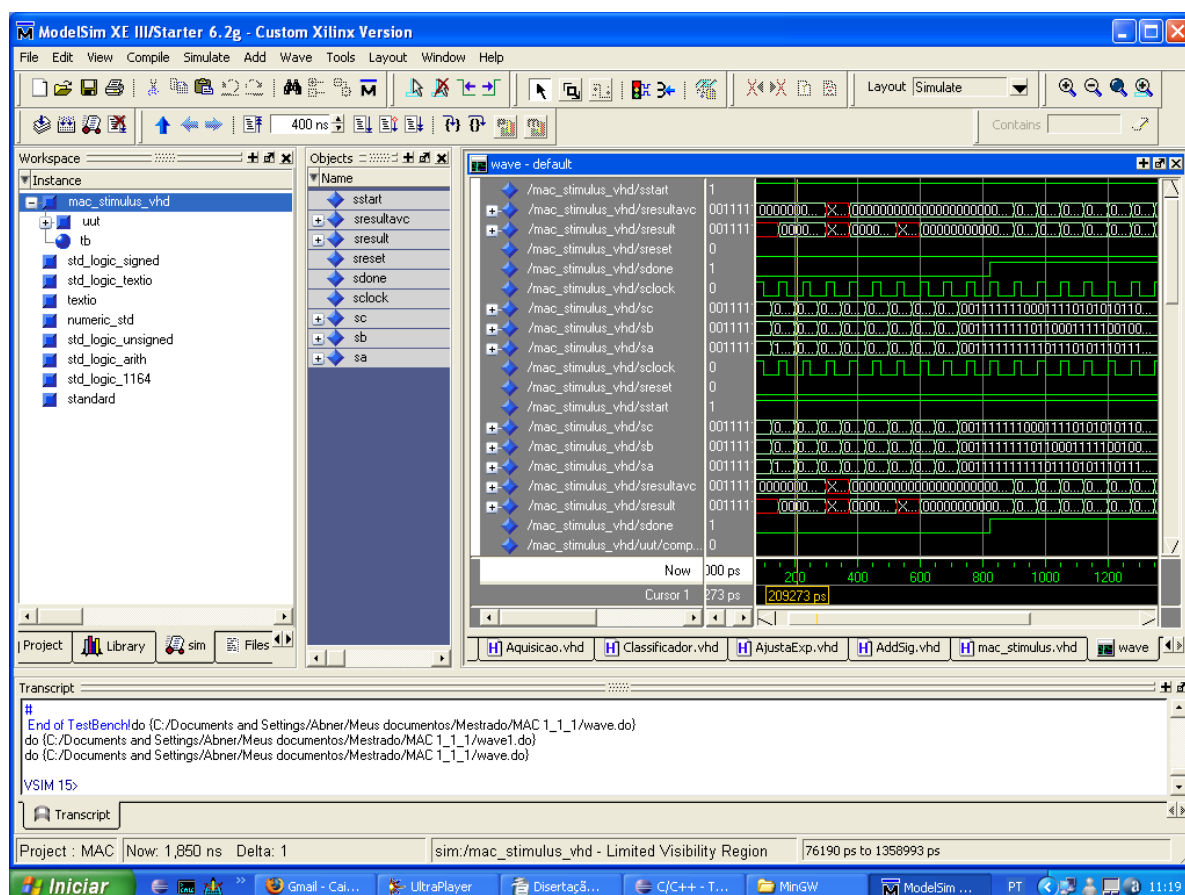


Figura 4.9: Ferramenta de desenvolvimento de hardware - Modelsim XE 6.2

Finalmente, após haver testado e validado funcionalmente os módulos de hardware desenvolvidos, utilizou-se a ferramenta *Xilinx ISE 9.2i* para se fazer a síntese lógica e mapeamento tecnológico dos módulos de hardware na tecnologia *FPGA* alvo. Como resultado, arquivos de configuração foram gerados(*bitstreams*), para a programação do *FPGA*.

Antes do processo de síntese, a ferramenta utilizada deve ser configurada com os parâmetros de projeto que são inteiramente dependente do dispositivo

*FPGA* ou do sistema onde este irá ser utilizado. Dentre estas características temos: estimativa da velocidade de trabalho do módulo de hardware, definição das funcionalidades dos pinos do *FPGA*, estimativa da área componente, quantidade e configuração dos módulos multiplicadores rápidos, blocos somadores, elementos de memória, etc.

Estas características normalmente fazem parte dos requisitos não funcionais do projeto, e devem ser levadas em conta durante a construção de cada um dos seus módulos.

Após esta configuração inicial, deve-se escolher o modo de síntese a ser utilizado. A ferramenta Xilinx ISE 9.2i permite a princípio três modos de síntese:

1. O primeiro e mais utilizado é o modo automático, no qual a ferramenta se encarrega de, a partir das especificações iniciais, definir automaticamente como os recursos disponíveis no *FPGA* serão utilizados.
2. O segundo é o modo semi-automático. Neste modo o projetista pode impor restrições ao uso de algum dos recursos disponíveis no *FPGA*. Este modo é muito utilizado quando se deseja aumentar o desempenho ou otimizar ao uso de algum dos recursos disponíveis.
3. O terceiro e último modo é o modo manual. Neste modo o projetista tem acesso às conexões internas do *FPGA*, definindo inclusive por onde e de que forma estas conexões serão estabelecida. Devida a complexidade dos *FPGAs*, este modo dificilmente é utilizado, ficando reservado à intervenções pontuais em projetos já sintetizados.

A Figura 4.10 a seguir traz uma imagem de uma tela da ferramenta ISE 9.2i.

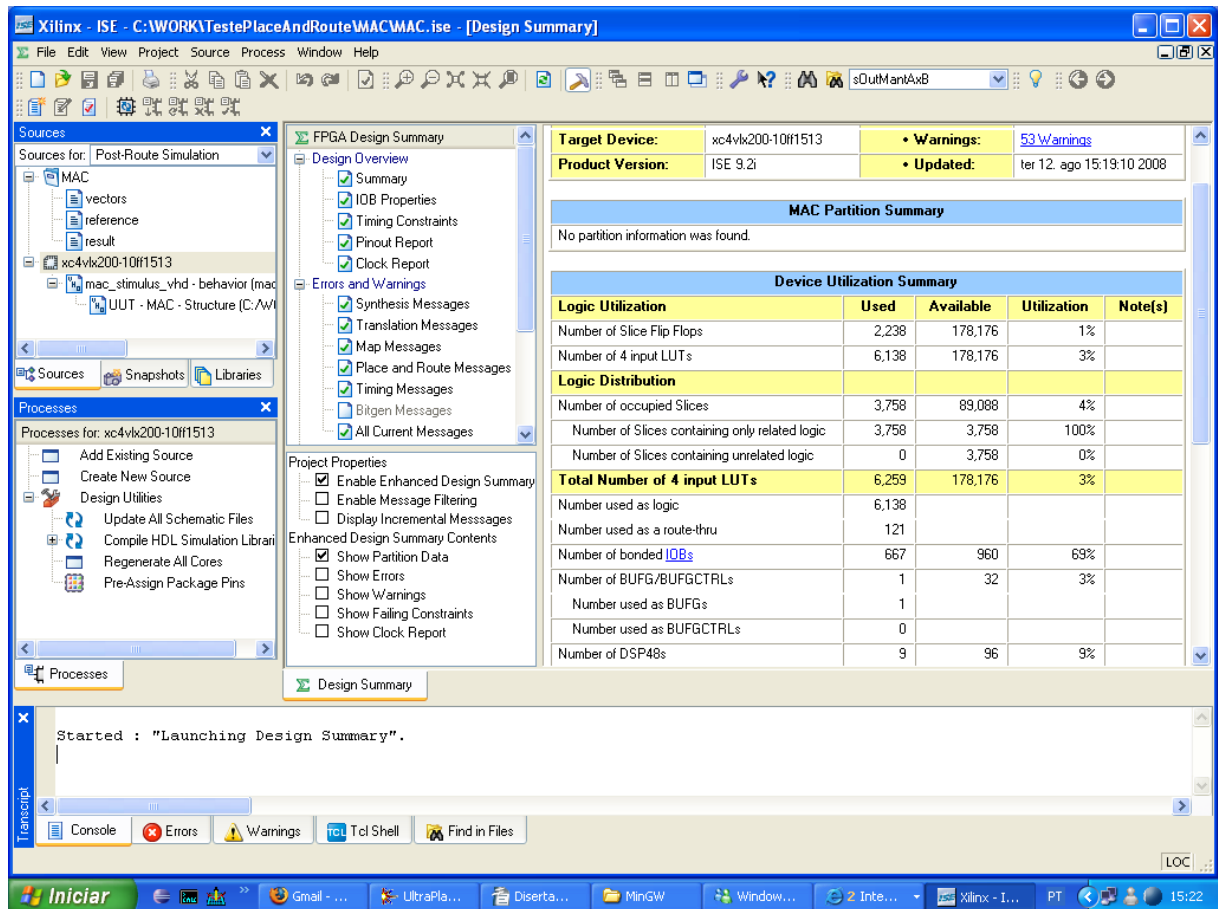
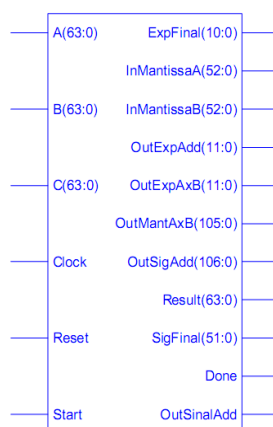


Figura 4.10: Ferramenta de desenvolvimento de hardware - Xilinx ISE 9.2i

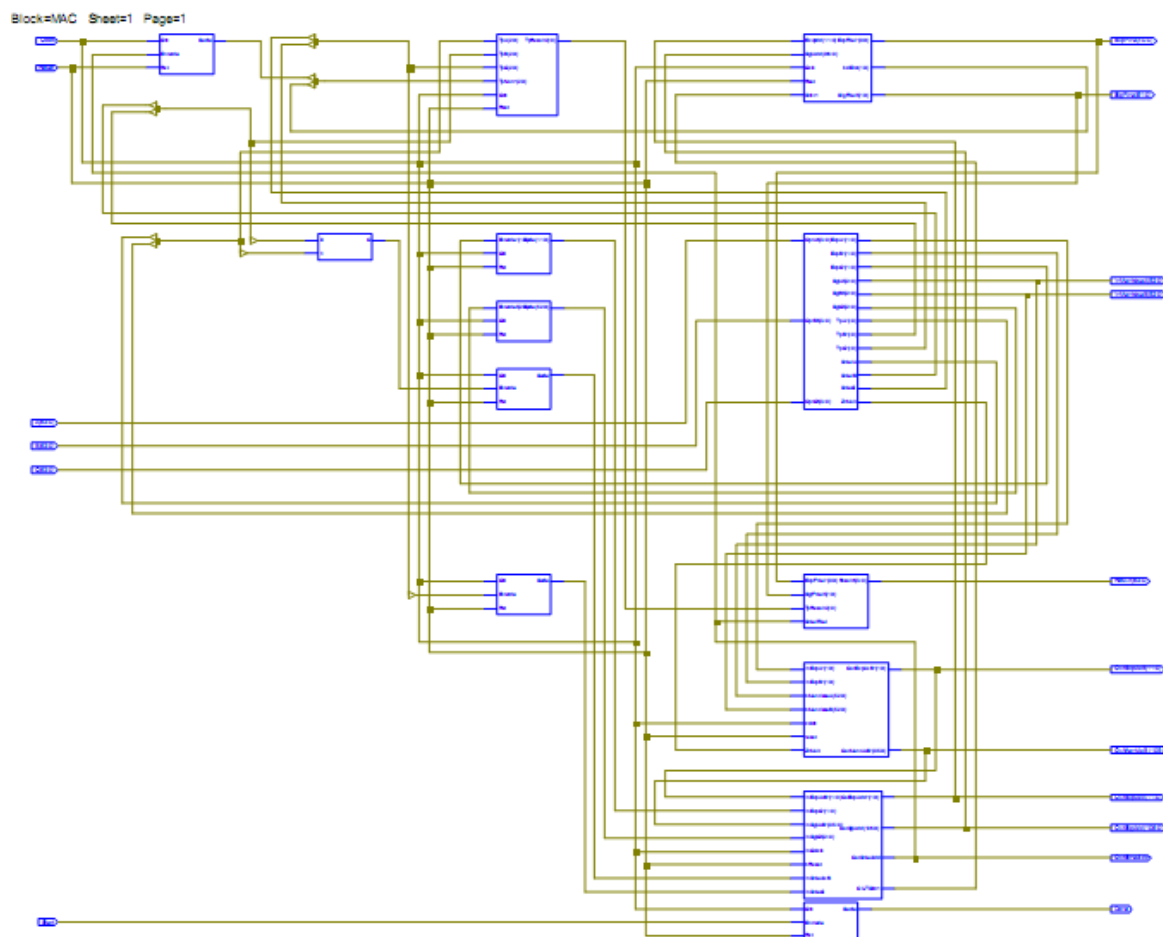
Os resultados obtidos bem como as características do dispositivo utilizado na síntese deste projeto estão disponíveis na Seção 5, a qual trata da implementação de um estudo de caso real.

As Figuras 4.11 e 4.22 a seguir trazem uma visão geral, simbólica, do projeto de hardware do multiplicador e do acumulador.





**Figura 4.11: Visão geral do multiplicador e acumulador implementado, identificando as suas portas de entrada e de saída**



**Figura 4.12: Visão dos módulos internos do multiplicador e acumulador**

### 4.3 ARQUITETURA INTERNA

A arquitetura interna do multiplicador e acumulador, *MAC*, aqui implementado, baseou-se em grande parte no trabalho apresentado em [28], conforme descrito no capítulo referente a Trabalhos Relacionados.

Esta arquitetura proposta tem a vantagem de prover a execução integrada das operações de soma e multiplicação, o que dispensa a necessidade de se efetuar a normalização do resultado da operação de multiplicação antes que se proceda a operação de soma. Todas os ajustes sugeridos no capítulo anterior para a melhora de desempenho de multiplicação de matrizes foram aqui implementadas.

$$\begin{array}{c}
 0,75 * 1,5 + 1,125 = 2,5 \\
 \swarrow \quad \downarrow \quad \searrow \quad \swarrow \\
 (1.1 * 2^{-1}) * (1.1 * 2^0) + (1.001 * 2^0) = (1.01 * 2^1) \\
 \underbrace{\hspace{10em}} \\
 (10.01 * 2^{-1}) + (1.001 * 2^0) = (1.01 * 2^1)
 \end{array}$$

**Figura 4.13: Exemplo de operação integrada das operações de multiplicação e soma sem a normalização do resultado intermediário**

A Figura 4.13 a seguir traz um exemplo de como ocorre a operação integrada de multiplicação e soma. Aqui pode-se perceber que o fato de o resultado da operação de multiplicação não estar normalizado não prejudica em nada o resultado final da operação integrada.

Neste trabalho estendeu-se o que foi proposto por [28] com a inclusão de uma unidade de controle de exceções, a qual busca identificar e tratar exceções que possam ocorrer devido ao conteúdo dos operandos ou ao resultado da operação. Neste contexto, são consideradas exceções operações que resultem em *NaN*, *+Infinito* ou em *-Infinito*.

Outra alteração importante que introduzimos na arquitetura proposta por [34] foi à inclusão de mais dois estágios no pipe-line. Assim, esta implementação do MAC ficou com 14 estágios de pipe-line. Esta alteração, apesar de ter aumentado a latência de resposta do MAC, permitiu uma melhora significativa no desempenho final do sistema, uma vez que pode-se construir módulos internos mais simples, os quais operam a uma frequência de trabalho mais alta.

Vale salientar que soluções como estas, com o aumento do pipe-line, só devem ser adotadas em sistemas que, a semelhança do aqui descrito, destinam-se ao processamento massivo de dados, em regime de fluxo contínuo, de tal forma que o prejuízo causado pelo aumento na latência de resposta do sistema não seja significativo no tempo total de processamento.

Em linhas gerais, o MAC pode ser descrito como sendo a associação de 6 grandes unidades funcionais, denominadas:

- Unidade de Aquisição
- Unidade de Multiplicação
- Unidade de Soma
- Unidade de Normalização e arredondamento
- Unidade de controle de exceções
- Unidade de formatação dos resultados

Na Figura 4.14 temos o diagrama dos módulos internos do MAC.

Cada unidade é formada por um ou mais módulos internos, podendo estar distribuída em um ou mais estágios de pipe-line.

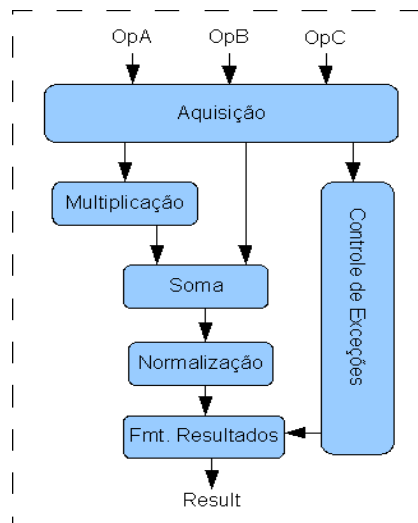


Figura 4.14: Diagrama de distribuição dos módulos internos do MAC

### 4.3.1 Unidade de Aquisição

A Unidade de Aquisição é responsável tanto pela recuperação dos valores numéricos representados nos vetores binários carregados nas portas Op.A, Op.B e Op.C, quanto pela classificação do tipo do dado representado, se este é um número válido ou a representação de Infinito, Denormal ou NaN.

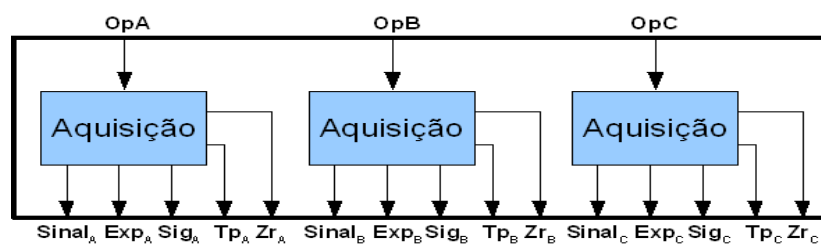


Figura 4.15: Diagrama interno da Unidade de Aquisição

Internamente a Unidade de Aquisição é formada por três blocos funcionais idênticos, completamente independentes, um para cada porta. A Figura 4.15 a seguir traz o diagrama interno da Unidade de Aquisição.

Como entrada a Unidade de Aquisição recebe o conteúdo das portas OpA, OpB e OpC.

Como saída esta unidade fornece o valor dos campos Sinal, Expoente e Significando de cada operando, além de indicar através do sinal  $Tp_A$ ,  $Tp_B$  e  $Tp_C$  o tipo de dado representado em cada operando. Também são fornecidos os sinais  $Zr_A$ ,  $Zr_B$  e  $Zr_C$  para indicar que o operando associado contém o valor zero.

A Figura 4.16 a seguir traz o diagrama esquemático interno simplificado dos blocos funcionais que formam a Unidade de Aquisição.

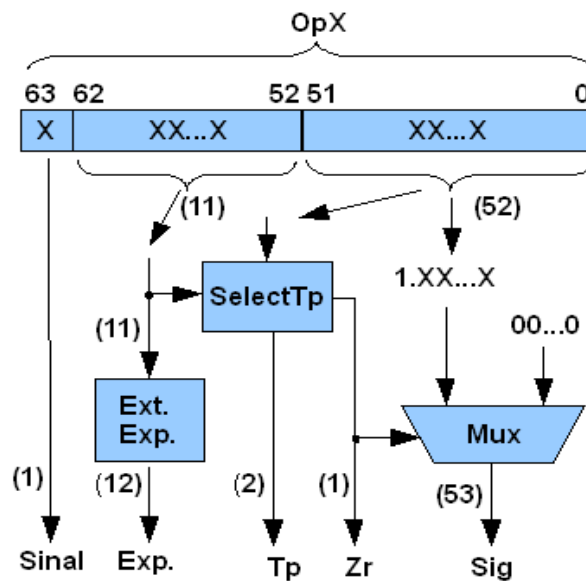


Figura 4.16: Diagrama esquemático interno dos blocos funcionais que formam a Unidade de Aquisição

Dois detalhes são importantes na arquitetura interna desta unidade:

- O primeiro diz respeito ao valor a ser atribuído ao campo expoente: nesta implementação, a fim de minimizar as ocorrências de *overflow* e *underflow* do expoente durante as operações de multiplicação, as quais teriam que ser sinalizadas ou como  $\pm$ Infinito ou como zero, ocorrências estas que em alguns casos são passíveis de reversão durante o processo de arredondamento e normalização, optou-se por estender o campo expoente em mais um bit, de forma que este passasse a ser representado com 12 bits em vez dos 11 bits definidos no padrão IEEE 754. Esta extensão não se dá apenas pelo simples

acréscimo de mais um bit ao expoente, mas através do remapeamento dos valores de uma representação para outra, conforme se pode observar pela Erro: Origem da referência não encontrado mais a frente.

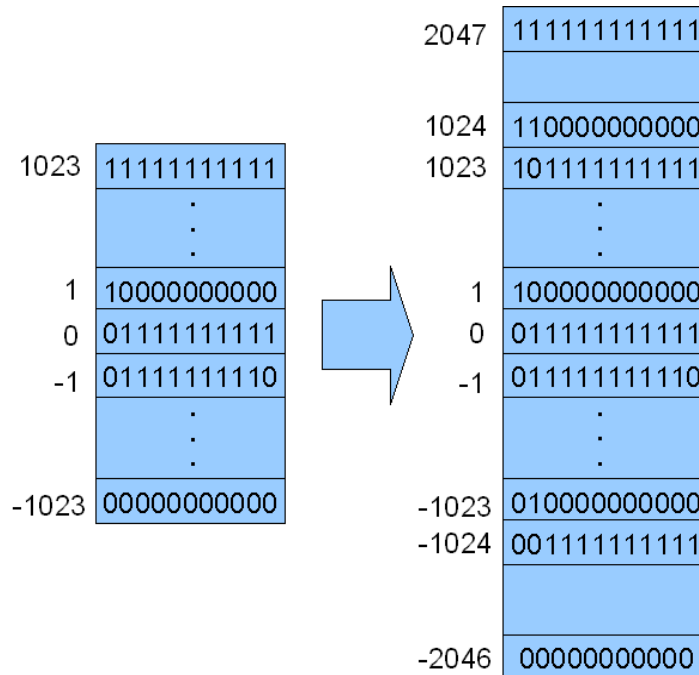


Figura 4.17: Remapeamento dos valores do expoente de 11 para 12 bits

Vale salientar que esta extensão é válida apenas para as operações internas do MAC. Ao final do processo, durante o processo de normalização e arredondamento o expoente do resultado final obtido volta a ser representado com apenas 11 bits, conforme definido no padrão IEEE 754.

A extensão do expoente é feita através do módulo assinalado como Ext. Exp. na Figura 4.16, o qual tem o seu diagrama interno descrito na Figura 4.18 a seguir.

- O segundo ponto a destacar diz respeito ao valor a ser atribuído ao campo significando, o qual depende da análise conjunta tanto do conteúdo dos *bits* do expoente original, *bits* 52 à 62 do operando, quanto dos *bits* da parte fracionária do significando, *bits* 0 à 51 do

operando, a qual é feita pelo pelo módulo *SelectTp*. Se o expoente indicar que o conteúdo representado é zero ou um número no formato *denormal*, o sinal *Zr* será ativado e o significando será totalmente carregado por *bits* zeros. Caso contrário o significando será carregado com o conteúdo dos *bits* 0 à 51 do operando, precedidos do *bit* escondido, o qual é sempre igual a '1'.

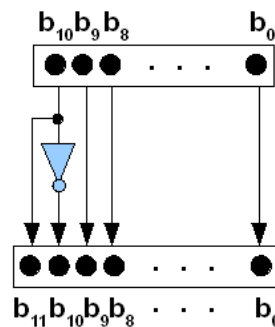


Figura 4.18: Diagrama do módulo que estende o expoente

É também a partir da análise do expoente que é definido o tipo de dado representado pelo operando, o qual é codificado e repassado para a Unidade de Controle de Exceção através dos sinais  $Tp_A$ ,  $Tp_B$  e  $Tp_C$ , para análise e definição do tipo de dado a ser associado ao resultado da operação. A Tabela 11 a seguir traz o padrão de codificação adotado para os sinais  $Tp$ .

Tabela 11: Codificação do tipo de dado do operando

Tipo Dado	Codificação
Número Positivo	000
Número Negativo	100
Infinito Positivo	001
Infinito Negativo	101
NaN	011
NaN	111

### 4.3.2 Unidade de Multiplicação

A Unidade de Multiplicação é a unidade responsável pela multiplicação do conteúdo representado nos operandos A e B, os quais são recuperados pela unidade de aquisição a partir dos valores carregados nas portas Op.A e Op.B respectivamente.

A Figura 4.19 a seguir traz o diagrama interno desta unidade. Como se pode perceber, esta unidade é formada por dois blocos funcionais denominados AddExp, responsável pela operação de soma dos expoentes, e MultSig, responsável pela multiplicação dos significandos.

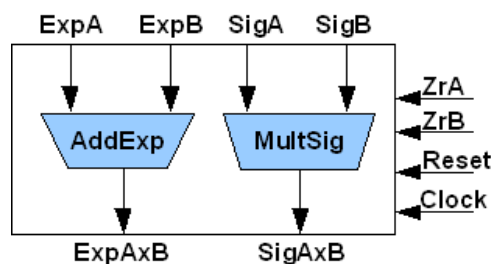


Figura 4.19: Diagrama interno da Unidade de Multiplicação

Como entrada esta unidade recebe os sinais de *Clock* e *Reset*, responsáveis pelo sincronismo e funcionamento da unidade; os vetores *ExpA* e *ExpB*, contendo os expoentes dos operandos, com 12 bits cada; os vetores *SigA* e *SigB*, contendo os significandos dos operandos e os sinais *ZrA* e *ZrB* que sinalizam se o significando de um dos operandos é zero. Caso um dos operandos seja zero força-se o resultado para zero. Este sinal é necessário para corrigir uma deficiência do *MultSig*, que pode falhar se um dos operandos for zero. Esta deficiência será ainda discutida mais a frente neste capítulo.

Como saída, temos os vetores *ExpAxB* e *SigAxB*, com respectivamente 12 e 106 bits cada, resultante das operações de soma dos expoentes e multiplicação dos significandos.



A seguir será feita uma descrição detalhada do funcionamento interno dos blocos AddExp e MultSig.

#### 4.3.2.1 AddExp

Conforme descrito na Seção 2.3.7.2, a operação de soma dos expoentes deve ser acompanhada da devida subtração do valor do *Bias* a fim de não haver sua duplicidade no resultado obtido.

Nesta implementação, ao invés de efetuar-se diretamente uma operação de subtração entre o *Bias* e um dos expoentes ou com o resultado obtido da soma de ambos, optamos por implementar uma operação integrada de soma dos expoentes com o inverso aditivo do *Bias*, ou *-Bias*, obtendo com isto um circuito mais simples e bem mais eficiente. A Figura 4.20 a seguir demonstra como ficou o circuito final utilizado.

Esta implementação só se tornou possível devido a uma peculiaridade do valor do inverso aditivo do Bias, o qual pode ser expresso pela string de bits “100000000001”.

Desta forma, visto que:

1. poder-se-ia utilizar o sinal *Carry-In* do primeiro somador para adicionar o *bit* menos significativo do *-Bias* à soma dos dois expoentes, e;
2. dado que não seria necessário adicionar os *bits* com valor igual a '0', uma vez que os mesmos não alteraram o resultado obtido,
3. precisou-se apenas acrescentar mais um somador para operar o *bit* mais significativo *-Bias* com o bit de *Carry-out* da operação de soma dos dois expoentes.

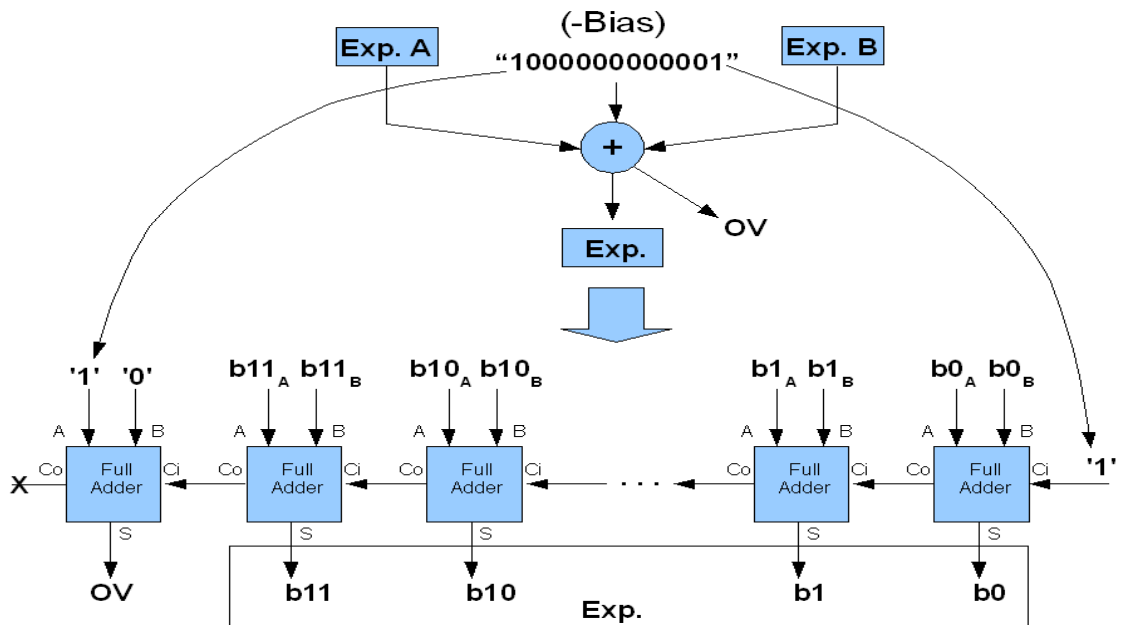


Figura 4.20: Diagrama esquemático do bloco AddExp

#### 4.3.2.1 MultSig

Diferentemente da operação de soma dos expoentes, a qual pode ser facilmente implementada com lógica discreta elementar, a operação de multiplicação dos significandos exige recursos de hardware mais elaborados e portanto mais escassos nos *FPGAs*.

De um modo geral, operações de multiplicação com números binários com vários bits podem ser implementadas de maneira análoga ao procedimento adotado com números decimais, através de operações de multiplicação direta ente os seus dígitos, um a um, acompanhadas do deslocamento e soma dos resultados obtidos.

Nas expressões 11 e 12 a seguir temos respectivamente as funções matemáticas que descrevem o procedimento de multiplicação para números expressos em uma base numérica  $\beta$  qualquer e, de uma forma simplificada, para a aplicação em base binária.

$$X * Y \equiv \sum_{i=0}^{n-1} Y_i * (X_{n-1} \dots X_0) * \beta^i = (P_{2n-1} \dots P_0) \quad (11)$$

$$X * Y \equiv \sum_{i=0}^{n-1} (Y_i \wedge (X_{n-1} \dots X_0)) \ll i = (P_{2n-1} \dots P_0) \quad (12)$$

Na Figura 4.21 temos a demonstração de como fica então uma operação multiplicação de dois números em base binária seguindo o esquema simplificado.

$$\begin{array}{r}
 (bx_{n-1}bx_{n-2}\dots bx_1bx_0) \wedge by_0 \\
 (bx_{n-1}bx_{n-2}\dots bx_1bx_0) \wedge by_1 \\
 \vdots \\
 (bx_{n-1}bx_{n-2}\dots bx_1bx_0) \wedge by_{n-2} \\
 (bx_{n-1}bx_{n-2}\dots bx_1bx_0) \wedge by_{n-1} \\
 \hline
 P_{2n-1}P_{2n-2}bx_{2n-3}P_{2n-4}\dots\dots\dots P_4bx_3P_2P_0P_1P_0
 \end{array}$$

**Figura 4.21: Operação de multiplicação entre os números X e Y em representação binária**

Mesmo operando segundo o esquema simplificado apresentado na expressão 12, devido ao grande número de portas lógicas encontradas no caminho crítico do circuito final obtido, esta solução ainda apresenta uma resposta muito lenta, sendo desta forma considerado uma parte sensível do projeto, podendo ter um grande impacto no desempenho total do sistema.

Existem aqui dois problemas distintos a serem tratados:

1. Primeiramente temos as operações de deslocamento dos bits, as quais podem ser implementadas ou por circuitos de deslocamento síncronos, baseados em flip-flops ou através de lógica combinacional, com o uso de deslocadores tipo Barrel-shift.
2. Em segundo lugar temos as operações de soma entre os produtos dos deslocamentos. Devido a dependência dos resultados entre as operações e ao grande número de *bits* a ser operado, a latência causada por esta parte do circuito pode ter um impacto muito negativo na frequência de operação do sistema como um todo.

A solução para ambos os problemas na maioria dos casos passa pela escolha entre adotar um circuito mais eficiente, que explore o paralelismo intrínseco existente entre as operações, o qual consome mais recursos de hardware mas permite uma execução mais rápida, ou optar-se por um circuito que exija menos recursos de hardware, normalmente utilizando circuitos seqüenciais, procurando com isto ao mesmo tempo aumentar a frequência de operação e o reuso dos recursos de hardware disponíveis, mas que por outro lado, pode exigir vários ciclos de relógio para que a operação seja executada.

Para projetos de alto desempenho, implementados a partir de lógica reconfigurável, normalmente *FPGAs*, atualmente a melhor solução é a utilização dos blocos multiplicadores já disponíveis em diversas *FPGAs* [35][36].

Estes multiplicadores são extremamente rápidos e eficientes, pois são construídos a partir da associação de blocos de deslocamento tipo *Barrel-Shifts* juntamente com somadores tipo *carry look-ahead*.

A grande vantagem desta arquitetura é que ao mesmo tempo em que se reduz o caminho crítico do circuito, permitindo com isto o aumento da frequência de trabalho do sistema, pela utilização dos somadores *carry look-ahead*, permite-se também que a multiplicação ocorra em apenas um ciclo de relógio, através do uso dos blocos *Barrel-shift*.

Entretanto, um cuidado especial deve ser tomado no uso destes blocos multiplicadores, uma vez que os mesmos são construídos a partir de blocos *Barrel-shift*. Estes blocos por operarem deslocamentos de bits através de muxes, e não através de circuitos seqüenciais, flip-flops, podem fazer com que, quando um dos operandos, que será utilizado para controlar o deslocamento, for igual a zero, o resultado não seja zero e sim o outro operando. Para evitar este problema, no circuito proposto optou-se por utilizar uma sinalização externa para indicar a ocorrências destas situações. Nestes casos, o resultado da operação do bloco multiplicador é substituída por zero ao final da operação.

A operação de multiplicação por partes normalmente segue um algoritmo denominado *Dadda multiplier* [37]. Neste algoritmo, primeiramente os bits que formam os operandos são divididos em grupos, de forma que possam ser operados independentemente uns dos outros. Os resultados obtidos são então somados a fim de obter o resultado final desejado. No nosso caso, os bits de entrada foram divididos em 6 grupos com 17 bits e 2 com 2 bits cada, os quais foram multiplicados entre si utilizando os blocos multiplicadores rápidos disponíveis no FPGA, e alguma lógica auxiliar. A escolha por dividir os bits dos operandos em grupos com 17 bits cada diz respeito ao tamanho do maior vetor de bits que os blocos multiplicadores disponíveis no *FPGA* utilizado no projeto conseguem multiplicar.

As operações de soma e multiplicação utilizadas neste algoritmo podem ser executadas em qualquer ordem, tanto em série quanto em paralelo. Naturalmente a operação em série prioriza o uso de recursos enquanto a operação em paralelo prioriza o tempo de execução.

Neste projeto, devido ao foco em alto desempenho escolheu-se operar com a multiplicação em paralelo. Assim sendo, conforme descrito anteriormente, primeiramente dividiu-se os bits de cada um dos significandos em três grupos com 17 bits cada, mais um quarto grupo com dois bits, perfazendo o total de 53 bits.

Desta forma, uma vez que foram organizados 4 grupos de bits, a fim de poder operar diretamente com todos os grupos simultaneamente, precisaríamos utilizar 16 blocos multiplicadores.

A fim de economizar o uso dos blocos multiplicadores, os quais ainda são um recurso escasso nos *FPGAs*, adotou-se uma arquitetura mista, na qual utiliza-se os blocos multiplicadores apenas para as operações onde ambos os operandos continham bits e, para aquelas operações onde pelo menos um dos operandos tem apenas dois bits, utilizamos lógica auxiliar discreta, baseada em portas lógicas simples, em vez dos blocos multiplicadores. Esta arquitetura nos permitiu economizar 7 dos 16 blocos multiplicadores originalmente necessários, com uma redução de 43% no uso dos recursos disponíveis.

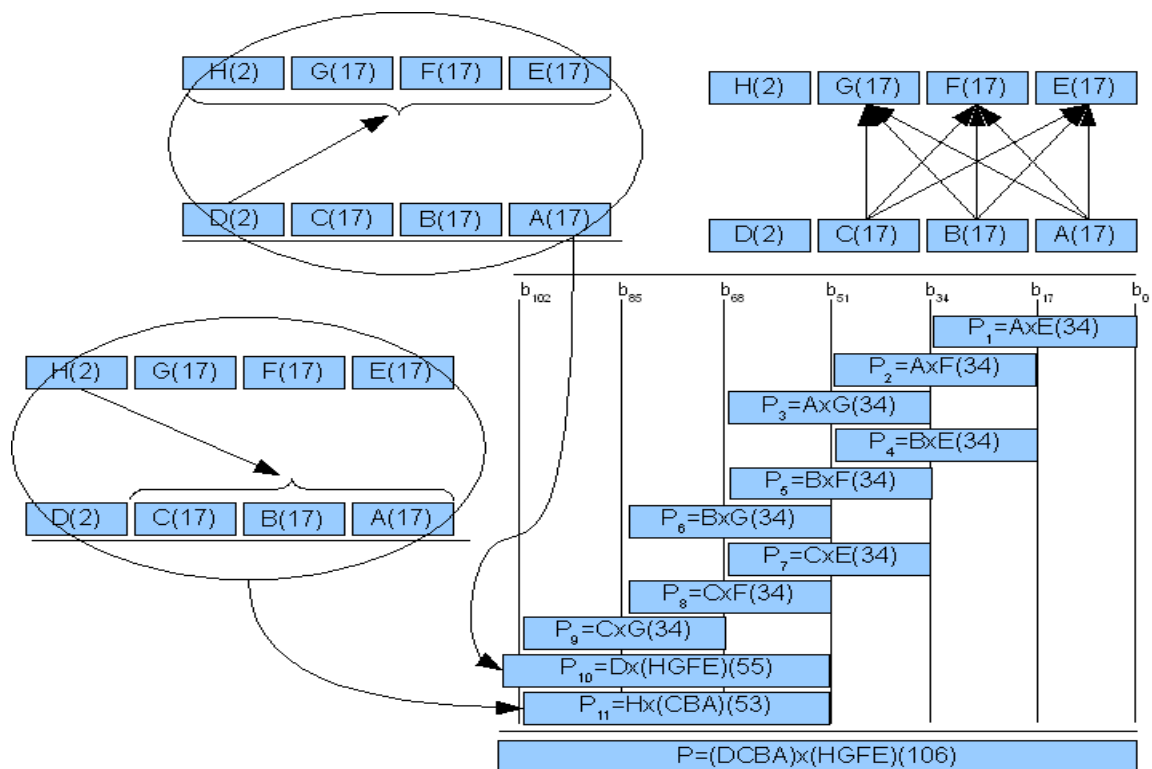


Figura 4.22: Algoritmo Dadda aplicado a multiplicação dos significandos

A Figura 4.22 traz demonstrar o funcionamento da arquitetura implementada.

Os blocos assinalados como A, B, C, E, F e G, todos com 17 bits, são operados diretamente entre si através do uso dos multiplicadores. O bloco assinalados como D e H, ambos com apenas dois bits, são operados de uma só vez respectivamente com todos os bits dos blocos E, F, G e H, e com todos os bits dos

blocos E, F, e G. Em ambos os casos a operação é feita seguindo o esquema apresentado na Figura 4.23 mais a frente.

Após proceder as multiplicações dos grupos de bits, adotou-se um circuito de redução para operar a soma dos produtos parciais, através do qual procurou-se mais uma vez, ao mesmo tempo, otimizar o uso dos recursos de hardware e melhorar o desempenho do circuito como um todo. Na Figura 4.24 mais a frente, temos um diagrama que demonstra o funcionamento do esquema adotado.

Primeiramente, procedemos um rearranjo nos produtos parciais obtidos, de forma a concatenar os produtos que não se sobreponham, a fim de reduzir o número de operações a serem efetuadas.

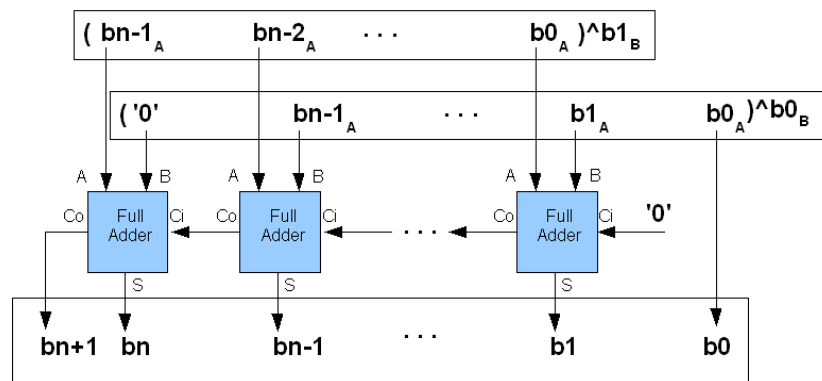


Figura 4.23: Esquema funcional da multiplicação dos operandos D e H, aqui representados pelo operando B

Em seguida, procedemos as somas dos operandos resultantes dois a dois, obtendo os resultados parciais S1, S2 e S3. Neste ponto, uma vez os bits 0 à 16 de P1 e de S3 não se sobrepõem aos bits dos outros operandos, eles são separados para operação nos próximos passos. No passo seguinte operamos novamente a soma dos resultados obtidos chegando aos resultados parciais S4 e S5 os quais são finalmente adicionados e concatenados com os bits já separados de P1 e S3 para compor o resultado final, o qual é atribuído a saída SigAxB.

Com este esquema pudemos reduzir em 43%o uso de unidades somadoras de um bit, as quais passaram das 636 que seriam necessária originalmente para 361.

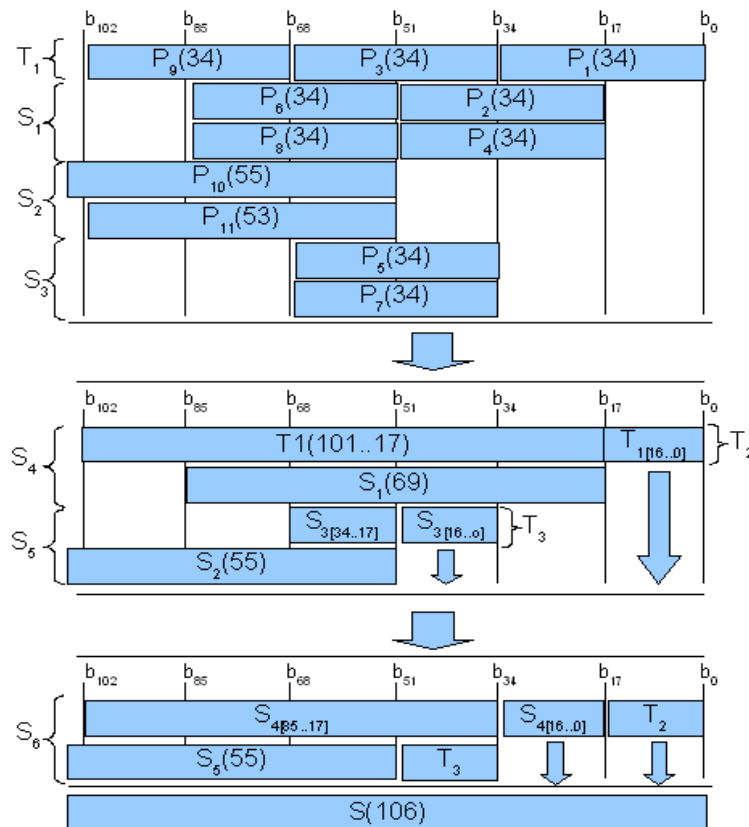


Figura 4.24: Árvore de somas parciais

### 4.3.3 Unidade de Soma

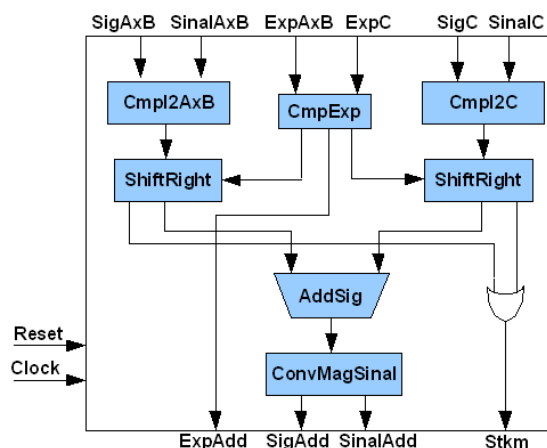
Esta unidade é responsável por efetuar a operação de soma existente entre o resultado da operação de multiplicação entre os operandos A e B e o operando C.

Na Figura 4.25 temos o diagrama em blocos da arquitetura interna desta unidade.

A Unidade de Soma recebe como uma das entradas o significando, o expoente e o sinal resultantes da operação de multiplicação dos operandos A e B,



com respectivamente 106, 12 e 1 bits. Como a outra entrada esta o significando, o expoente e o sinal do operando C, com respectivamente 53, 11 e 1 bits.



**Figura 4.25: Diagrama esquemático funcional da Unidade de Soma**

Como saída a Unidade de Soma fornece o significando, o expoente e o sinal do resultado obtido, com respectivamente 107, 12 e 1 bits, e ainda o sinal Stkm, que será utilizado como entrada na Unidade de Normalização e Arredondamento a fim de melhorar a precisão da resposta desta unidade.

Um detalhe importante com relação a implementação da Unidade de Soma é que, apesar de receber os significandos a serem operados e fornecer o significando resultante no formato de magnitude e sinal, internamente as operações são executadas com os significandos nos formato de complemento a dois. Esta estratégia foi tomada com vistas a simplificar a operação conjunta de números positivos e negativos e reduzir a quantidade de hardware utilizado.

A seguir serão descritos os blocos que formam esta unidade.

#### 4.3.3.1 ConvCmpl2

Os blocos assinalados de *ConvCmpl2* são responsáveis por efetuar a conversão dos significandos dos operandos do formato de magnitude e sinal para o formato de complemento a dois.

Uma vez que o complemento a dois de um número negativo é um bit maior que a sua representação no formato de magnitude e sinal, a saída deste bloco terá sempre 107 bits, independente de o número representado ser positivo ou negativo.

#### 4.3.3.2 ShiftRight

Os blocos assinalados como *ShiftRight* são responsáveis por efetuar os deslocamentos necessários à realização do realinhamento dos significandos.

Internamente, a fim de se obter um melhor desempenho do sistema, estes blocos foram implementados como um *Barrel-Shift* de 106 bits, construído a partir de blocos multiplexadores existentes no *FPGA*, adicionados de uma lógica auxiliar que identifica quando houver o descarte de um *bit* em '1' durante o processo de deslocamento, sinalizando esta ocorrência através do sinal *Stkm*. Esta informação será utilizada para melhorar a precisão do processo de normalização e arredondamento.

#### 4.3.3.3 CmpExp

O bloco *CmpExp* é responsável por definir, a partir da comparação dos expoentes dos operandos, de quanto será o deslocamento a ser aplicado no significando a ser realinhado e qual será o expoente do resultado. Esta definição é feita a partir da observação do resultado da operação de subtração entre os seus

expoentes, de tal forma que, se o expoente resultante da operação de multiplicação entre os operandos A e B for maior que o expoente de C, na saída *NSftA* tem-se zero e na saída *NSftB* a diferença entre os dois expoentes. Caso contrário, se o expoente de C for maior que o expoente de AxB, na saída *NSftB* tem-se zero e na saída *NSftA* a diferença entre os dois expoentes. Em ambos os casos, na saída *ExpAdd* tem-se sempre o maior dentre os dois expoentes comparados.

A Figura 4.26 a seguir, mostra o diagrama interno do bloco CmpExp.

#### 4.3.3.4 AddSig

É apenas um bloco somador, baseado na arquitetura *Look-carry-ahead* onde os significandos são efetivamente adicionados.

#### 4.3.3.5 ConvMagSinal

Este bloco é responsável por converter o resultado obtido na operação de soma dos significandos do modo de complemento a dois para o modo de magnitude e sinal novamente.

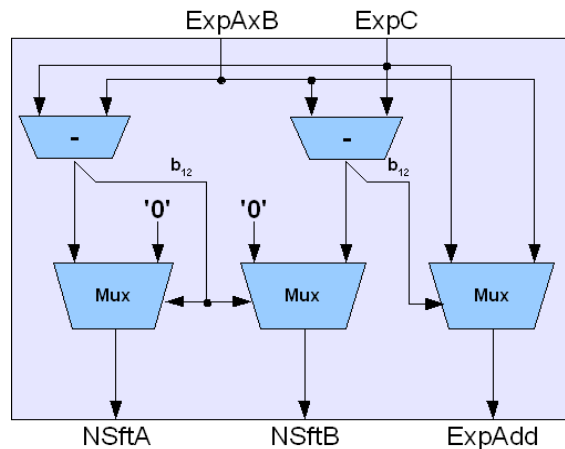


Figura 4.26: Diagrama funcional interno do bloco *CmpExp*

#### 4.3.4 Unidade de Normalização e Arredondamento

Esta unidade é responsável por proceder a normalização e o arredondamento do resultado obtido com as operações de soma e multiplicação. Ou seja, passar do formato interno do *MAC*, tido como infinitamente preciso, com 107 *bits* no significando e 13 no expoente, para o formato padrão do IEEE 754, ponto flutuante de precisão dupla, com 53 *bits* no significando e 11 no expoente.

Internamente esta unidade é formada por três blocos funcionais denominados *NormSig*, *ArrdSig* e *AjustaExp* descritas a seguir.

A Figura 4.27 a seguir traz o diagrama interno desta unidade.

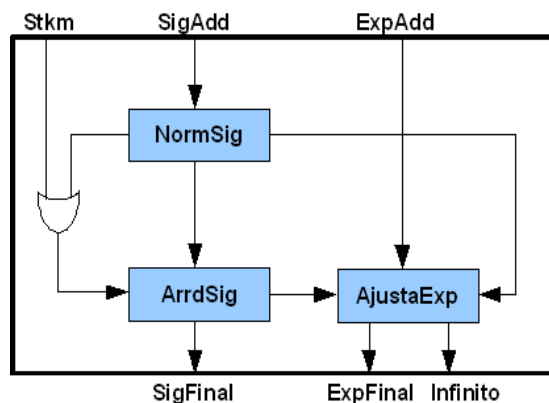


Figura 4.27: Diagrama interno da Unidade de Normalização e Arredondamento

#### 4.3.4.1 NormSig

Este módulo é responsável por efetuar a normalização do significando e por indicar em quanto o expoente deve ser ajustado de forma a poder compor juntamente com o significando normalizado o resultado das operações realizadas.

Conforme exposto na Seção 2.3.6 desta dissertação, o procedimento de normalização está baseado no cálculo da diferença entre a posição do primeiro bit '1' encontrado no significando e a posição onde este deveria se encontrar. No diagrama da Figura 4.28 a seguir, esta operação é executada pelo *CntDslc*.

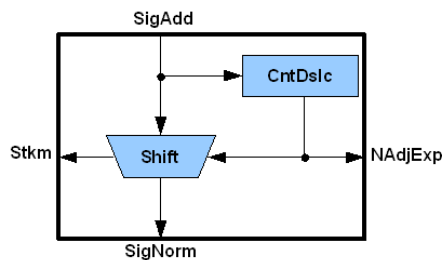


Figura 4.28: Diagrama interno do bloco funcional NormSig

O resultado obtido é então utilizado ao mesmo tempo para efetuar o deslocamento do significando e para ajustar o expoente.

O deslocamento do significando é efetuado instantaneamente através de multiplexadores, em um esquema tipo *Barrel-Shift*. Durante o deslocamento também é observado se algum bit em '1' está sendo descartado, sendo esta ocorrência sinalizada através do sinal *Stkm*. Esta informação será utilizada em conjunto com o sinal *Stkm* proveniente da Unidade de Soma, a fim de melhorar a precisão do resultado final obtido pelo processo de arredondamento.

#### 4.3.4.2 ArrdSig

Este módulo é responsável por efetuar o arredondamento do significando, agora já normalizado, de 103 para 53 bits, utilizando o modo *Round to Nearest*. A Figura 4.29 traz o diagrama interno deste o módulo.

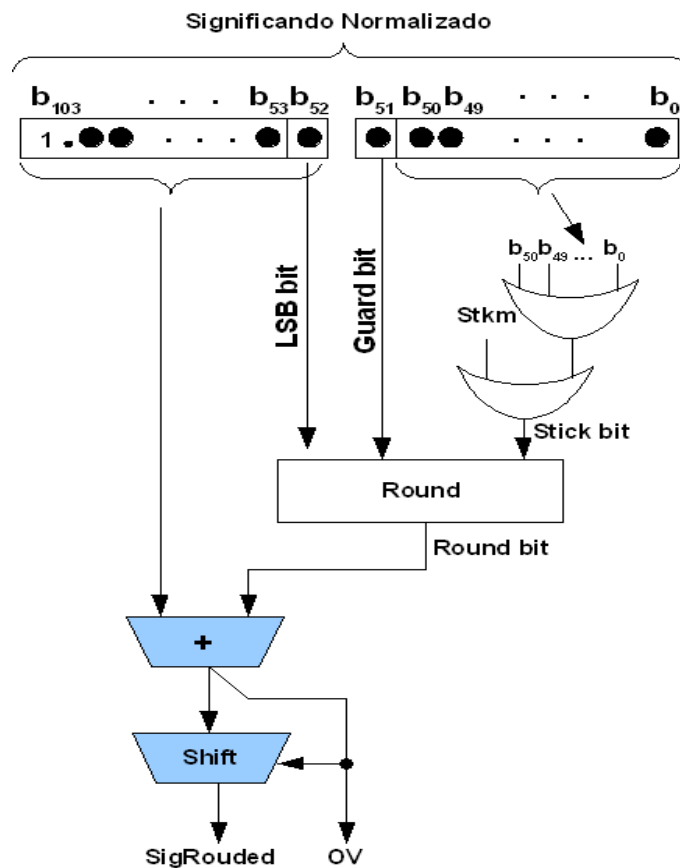


Figura 4.29: Diagrama interno do ArrdSig

O procedimento aqui adotado segue em tudo o que já foi apresentado na Seção 2.3.5, com exceção da geração do *Stick-bit*, o qual, diferentemente do que foi apresentado, leva em conta também os sinais *Stkm* gerados pela Unidade de Soma e durante processo de normalização. O procedimento aqui adotado prevê também a possibilidade de efetuar uma re-normalização no resultado obtido pela soma do significando pré-arredondado com o *Round bit*, através da aplicação de mais um *shift* ao resultado obtido. Esta re-normalização pode ser necessária caso ocorra um

*overflow* no resultado da soma do significando pré-arredondado com o *Round bit*. A ocorrência deste *overflow* é também sinalizado ao módulo que irá proceder o ajuste do expoente final através do sinal OV (Overflow).

#### 4.3.4.3 AjustaExp

Este módulo é responsável por efetuar o ajuste do expoente para compatibilizá-lo com o significando arredondado.

Este ajuste do expoente se dá em duas etapas:

1. Primeiramente procede-se a adição do expoente original com o valor contido no sinal *NAdjExp* calculado durante a normalização e com a indicação de overflow proveniente do arredondamento.
2. Em seguida verifica-se se o expoente obtido pode ser representado no formato padrão do IEEE 754, ou seja com 11 bits. Caso contrário, isto indicará que ocorreu um *overflow* ou um *underflow* no expoente obtido, e que o resultado final das operações conjuntas de multiplicação e adição deve ser representado respectivamente ou por  $\pm$  Infinito ou pelo zero.
3. Esta verificação é feita simplesmente comparando-se os dois bits mais significativos do expoente, os quais devem ser diferentes um do outro, ou seja, se o bit 11 for '0' o bit 10 deve ser '1' e vice-versa. Um resultado "11" nesta comparação indicará a ocorrência de um *overflow*, o qual será sinalizado como infinito no resultado final das operações. Por outro lado, a ocorrência de um "00" indicará a ocorrência de um *underflow*, o qual forçará o resultado final das operações a ser representado como zero.

4. Por fim, converte-se o expoente da notação interna com 12 bits para a notação padrão do IEEE 754, com 11 bits, através do esquema demonstrado na Figura 4.30.

Por fim, compara-se o resultado obtido para verificar se este é um expoente válido, ou seja, se este não é igual de “1111111111”, o que indicaria um *overflow*, e nem igual a “00000000000”, o que indicaria um *underflow*. Caso tenha ocorrido um *overflow* ou *underflow* durante as operações, o resultado obtido dever ser convertido ou para  $\pm$  Infinito ou para zero, respectivamente.

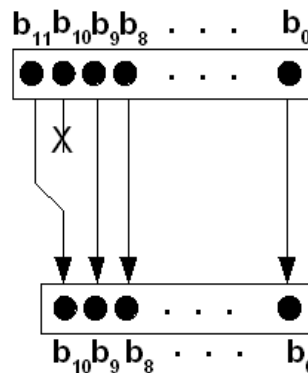


Figura 4.30: Esquema para conversão do expoente de 12 para 11 bits

### 4.3.5 Unidade de Controle de Exceção

A Unidade de Controle de Exceção é responsável por monitorar os resultados das operações internas do *MAC*, com vistas a identificar ocorrências que devam ser sinalizadas ou como  $\pm$  Infinito ou como NaN.

Este monitoramento é baseado no tipo dos operandos envolvidos nas operações e na verificação da ocorrência de um resultado infinito durante o processo de normalização e arredondamento.



Internamente esta unidade possui apenas três módulos denominados *CtrlMult*, *CtrlSoma* e *CtrlNorm*, responsáveis por monitorar respectivamente as operações de multiplicação, soma, normalização e arredondamento dos resultados.

A Figura 4.31 traz a arquitetura interna desta unidade.

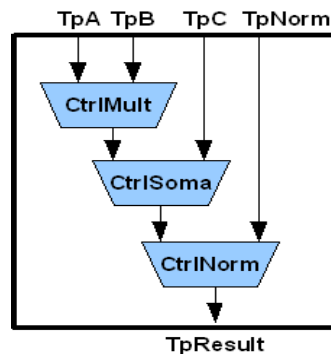


Figura 4.31: Diagrama interno da Unidade de Controle de Exceções

Como entrada esta unidade recebe os vetores *TpA*, *TpB* e *TpC*, respectivamente os tipos dos operadores A, B e C, e ainda o vetor *TpNorm* com o tipo do resultado obtido no processo de normalização e arredondamento.

Todos os quatro vetores seguem o padrão de sinalização adotado pela unidade de aquisição para os tipos de operadores, descrito na Tabela 13.

As Tabelas a seguir trazem os resultados fornecidos à saída dos módulos *CtrlMult* e *CtrlSoma* de acordo com o tipo de operando utilizado em cada operação.

Tabela 12: Resultado à saída do *CtrlMult* de acordo com os tipos dos operandos para as operações de multiplicação

	<b>+Num</b>	<b>-Num</b>	<b>+∞</b>	<b>-∞</b>	<b>NaN</b>
<b>+Num</b>	+Num	-Num	+∞	-∞	NaN
<b>-Num</b>	-Num	-Num	-∞	+∞	NaN
<b>+∞</b>	+∞	-∞	+∞	-∞	NaN
<b>-∞</b>	-∞	+∞	-∞	+∞	NaN
<b>NaN</b>	NaN	NaN	NaN	NaN	NaN

Tabela 13: Resultado à saída do CtrlSoma de acordo com os tipos dos operandos para as operações de Soma

	<b>+Num</b>	<b>-Num</b>	<b><math>+\infty</math></b>	<b><math>-\infty</math></b>	<b>NaN</b>
<b>+Num</b>	<b>+Num</b>	<b>+/- Num</b>	<b><math>+\infty</math></b>	<b><math>-\infty</math></b>	<b>NaN</b>
<b>-Num</b>	<b>+/- Num</b>	<b>-Num</b>	<b><math>+\infty</math></b>	<b><math>-\infty</math></b>	<b>NaN</b>
<b><math>+\infty</math></b>	<b><math>+\infty</math></b>	<b><math>+\infty</math></b>	<b><math>+\infty</math></b>	<b>NaN</b>	<b>NaN</b>
<b><math>-\infty</math></b>	<b><math>-\infty</math></b>	<b><math>-\infty</math></b>	<b>NaN</b>	<b><math>-\infty</math></b>	<b>NaN</b>
<b>NaN</b>	<b>NaN</b>	<b>NaN</b>	<b>NaN</b>	<b>NaN</b>	<b>NaN</b>

Com relação ao módulo *CtrlNorm*, que monitora o resultado da operação de normalização e arredondamento, este na verdade apenas compara o resultado obtido pela análise das operações de soma e multiplicação e, caso não tenha ocorrido nenhuma exceção em nenhuma destas operações assume como tipo da saída o tipo reportado no sinal *TpNorm*.

#### 4.3.6 Unidade de Formatação dos Resultados

A Unidade de Formatação dos Resultados simplesmente seleciona o resultado a ser apresentado a partir do tipo de dado informado pela Unidade de Controle de Exceção.

Esta unidade recebe como entradas o sinal, o expoente e o significando do resultado normalizado, com os quais reconstrói a representação do número em ponto flutuante, além do *TpResult* que informa o tipo de dado a ser fornecido.

Internamente esta unidade já dispõe das representações padrão para +Infinito, -Infinito e NaN, as quais são utilizadas como saída caso seja necessário.

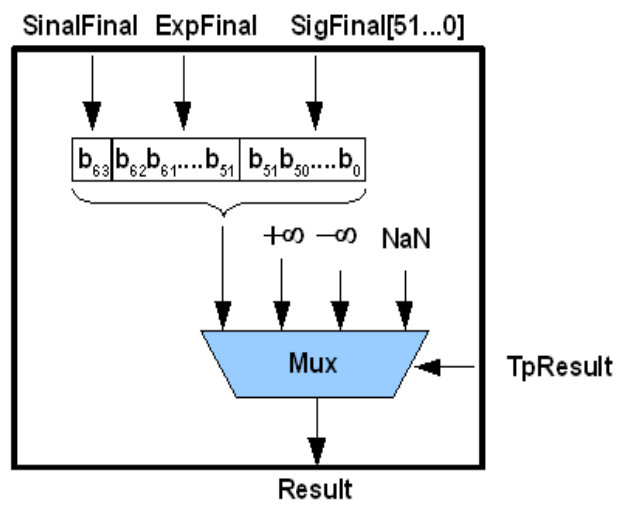


Figura 4.32: Diagrama interno da Unidade de Formatação dos Resultados

## 4.4 ARQUITETURA FINAL E PIPE-LINE DO SISTEMA

A Figura 4.33 mais a frente traz a arquitetura final e o pipe-line do sistema implementado. As marcações laterais indicam o estágio do pipe-line ou, em outras palavras, o ciclo de relógio no qual as operações ocorrem.

### 4.4.1 Descrição dos Estágios do Pipe-line

Antes de chegarem ao primeiro estágio de pipe-line, os operandos passam pela unidade de aquisição, onde são classificados com relação ao tipo de dado que representam, e seus bits são agrupados e distribuídos para os demais módulos do multiplicador e acumulador. A unidade de aquisição é totalmente combinacional, ou seja, não necessita do sinal de relógio para operar, por este motivo ela não está incluída como fazendo parte dos estágio do pipe-line.

A seguir descreveremos a operações efetuadas em cada estágio do pipe-line.

Estágio S1:

- Na unidade de multiplicação ocorrem simultaneamente a multiplicação parcial dos bits dos significandos de OpA e OpB, conforme descrito na Seção 4.2.1, e a soma dos expoentes de OpA e OpB.
- Na unidade de controle de exceções ocorre a verificação de possíveis exceções que possam ser geradas pelos tipos de dados representados por OpA e OpB.
- Os sinais ExpC, SinalC, SigC e TpC, provenientes da unidade de aquisição, e o sinal algébrico da operação de multiplicação, obtido pela aplicação da função lógica OU EXCLUSIVO com os sinais

algébricos de OpA e OpB, são introduzidos em memórias tipo Fifo para propagados sem alteração até ao estágio S5 do pipe-line.

Estagio S2:

- Na unidade de multiplicação tem início a operação de soma dos produtos parciais gerados pelo multiplicador parcial. A operação de soma dos produtos parciais consome 3 ciclos de relógio, estendendo-se até o estágio S4. O expoente resultante da da operação se soma entre os expoentes de OpA e OpB é introduzido em uma memória tipo Fifo para ser propagado sem alteração até ao estágio S5 do pipe-line.
- Na unidade de controle de exceções ocorre a verificação de possíveis exceções que possam ser geradas pela operação de adição entre o tipo de dado previsto para como resultado para a operação de multiplicação e o tipo de dado de OpC.

Estagio S3:

- Na unidade de multiplicação continua a operação de soma dos produtos parciais, e a propagação do resultado da soma dos expoentes de OpA e OpB;
- Na unidade de controle de exceção o resultado da verificação de possíveis exceções geradas pelas operações é introduzida em uma memoria tipo Fifo para der propagada sem alteração até fim do último estágio do pipe-line

Estagio S4:

- Na unidade de multiplicação continua a operação de soma dos produtos parciais.

Estagio S5:

- Na unidade de soma ocorrem simultaneamente as operações de conversão do modo de magnitude e sinal para o modo de complemento a dois do significando resultante da operação de multiplicação de OpA e OpB e do significando de OpC.
- Ainda na unidade de soma ocorre a comparação entre o expoente resultante da operação de multiplicação entre OpA e OpB e o expoente de OpC. A partir desta comparação são definidos o expoente resultante da operação integrada de multiplicação e soma dos operandos OpA, OpB e OpC, e como se dará o realinhamento do significando resultante da soma dos produtos parciais e do significando de OpC.

#### Estágio S6:

- Na unidade de soma, baseada na análise dos expoentes ocorrida no estágio anterior, efetua-se o deslocamento à direita do significando que será realinhado.
- Ainda na unidade de soma, o expoente escolhido como expoente resultante da operação é introduzido em uma memória tipo Fifo para ser propagada sem alteração até ao estágio S9 do pipe-line.

#### Estágio S7:

- Na unidade de soma, efetua-se a operação de soma entre os significandos já complementados e realinhados, ao mesmo tempo em que a indicação de descarte de bit válido durante as operações de deslocamento dos significandos é introduzida em uma memória tipo Fifo para ser propagada sem alteração até ao estágio S9 do pipe-line.

#### Estágio S8:

- Na unidade de soma, o resultado da operação de soma entre os significandos é convertida do modo de complemento a dois para o modo de magnitude e sinal.

**Estagio S9:**

- O sinal algébrico obtido da conversão de complemento a dois para magnitude e sinal é introduzido em uma memória Fifo para ser propagado sem alteração até ao fim do pipe-line.
- Na unidade de normalização e arredondamento, a indicação de descarte de bit válido e o expoente resultante da operação de soma, ambos proveniente da unidade de soma, são introduzidos em memórias tipo Fifo para serem propagados sem alteração respectivamente até o fim dos estágios S11 e S10 do pipe-line.
- Ainda na unidade de normalização e arredondamento, tem início o processo de normalização do significando resultante da operação de soma. Este processo consome dois ciclos de relógio e se estenderá até ao estágio S10 do pipe-line.

**Estagio S10:**

- Na unidade de normalização e arredondamento, é concluído o processo de normalização do significando resultante da operação de soma.

**Estagio S11:**

- Na unidade de normalização e arredondamento, efetua-se o arredondamento do significando ao mesmo em que é feita a verificação da ocorrência de overflow ou underflow no significando. Baseada nesta verificação é feito o ajuste do expoente e a sinalização do tipo final do dado obtido.

Após a execução de todos os estágios do pipe-line, a unidade de controle de exceção decide, a partir das possíveis indicações de exceções ocorridas tanto na análise dos operandos quanto durante o processo de normalização e arredondamento o tipo do dado resultante das operações integradas. A partir da

definição do tipo do dado resultante a unidade de formatação dos resultados fornece o resultado final das operações integradas de multiplicação e soma dos operandos OpA, OpB e OpC.

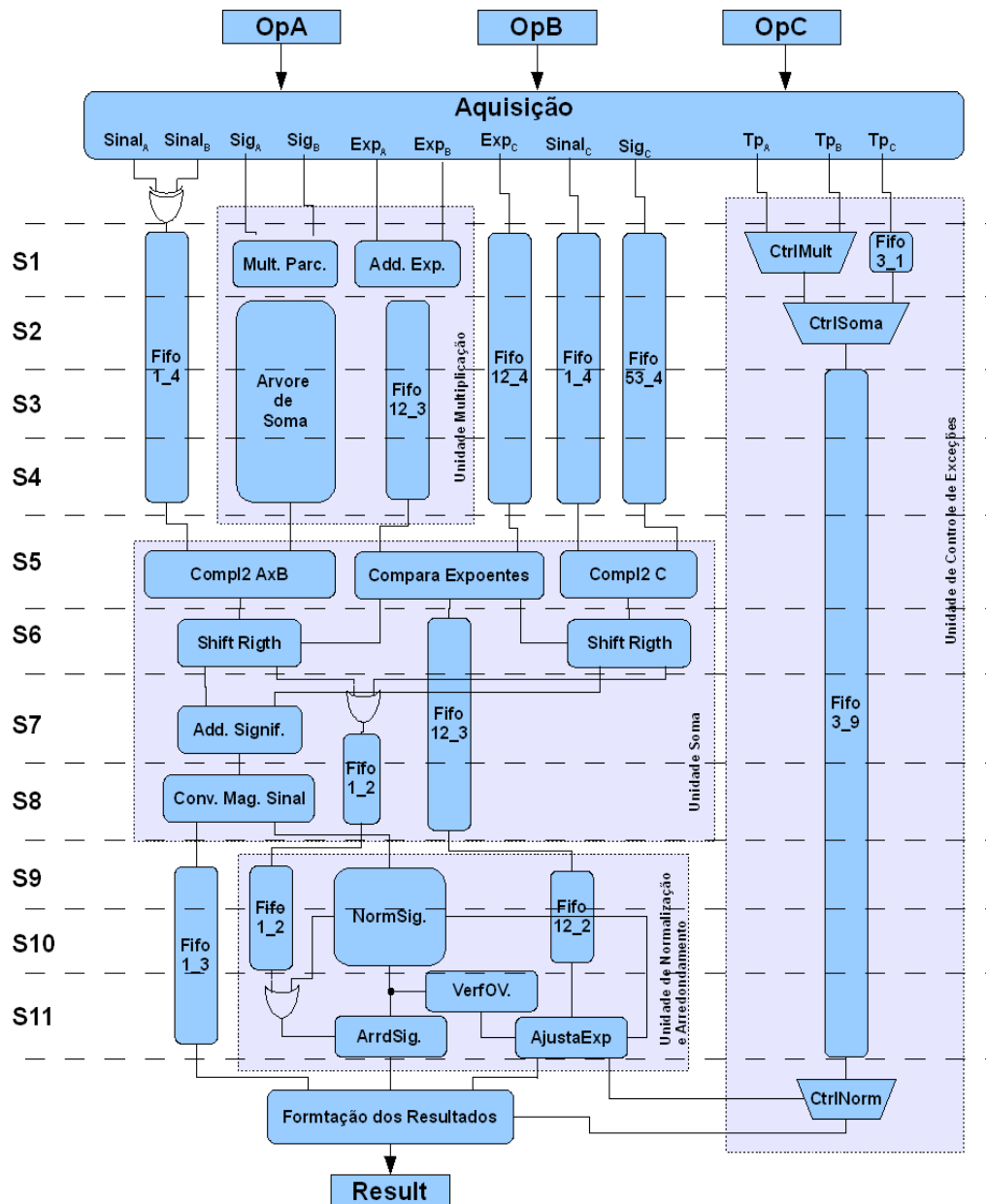


Figura 4.33: Arquitetura final e Pipe-line do Sistema



## 5 ESTUDO DE CASO - IMPLEMENTAÇÃO NA PLACA PCI

A fim de poder validar o funcionamento do módulo *MAC* desenvolvido, foi implementado um projeto no qual o *MAC* foi mapeado em um *FPGA*, pertencente a uma placa de prototipação, com interface de barramento *PCI*. Uma aplicação foi desenvolvida em C++, com o intuito de gerenciar a transferência de dados e a execução do algoritmo de multiplicação e soma de números de ponto flutuante, 64 bits precisão dupla no *FPGA*.

O projeto do *MAC* foi sintetizado e configurado para um dispositivo da *Xilinx*, o *FPGA Virtex II XC2V6000*[35], disponível na placa de prototipação acima citada, denominada *Virtex-II Development Board* [38]. A Figura 5.1 a seguir apresenta uma foto da placa utilizada.

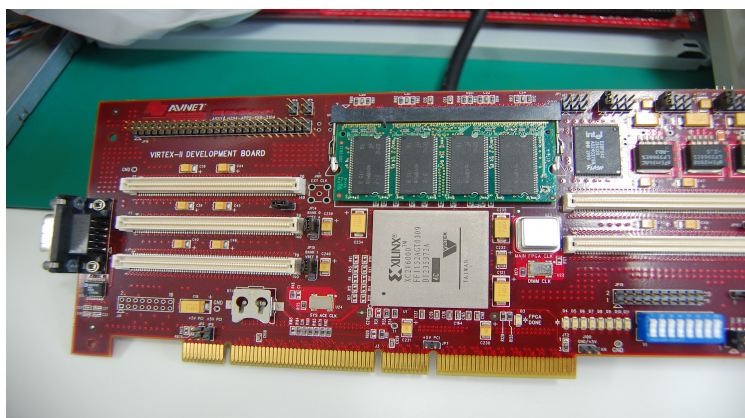


Figura 5.1: Placa de prototipação Virtex-II Development Board da AVNET

O família *XC2V6000* utilizado neste estudo de caso é descrita como sendo destinada a aplicações de alto desempenho, tendo sido especialmente desenvolvida para emprego em aplicações em telecomunicações, *wireless*, rede de computadores, processamento de vídeo e processamento digital de sinal. Os

componentes desta família dispõem ainda de suporte para interfaceamento direto com barramentos PCI e com memória DDR[35].

A Figura 5.2 a seguir traz uma listagem das principais características dos componentes da família Virtex-II. Como se pode observar, o dispositivo utilizado neste projeto possui 6 milhões de portas lógicas equivalentes, distribuídas em 33792 unidades funcionais, ou *slices*, 1056 *kbits* de memória *RAM* distribuídas, 144 blocos de multiplicadores, 12 controladores internos da frequência de trabalho, os DCM, e

Device	System Gates	CLB (1 CLB = 4 slices = Max 128 bits)			Multiplier Blocks	SelectRAM Blocks		DCMs	Max I/O Pads <sup>(1)</sup>
		Array Row x Col.	Slices	Maximum Distributed RAM Kbits		18 Kbit Blocks	Max RAM (Kbits)		
XC2V40	40K	8 x 8	256	8	4	4	72	4	88
XC2V80	80K	16 x 8	512	16	8	8	144	4	120
XC2V250	250K	24 x 16	1,536	48	24	24	432	8	200
XC2V500	500K	32 x 24	3,072	96	32	32	576	8	264
XC2V1000	1M	40 x 32	5,120	160	40	40	720	8	432
XC2V1500	1.5M	48 x 40	7,680	240	48	48	864	8	528
XC2V2000	2M	56 x 48	10,752	336	56	56	1,008	8	624
XC2V3000	3M	64 x 56	14,336	448	96	96	1,728	12	720
XC2V4000	4M	80 x 72	23,040	720	120	120	2,160	12	912
XC2V6000	6M	96 x 88	33,792	1,056	144	144	2,592	12	1,104
XC2V8000	8M	112 x 104	46,592	1,456	168	168	3,024	12	1,108

Figura 5.2: Características dos componentes da família Xilinx Virtex II [35]

1104 pinos de entrada/saída(IOB).

A Figura 5.3 a seguir traz um diagrama reduzido da arquitetura interna dos componentes da família Virtex-II.

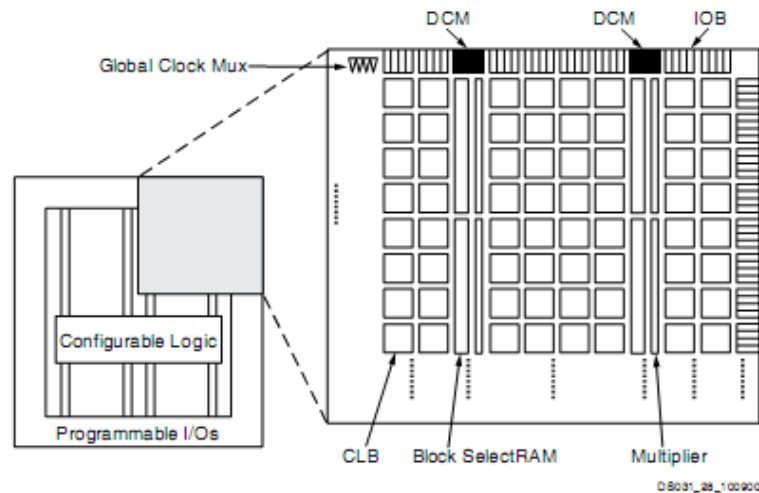


Figura 5.3: Diagrama sucinto da arquitetura interna da Virtex II [35]

Os resultados obtidos da síntese conjunta do controlador do MAC(módulo multiplicador/somador) e do *Soft-Core* da interface *PCI* podem ser vistos na Figura 4.17.

Device Utilization Summary				
Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slice Flip Flops	5,027	67,584	7%	
Number of 4 input LUTs	6,061	67,584	8%	
Logic Distribution				
Number of occupied Slices	4,949	33,792	14%	
Number of Slices containing only related logic	4,949	4,949	100%	
Number of Slices containing unrelated logic	0	4,949	0%	
Total Number of 4 input LUTs	7,256	67,584	10%	
Number used as logic	6,061			
Number used as a route-thru	885			
Number used for Dual Port RAMs	64			
Number used as Shift registers	246			
Number of bonded IOBs	49	824	5%	
IOB Flip Flops	97			
Number of Tbufs	256	16,896	1%	
Number of Block RAMs	17	144	11%	
Number of MULT18X18s	9	144	6%	
Number of GCLKs	2	16	12%	
Number of BSCANs	1	1	100%	
Number of RPM macros	19			
Total equivalent gate count for design	1,265,371			
Additional JTAG gate count for IOBs	2,352			

Figura 5.4: Resultado da síntese conjunta

O projeto final atingiu uma frequência de trabalho de 69.44 Mhz. Entretanto, devido à restrições relativas a frequência de trabalho do barramento PCI adotado, o sistema final ficou operando a uma frequência de 33 Mhz.

Como se pode observar, o módulo do *MAC* junto com a interface do barramento PCI, ocuparam 4934 *Slices* do chip, ou apenas 14% da lógica disponível no *Virtex-II(XC2V6000)*. O *MAC* consumiu apenas 9 blocos multiplicadores, ou 6% dos disponíveis no *FPGA*. Sendo assim, pode-se afirmar que o dispositivo *XC2V6000* empregado neste projeto pode suportar até 7 unidades semelhantes de aritmética de ponto flutuante de 64 bits.

Para a interação com o aplicativo em software e reconhecimento pelo Sistema Operacional do novo módulo de processamento aritmético tornou-se necessário também, o desenvolvimento de um *device-driver* especial para a plataforma PCI e o *MAC*. Este *device-driver* foi desenvolvido através de outro um projeto do mesmo grupo de pesquisa.

Conforme se pode observar pela Figura 5.5, o *device-driver* faz parte de uma camada especial de aplicações, responsáveis por intermediar o acesso aos recursos de hardware por qualquer outra aplicação do sistema.

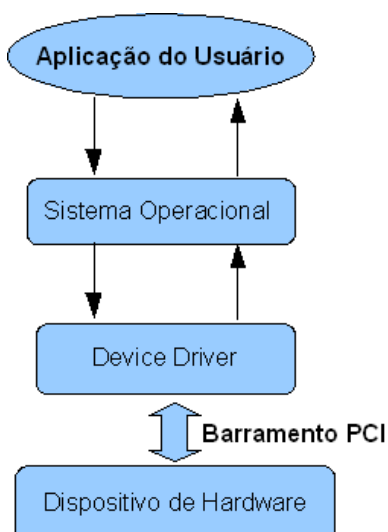


Figura 5.5: Camadas de abstração de software

Como aplicativo de teste foi desenvolvido uma aplicação básica em C++, tipo linha de comando, a qual solicita ao usuário que digite os três valores a serem operados. Os valores podem ser informados em qualquer das formas válidas para o formato de ponto flutuante de precisão dupla. A aplicação se encarrega então de converter os valores de entrada para o formato de vetor binário e de repassá-los ao módulo *MAC* de forma adequada.

O módulo *MAC* efetua as operações de multiplicação e adição, retornando o resultado obtido via SO. O resultado é então convertido pela aplicação, do formato binário, para os formatos hexadecimal e de ponto flutuante, para que estes possam ser conferidos pelo usuário.



## 6 CONCLUSÕES E TRABALHOS FUTUROS

O uso de co-processadores aritméticos de aplicação específica, implementados em *FPGA*, são uma realidade presente em muitos sistemas atualmente oferecidos pelos grandes fabricantes de supercomputadores.

Cada vez mais tem-se observado a busca por métodos alternativos para melhorar o desempenho e reduzir o consumo de energia, não apenas dos computadores em geral, mas de diversos outros tipos de sistemas, principalmente aqueles que requerem um alto desempenho computacional.

Por outro lado, os freqüentes avanços da matemática computacional em aplicações complexas, têm resultado em sistemas extremamente custosos e ineficientes se implementados com os recursos aritméticos atualmente disponíveis nos processadores convencionais, muitas das quais têm se demonstrado simplesmente inexecutáveis sem um suporte adequado de hardware. Esta realidade aponta para a necessidade do desenvolvimento de novas tecnologias e paradigmas de computação que permitam o desenvolvimento de computadores mais flexíveis, com componentes de hardware que possam ser moldados a novos algoritmos, dissipem menos energia, reduzindo assim a necessidade constante de novos investimentos.

Tecnologias como as de hardware reconfigurável têm sido vista como uma solução para este tipo de problema e, neste cenário, os co-processadores de aplicação específica implementados nestes dispositivos têm despontado como uma das melhores alternativas na busca de se atingir estes objetivos.

Entretanto, o desenvolvimento de tais co-processadores reconfiguráveis tem exigido um profundo conhecimento não só dos detalhes funcionais da aritmética envolvida no algoritmo a ser implementado, mas principalmente dos recursos disponíveis no dispositivo reconfigurável a ser utilizado. Além disso, vale salientar

a necessidade de conhecimento adequado das metodologias de projeto e linguagem de descrição de hardware.

Nesta dissertação, considerando este novo paradigma de computação reconfigurável, apresentamos o projeto de um co-processador de aplicação específica, capaz de executar de maneira integrada as operações de multiplicação e soma/subtração de números em ponto flutuante de precisão dupla, totalmente compatível com o padrão IEEE 754. Este sistema foi desenvolvido com vistas a construção de um co-processador de aplicação específica dedicado à operação de multiplicação de matrizes densas.

O projeto aqui descrito foi totalmente modelado, implementado e testado, tendo sido validado com sucesso em uma plataforma de prototipação de hardware diretamente conectada ao barramento PCI de um computador tradicional tipo PC. Um programa em C++ foi desenvolvido para demonstração do módulo aritmético. Seu desempenho nesta plataforma com apenas 33 Mhz, foi de 66 MFLOPS. Na plataforma reconfigurável *RASC*, para a qual se destina o projeto do co-processador específico dedicado à multiplicação de matrizes, executando a 200 MHz, o *MAC* teve um desempenho unitário, previsto, de 400 MFLOPS.

Como continuação do trabalho aqui apresentado, identificamos a possibilidade de implementação de núcleos(*cores*) aritméticos mais densos, com um maior número de operações aritméticas integradas e que permitam uma exploração maior do paralelismo entre as operações e um maior reuso dos dados utilizados.

Estas características de distribuição de dados estão presentes em algoritmos utilizados em implementações de transformadas de Fourier, redes neurais artificiais, processamento de imagem, modelagem computacional, etc. Identificamos também a possibilidade da implementação de novos algoritmos aritméticos que façam uso intensivo de operações em ponto flutuante, tais como as operações aritméticas envolvendo matemática intervalar.



## 7 REFERÊNCIAS BIBLIOGRÁFICAS

- [1] PALMER, J.. The Intel 8087 Numeric Processor. **ISCA '80: Proceedings of the 7th annual symposium on Computer Architecture**. New York, NY, USA: p.174--181, 1980
- [2] PAWLOWSKI, S. . Petascale Computing Research Challenges Petascale Computing Research Challenges - A Manycore Perspective. **HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture**. Phenix, Arizona: p.96, 2007
- [3] SASS, R. et al.. Investigating the Feasibility of FPGA-Based Petascale Computing. **FCCM 2007: 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines**. North Carolina Univ., Charlotte: p. 127-140, 2007
- [4] Silicon Graphics, Inc.. . Disponível em: <www.sgi.com>. Acesso em: ago. 2008.
- [5] International Business Machines. . Disponível em: <www.ibm.com>. Acesso em: ago. 2008.
- [6] Nallatech. . Disponível em: <www.nallatech.com>. Acesso em: ago. 2008.
- [7] Cray Inc.. . Disponível em: <www.cray.com>. Acesso em: ago. 2008.
- [8] CAMPBELL, S. J.; KHATRI, S. P. . Resource and Delay EfficientMatrix Multiplication using Newer FPGA Devices. **GLSVLSI '06: Proceedings of the 16th ACM Great Lakes symposium on VLSI**. New York, NY, USA: p.308--311, 2006
- [9] Open SystemC Initiative. **IEEE Std. 1666-2005**. Disponível em: <www.systemc.org>. Acesso em: ago. 2008.

- [10] IEEE 1364. . Disponível em: <[www.systemverilog.org](http://www.systemverilog.org)>. Acesso em: ago. 2008.
- [11] XILINX Inc. **Virtex-5 FPGA Data Sheet**. Disponível em: <[http://www.xilinx.com/support/documentation/data\\_sheets/ds202.pdf](http://www.xilinx.com/support/documentation/data_sheets/ds202.pdf)>. Acesso em: ago. 2008.
- [12] ALTERA Inc. **Stratix III FPGAs**. Disponível em: <[www.altera.com/literature/br/br-stratixIII.pdf](http://www.altera.com/literature/br/br-stratixIII.pdf)>. Acesso em: ago. 2008.
- [13] GUO, Z.; NAJJAR, W.; VAHID, F.; VISSERS, K.. A Quantitative Analysis of the Speedup Factors of FPGAs over Processors. **FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays**. Monterey, California, USA: p.162-170, 2004
- [14] GOLDBERG, D. . What every computer scientist should know about floating-point arithmetic. **ACM Comput. Surv.** : v.**23**, p.5--48, 1991
- [15] GILOIM, W. K.. Reflections on the 80th birthday of the German computing pioneer. **SIGNUM Newsl.** : v.**33**, p.11-16, 1998
- [16] SEVERANCE, C. . IEEE 754: An Interview with William Kahan. **Computer** . : v.**31**, p.114-115, 1988
- [17] IEEE Standards Activities Department. **DRAFT Standard for Floating-Point Arithmetic P754**. Disponível em: <<http://www.validlab.com/754R/drafts/archive/2006-10-04.pdf>>. Acesso em: ago. 2008.
- [18] CORNEA, M. et al. A Software Implementation of the IEEE 754R Decimal Floating-Point Arithmetic Using the Binary Encoding Format. **IEEE Transactions on Computers** . : v.**58**, p.148-162, 2009

- [19] IEEE Standards Committee 754. **IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985**. New York, USA: Institute of Electrical and Electronics Engineers, 1985
- [20] IEEE Standards Committee 854. **IEEE standard for radix-independent floating-point arithmetic**. New York, USA: Institute of Electrical and Electronics Engineers, 1987
- [21] KULISHI; MIRANKER, W. L.. **Computer Arithmetic in theory and practice**. New York, USA: Academic Press, 1981
- [22] Huckle, T. . **Collection of Software Bugs** . Disponível em: <<http://www5.in.tum.de/~huckle/bugse.html>>. Acesso em: ago. 2008
- [23] EVEN, G.; PAUL, W.J. . On the design of IEEE compliant floating point units. **Computers, IEEE Transactions on** . : v.49, p.398-413, 2000
- [24] QUACK, N.; KAKAQI, N.; FLYNN, M.. On Fast IEEE Roundign. **Technical Report: CSL-TR-91-459** . Stanford, CA, USA: 1991
- [25] SANTORO, M. R.; BEWICK, G.; HAROWITZ, M. A.. Rounding algorithms for IEEE multipliers. **Proceedings of 9th Symposium on Computer Arithmetic**. Santa Monica, CA, USA: p.176-183, 1989
- [26] ALLISON, C. . WHERE DID ALL MY DECIMALS GO. **J. Comput. Small Coll.**. USA: v.21, p.47-59, 2006
- [27] GOVINDU, G.; SCROFANO, R.; PRASANA, V. K.. A Library of Parameterizable Floating-Point Cores for FPGAs and Application to Scientific Computing. **Proc. Int'l Conf. Eng. Reconfigurable Systems and Algorithms (ERSA '05)**. : 2005
- [28] DOU, Y.; VASSILIADIS, S.; KUZMANOV, G. K.; GAYDADJIEV, G. N.. 64-bit floating-point FPGA matrix multiplication. **FPGA '05: Proceedings of the 2005 ACM/**

**SIGDA 13th international symposium on Field-programmable gate arrays.**

Monterey, California, USA: p.86-95, 2005

[29] GOVINDU, G.; ZHUO, L.; CHOI, S.; PRASANA, V.. Analysis of High-performance Floating-point Arithmetic on FPGAs. **18th International Parallel and Distributed Processing Symposium, 2004.** . USA: p.149-149, 2004

[30] Eclipse Foundation. **Eclipse Europa IDE.** Disponível em: <www.eclipse.org>. Acesso em: ago. 2008.

[31] MinGW community team. **Minimalist GNU for Windows.** Disponível em: <www.mingw.org>. Acesso em: ago. 2008.

[32] Mentor Graphics Inc.. . Disponível em: <www.model.com>. Acesso em: ago. 2008.

[33] Xilinx Inc.. . Disponível em: <www.xilinx.com>. Acesso em: ago. 2008.

[34] DOU, Yong; VASSILIADIS, S.; KUZMANOV, G. K.; GAYDADJIEV, G. N.. 64-bit floating-point FPGA matrix multiplication. **FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays.** Monterey, California, USA: p.86--95, 2005

[35] Xilinx Inc.. **Virtex-II Platform FPGAs.** Disponível em: <http://www.xilinx.com/support/documentation/data\_sheets/ds031.pdf>. Acesso em: ago. 2008.

[36] Altera Corporation. **Stratix III FPGAs.** Disponível em: <http://www.altera.com/products/devices/stratix-fpgas/stratix-iii/st3-index.jsp>. Acesso em: ago. 2008.

[37] Dadda, L. Some schmes for parallel multiplies. **Alta Frequenza.** Instituto di Elettrotecnica ed Electronica del Politecnico di Milano: p.349–356, 1965

[38] Avnet. **Xilinx Virtex II Development Kit.** Disponível em: <http://www.ee.ucla.edu/~herwin/ocdma/Avnet/Xilinx\_Virtex-IIDevelopmentKitDatasheet120302F.pdf>. Acesso em: ago. 2008.

