DESENVOLVIMENTO DE UM NÚCLEO ARITMÉTICO HÍBRIDO EM HARDWARE RECONFIGURÁVEL PARA IMAGEAMENTO SÍSMICO SEGUNDO O ALGORITMO RTM

Bruno Pessôa Neves

Dissertação de Mestrado

UNIVERSIDADE FEDERAL DE PERNAMBUCO
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE
2015

Universidade Federal de Pernambuco
Centro de Informática
Pós-graduação em Ciência da Computação

Bruno Pessôa Neves

DESENVOLVIMENTO DE UM NÚCLEO ARITMÉTICO HÍBRIDO EM HARDWARE RECONFIGURÁVEL PARA IMAGEAMENTO SÍSMICO SEGUNDO O ALGORITMO RTM

Dissertação de Mestrado apresentada à Universidade Federal de Pernambuco, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do Título de Mestre.

Orientador: Manoel Eusébio de Lima

RECIFE
2015

Catalogação na fonte
Bibliotecária Jane Souto Maior, CRB4-571

N518d Neves, Bruno Pessoa
Desenvolvimento de um núcleo aritmético híbrido em hardware reconfigurável para imageamento sísmico segundo o algoritmo RTM / Bruno Pessoa Neves – Recife: O Autor, 2015.
131 f.: il., fig., tab.

Orientador: Manoel Eusébio de Lima.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, 2015.
Inclui referências e anexo.

1. Engenharia da computação. 2. FPGA. 3. Computação científica. 4. Computação de alto desempenho. I. Lima, Manoel Eusébio de (orientador). II. Título.

621.39 CDD (23. ed.) UFPE- MEI 2015-185

Dissertação de Mestrado apresentada por **Bruno Pessoa Neves** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título "**Desenvolvimento de um Núcleo Aritmético em Hardware Reconfigurável para Imageamento Sísmico Segundo o Algoritmo RTM**" orientada pelo **Prof. Manoel Eusebio de Lima** e aprovada pela Banca Examinadora formada pelos professores:

_____

Profa. Edna Natividade da Silva Barros
Centro de Informática/UFPE

_____

Prof. Abner Corrêa Barros
Departamento de Estatística e Informática / UFRPE

_____

Prof. Manoel Eusebio de Lima
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 19 de agosto de 2015.

_____

Profa. Edna Natividade da Silva Barros
Coordenadora da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

Agradecimentos

Agradeço a minha família, que me deu suporte e apoio desde o início. A minha mãe, Gertrudes, e meu pai, Janôr, por sempre servirem como referência para meu aprendizado e me orientarem. Minhas irmãs, Amanda e Jéssica, pelo apoio, carinho e companheirismo. Minhas tias Giovana e Giovanilza pelo apoio, carinho e força.

Gostaria de agradecer ao professor Manoel Eusébio de Lima, que desde o início esteve presente em meu curso de graduação e pós-graduação. Agradeço pelo apoio e incentivo.

Agradeço a meus amigos da época de graduação João Paulo (JP) e Severino José (Biu). Ambos estiveram presentes desde o início do curso e também compartilharam dos percalços da pós-graduação e trabalho. Sou eternamente grato pela amizade, paciência e pelos conselhos amigos.

Gostaria de agradecer aos amigos que estão ou passaram pelo grupo HPCIn do Centro de Informática da UFPE, o qual estive presente durante grande parte da graduação e parte da pós-graduação. Nesse grupo ocorreu grande parte da pesquisa e dos testes necessários à realização deste trabalho. Também a Rodrigo Camarotti, ex-integrante do grupo e amigo, pelo apoio. Sou grato a Abner Barros pelo apoio e pelas aulas sobre aritmética de ponto flutuante.

Sou grato a todos os colegas de trabalho do grupo Lincs e demais membros do CETENE que também compartilharam, nos mais diversos graus, dos momentos que fizeram esta dissertação se concretizar.

Agradeço a meus amigos Marcelo, Gleybson e Elisvan pela força, amizade e paciência prestadas durante essa etapa. Também pelas trilhas ciclísticas percorridas com o grupo Polecats, fonte de muita inspiração para avançar em meus percursos.

Reservo um singular agradecimento a Nathalia Gaspar pelo auxílio, apoio e carinho a mim reservados durante a elaboração deste trabalho. Sou grato pela presença e constante companheirismo, minha querida.

Sou grato ao Cenpes e à Petrobras pelo incentivo a pesquisa feito junto ao projeto HPCIn do Centro de Informática da UFPE.

Agradeço a comunidade do Centro de Informática da UFPE, o qual foi objeto de anos de dedicação. Aos professores, funcionários, pesquisadores e demais integrantes do centro, os quais fizeram parte de minha formação acadêmica e humana durante essa importante etapa de minha vida.

Resumo

A computação de alto desempenho está presente em diversos setores do conhecimento humano. Ela busca atender a demanda por soluções para problemas em áreas como bioinformática, petroquímica, climatologia, dentre outros. Sabe-se que a grande maioria dessas áreas trabalha com quantidades massivas de dados, o que representa um desafio que a computação deve constantemente superar. Dentre algumas soluções atualmente adotadas, podemos citar os *Field Programmable Gate Array*s (*FPGAs*). Esses dispositivos permitem explorar a computação paralela com menor consumo de energia quando comparados a *Central Process Units* (*CPUs*) e *Graphic Process Units* (*GPUs*). Além disso, os *FPGAs* permitem explorar o reuso de dados, o que possibilita o desenvolvimento de arquiteturas computacionais mais eficientes. Essas características fazem dos *FPGAs* uma opção atraente para se desenvolver soluções para problemas que possuem uma alta demanda por processamento, como em aplicações científicas. Essas aplicações normalmente fazem uso massivo de números em ponto flutuante. Em 1977 o *Institute of Electrical and Electronics Engineers (IEEE)* propõe a criação do padrão IEEE-754 para a implementação da aritmética de ponto flutuante em base binária. No entanto, o padrão só foi concluído e lançado mais tarde, em 1985. Esse padrão numérico permite ao mesmo tempo tanto uma grande precisão, quanto uma grande capacidade de representação. O padrão IEEE-754 passou a ser seguido pelos fabricantes de computadores e desenvolvedores de software no tratamento da aritmética binária computacional. A indústria petrolífera faz uso massivo da aritmética de ponto flutuante para o mapeamento e geração de imagem das camadas do subsolo para detecção de poços de hidrocarbonetos. Um dos métodos de imageamento sísmico que tem apresentando melhores resultados em áreas com litologias mais complexas, tais como no pré-sal, é o algoritmo *Reverse Time Migration (RTM)*. Esse método faz uso de uma aproximação da equação de onda por meio dos operadores de diferenças finitas. Isso permite o mapeamento da variação dos campos de pressão e com isso se estimar as características litológicas das camadas em subsuperfície. Contudo, o custo do *RTM* é bastante elevado em termos computacionais. Por esse motivo, aplicações que otimizam desempenho ganham importância no cenário de mapeamento sísmico do subsolo realizado pelas indústrias petrolíferas. Esta dissertação aborda o desenvolvimento de um núcleo aritmético híbrido capaz de resolver a equação de diferenças finitas presentes no algoritmo de *RTM*, em *FPGA*. Foram desenvolvidos duas versões, uma totalmente em ponto flutuante padrão IEEE-754 e outra também com notação de ponto fixo para ganho de desempenho.

Palavras-chave: IEEE-754. HPC. RTM. FPGA.

Abstract

The high-performance computing is present in different sectors of human knowledge. It seeks to meet the demand for solutions to problems in areas such as bioinformatics, petrochemical, climatology, among others. It is known that the vast majority of these areas work with massive amounts of data, which is a challenge that the computational field should constantly overcome. Among some currently adopted solutions, we can mention the Field Programmable Gate Arrays (FPGAs). These devices allow exploit parallel computing with lower power consumption when compared to Central Process Units (CPUs) and Graphic Process Units (GPUs). Furthermore, FPGAs allow explore the data reuse, which enables the development of more efficient computing architectures. These characteristics make FPGAs an attractive option to develop solutions to problems that have a high demand for processing, such as in scientific applications. These applications typically make heavy use of floating point numbers. In 1977 the Institute of Electrical and Electronics Engineers (IEEE) proposes the creation of the IEEE-754 standard for implementing floating-point arithmetic in binary base. However, the standard was completed and released later in 1985. This numerical pattern allows the same time both a high precision, as a large capacity representation. The IEEE-754 standard then began to be followed by software developers and computer makers in the treatment of computer binary arithmetic. The oil industry makes massive use of floating-point arithmetic for mapping and generating image of the subsurface layers to detect hydrocarbon wells. One of seismic imaging methods that have presented better results in areas with more complex lithologies, such as the pre-salt, is the Reverse Time Migration algorithm (RTM). This method makes use of an approximation to the wave equation through the finite difference operator. This allows mapping the variation of pressure fields and thereby estimate the lithological characteristics of the layers in the subsurface. However, the cost of the RTM is computationally quite high. Therefore, applications that optimize performance gain importance in the underground seismic mapping scenario performed by the oil industry. This paper discusses the development of a hybrid arithmetic core able to solve the equation of finite differences present in the RTM algorithm in FPGA. Two versions, a fully floating point IEEE-754 standard and also with other fixed-point notation for performance gain were developed.

Keywords: HPC. IEEE-754. FPGA

Índice de Figuras

Lista de Tabelas

SUMÁRIO

Lista de Abreviaturas e Siglas

ALM	Adaptative Logic Module
ALUT	Adaptative Look-Up Table
ASIC	Application Specific Integrated Circuits
CENPES	Centro de Pesquisas e Desenvolvimento da Petrobras
CIN	Centro de Informática da UFPE
CLB	Configurable Logic Block
CPU	Central Process Unit
DSP	Digital Signal Processor
DUV	Device Under Verification
EEPROM	Electrically-Erasable Programmable Read-Only Memory
FIFO	Firs in, First Out
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Arrays
FPU	Floating Point Unit
GFLOPS	Giga Floating-Point Operations Per Second
GPU	Graphic Processing Unit
HDL	Hardware Description Language
HPC	High Performance Computing
HPCIn	High Performance Computing CIn-UFPE
IEEE	Institute of Electrical and Electronics Engineers
LUT	Look Up Table
MAC	Multiplier And Accumulator
MLAB	Memory Logic Array Block
NAN	Not A Number
OpenCL	Open Computing Language
PE	Process Element
PLL	Phase-Locked Loop
RAM	Random Access Memory
ROM	Read-Only Memory
RTL	Resistor–transistor logic
RTM	Reverse Time Migration
SNR	Signal-to-Noise Ratio

SPMD	Single Program Multiple Data
SRAM	Static Random Access Memory
UFPE	Universidade Federal de Pernambuco
UQI	Universal Quality Index
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
XOR	Exclusive OR

Capítulo

1

1. Introdução

Este capítulo contextualiza o tema abordado nesta dissertação, no campo da computação científica, e seu impacto no desenvolvimento de arquiteturas computacionais customizadas, em problemas que requeiram processamento aritmético com precisão e alto desempenho computacional, com ênfase a problemas de imageamento sísmico utilizando o algoritmo *RTM* (*Reverse Timing Migration)*.

A indústria petrolífera ainda é uma das mais atuantes e estratégicas como fonte energética em todo o mundo. Sua matéria prima, o petróleo, pode ser encontrada em diferentes áreas do planeta, seja em terra ou nas profundezas de nossos oceanos, com diferentes níveis de dificuldades de captação. Com a demanda crescente pelo consumo, custos oscilantes no mercado internacional, a necessidade por se dominar a tecnologia de prospecção deste minério se torna estratégica para países continentais como o Brasil. Esta tecnologia envolve não só equipamentos de perfuração e retirada de petróleo, mas sobretudo ferramentas computacionais que permitem, com precisão, estabelecer a localização correta destes reservatórios.

Particularmente no processo de descoberta de reservatórios de óleo e gás, através do uso de técnicas de imageamento e modelagem em sísmica, por exemplo, é possível se mapear com precisão, camadas geológicas de uma determinada região em busca destes componentes. Porém, para que este estudo detalhado do subsolo ocorra da melhor maneira possível, faz-se necessária a aquisição e a análise de uma grande quantidade de dados, com qualidade e em tempo hábil.

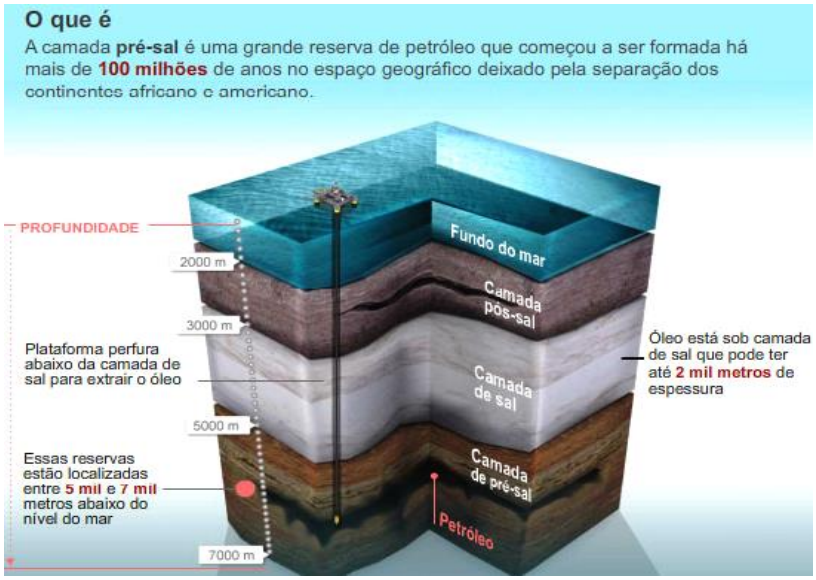
O tratamento inadequado de imagens sísmicas pode promover a perfuração de poços em regiões equivocadas, acarretando um prejuízo de milhões de dólares para as empresas exploradores. O fator tempo também é importante, desde que a inclusão de um novo poço na linha de produção garante a oferta e a estabilidade do produto no mercado. Portanto, a precisão com que os dados são tratados e o poder computacional para processá-los, tornam-se fatores cruciais para o desenvolvimento da indústria e da exploração de óleo e gás.

Com a demanda cada vez maior pelo mercado consumidor, a ser suprida por combustíveis fosseis, tornou-se necessário a busca por poços cada vez mais distantes, em áreas com uma geologia, muitas vezes, bem mais complexa que as até então exploradas. No Brasil, por exemplo, tal fenômeno aconteceu com a descoberta de depósitos de hidrocarbonetos como a do pré-sal[1], feitas pela Petrobras (2015), a profundidades de até 7000m. Tal descoberta, embora parecesse resolver todos os problemas de abastecimento, exigia no entanto, uma série de investimentos, seja em equipamentos de prospecção, seja na forma de tratar matematicamente os novos modelos geológicos que agora se faziam presentes. Tinha-se camadas geológicas bem diferentes das dos poços anteriores; camadas espessas de sal, formações rochosas irregulares, etc. Como tratar tudo isso de maneira precisa e na velocidade desejada de mercado foi sem dúvida o grande desafio para empresas como a

[1] O pré-sal é uma sequência de rochas sedimentares formadas há mais de 100 milhões de anos sob uma camada de sal rica em hidrocarbonetos (petróleo e gás natural). PETROBRAS. 2015.

Petrobras (PETROBRAS, 2015). A figura 1 ilustra simplificadamente, um corte geológico com as camadas de subsuperfície mais externas até a camada do pré-sal.

Figura 1 - Ilustração das camadas geológicas em subsuperfície até a camada pré-sal.



Fonte: G1, 2010.

Do ponto de vista de investimento e de mercado, não adianta ter-se equipamentos sofisticados de perfuração de poços, se não há uma precisão adequada de onde estão estes poços comercialmente interessantes, que garantem retorno financeiro adequado. Neste sentido é fundamental o desenvolvimento de algoritmos de modelagem sísmica que garantam com precisão as posições a serem consideradas na perfuração destes poços.

1.1. Motivação

A modelagem sísmica é um processo computacional, desenvolvido a partir de algoritmos complexos que, em geral, requer o processamento de uma grande quantidade de dados. Tal característica a torna uma operação computacionalmente custosa, exigindo expertise no tratamento, precisão e processamento destes dados, para se obter uma análise consistente de seus resultados.

Quanto mais precisa a operação, mais probabilidade de se encontrar um novo poço promissor para exploração. Quanto mais recursos computacionais mais possibilidade de se refinar o processo, atacar maiores áreas de exploração e de visualização do terreno e reduzir tempo de processamento.

Como dito acima, a medida que se vai explorando regiões mais profundas, como a que encontramos na região do pré-sal, novas abordagens precisam ser feitas para se garantir precisão nos resultados da modelagem. Observou-se que algoritmos de modelagem existentes, que processavam de forma adequada terrenos de menor profundidade e com uma geologia mais regular que o do pré-sal, não eram adequados ao processamento deste novo desafio. Entre os algoritmos tradicionais utilizados na modelagem sísmica podemos destacar: *Pre-stack Kirchhoff Time Migration* (PSTM), *Phase Shift Plus Interpolation* (PSPI) e o *Reverse Time Migration* (RTM) (BARROS, 2014).

O algoritmo *PSTM* permite o imageamento de camadas abaixo da superfície com uma qualidade aceitável e a um custo computacional bem mais acessível que o algoritmo *RTM*. Esse custo de execução é cerca de 15 vezes menor que o *RTM* (SANTOS, 2012). No entanto, os algoritmos baseados em *Kirchhoff* não são adequados para regiões que possuem estruturas geológicas muito complexas (PGS, 2007).

O algoritmo *RTM*, por outro lado, é o algoritmo que tem apresentado os melhores resultados em regiões de litologia mais complexa, como no pré-sal, razão pela qual vem sendo amplamente utilizado na exploração de petróleo nestas regiões (SANTOS, 2012) (MEDEIROS, 2013). Este algoritmo resolve diretamente a equação da onda acústica/elástica, com resultados bem mais precisos em terrenos complexos (ROCHA, 2010). Seu processamento requer, no entanto, uma alta capacidade computacional e de armazenamento de dados, bem superior aos demais modelos. Graças a evolução e oferta de supercomputadores e processamento massivo em *grids*, este tipo de abordagem em modelagem sísmica, seja 2D ou 3D se torna uma realidade. Uma discussão detalhada deste algoritmo é realizada no capítulo 2 desta dissertação.

A computação científica de alto desempenho vem cada vez mais sendo explorada e difundida em diferentes tecnologias e estilos de projeto. Nesse tipo de computação, arquiteturas totalmente customizadas como *ASICs* (Smith, 1997), programáveis, como Multi-core (Intel, 2015) e *GPUs (*NVIDIA, 2015) e reconfigurareis como *FPGAs (*NAVABI, 2007), encontram grande espaço para aplicações, cada qual com sua características, vantagens e desvantagens. Dentre elas, a tecnologia *FPGA* tem ganho destaque, devido a sua capacidade de reconfiguração, exploração de paralelismo e possibilidade de desenvolvimento de núcleos de processamento, que realizam operações customizadas, e de alto desempenho.

Particularmente, com relação ao uso dos *FPGA*s para o processamento de dados sísmicos, diversos trabalhos publicados nestes últimos anos têm buscado destacar suas vantagens. Esses trabalhos destacam os benefícios no desempenho de execução de algoritmos

e no consumo de energia, pela adoção de padrões especiais na representação numérica de dados (BARROS et al., 2011; MEDEIROS et al., 2013; BRAGANÇA et al., 2013). Isso demonstra a relevância que a representação numérica adotada possui para a computação de alto desempenho.

A representação numérica em geral adotada em problemas de modelagem em sísmica é aquela presente na notação em ponto flutuante, segundo o padrão IEEE-754. Esse padrão determina como são operados e representados os números reais em sistemas computacionais. Ele ficou conhecido como padrão de números em ponto-flutuante devido à maneira como esses números são representados (BARROS, 2008).

No entanto, não só as operações com números em ponto flutuante encontram espaço nos sistemas computacionais. Números em ponto fixo e suas operações aritméticas também podem ser utilizados a depender das restrições de precisão e custo computacional. Isso ocorre porque algumas operações aritméticas em notação de ponto fixo são menos custosas quando comparadas a operações em ponto flutuante.

De fato, pode-se ainda, visando requisitos tais como: redução de área física de elementos de processamento, dissipação de potência, precisão customizada, desenvolver elementos de processamento híbridos, customizados, com o melhor das características das notações aritméticas acima mencionadas.

Neste contexto, tecnologias reconfiguráveis, como as fornecidas pelas *FPGAs*, permitem que se tenha, para cada nova abordagem do usuário, uma nova configuração customizada para seus elementos de processamento.

Este novo paradigma é completamente diferente daqueles encontradas em tecnologias multi-core, em *CPUs* de propósito geral, ou em *GPUs*, onde estes possuem núcleos aritméticos com funcionalidades fixas, com diversos tipos de operações aritméticas diferentes, previamente implementadas, sem customizações programáveis em hardware.

Fazer uso de unidades aritméticas customizadas para atender um problema específico permite, em um primeiro momento, reduzir-se a quantidade de componentes de hardware que são usados para sua implementação. No caso de núcleos aritméticos desenvolvidos para um algoritmo específico, em *FPGA*, provoca a economia de hardware por núcleo e por consequência, a geração de uma maior quantidade de núcleos por dispositivo. Isso aumenta a quantidade de operações que podem ser feitas em paralelo e, por conseguinte, o desempenho computacional final do algoritmo.

Este aumento no número de núcleos de processamento pode acarretar porém, alguns problemas não desejáveis, como o limite de acesso a memória disponível, etc. Este aspecto,

além da análise do consumo de energia dos elementos de processamento não são abordados nesta dissertação.

O estudo das notações aritméticas de ponto flutuante e de ponto fixo, associado à possibilidade de se desenvolver um hardware customizado, permitiu a implementação de um núcleo aritmético específico, em hardware, utilizando a tecnologia *FPGA*, capaz de resolver a equação de modelagem sísmica 3D, pelo método *RTM*. Otimizações nesses núcleos levaram em conta o processamento de dados em ambas as representações aritméticas citadas acima, sem, no entanto, acarretar perda de precisão para o problema da sísmica. Este esforço resultou na redução de área ocupada por cada núcleo aritmético, e consequentemente, um maior número de núcleos pode ser alocado no *FPGA*. Esta estratégia permitiu um maior paralelismo na execução do algoritmo e redução de tempo em sua execução.

1.2. Objetivos desta dissertação

O objetivo desta dissertação é o desenvolvimento de um núcleo aritmético híbrido capaz de calcular a aproximação da equação de onda utilizada no algoritmo de *RTM*, abordada em (Medeiros, 2013), em um *FPGA*. Esse núcleo deve trabalhar com números em formato de ponto flutuante padrão IEEE-754.

A fim de comparar desempenho, foram desenvolvidos dois núcleos aritméticos. O primeiro deles realiza operações aritméticas de acordo com as normas definidas para o ponto flutuante no padrão IEEE-754. A segunda versão, a versão híbrida do núcleo aritmético, efetua internamente operações na notação de ponto flutuante e em notação de ponto fixo. As versões serão comparadas tanto em relação ao custo de ocupação de elementos lógicos disponíveis para *FPGA*, quanto à precisão dos resultados numéricos apresentados. A versão híbrida deve obedecer aos critérios de precisão exigidos para o algoritmo e, com isso, ser capaz de substituir a versão em ponto flutuante no algoritmo de *RTM* sem perda de precisão significativa[2].

Os núcleos aritméticos, em suas duas versões, foram implementados na arquitetura proposta pelo projeto HPCIn, do Centro de Informática da UFPE, em parceria com o Centro de Pesquisa e Desenvolvimento da Petrobras (CENPES). O projeto HPCIn teve como objetivo, gerar imagens das camadas do subsolo para exploração de óleo e gás através da execução do algoritmo de análises sísmicas *RTM 3D*. A arquitetura proposta foi

[2] A precisão definida para o modelo do algoritmo RTM exigido pelo Centro de pesquisa da Petrobrás (CENPES) no projeto HPCIN, do Centro de Informática da UFPE foi da ordem de 10^{-6}

implementada em uma linguagem de descrição de *hardware, System Verilog* () para dispositivos reconfiguráveis, *FPGA*s.

1.3. Estrutura da Dissertação

Inicialmente, no capítulo 1, foi feita uma contextualização do tema da dissertação e discutidos os tópicos que motivaram seu desenvolvimento. No capítulo 2 são abordados os fundamentos teóricos necessários para a compreensão deste trabalho. O capítulo 3 é dedicado à discussão de trabalhos relacionados ao tema do projeto, suas vantagens, desvantagens, limites e comparações. No capítulo 4 é apresentado em detalhes as arquiteturas das duas versões de núcleos de processamento; a totalmente implementada na versão padrão ponto-flutuante e a versa híbrida. Neste capítulo também são discutidos todos os módulos e sub-módulos dos núcleos aritméticos, metodologia de testes e desenvolvimento, além dos resultados de síntese para uma *FPGA* que foi usada para validação destes núcleos aritméticos. No capítulo 5 é apresentado o caso de uso no qual os núcleos aritméticos foram utilizados, a saber: a modelagem sísmica 3D utilizando o algoritmo *RTM*. O capítulo 6 apresenta as conclusões e futuros trabalhos. As referências bibliográficas são elencadas no capítulo 7. O capítulo 8 corresponde aos anexos do projeto, que dizem respeito as estruturas dos módulos da arquitetura.

Capítulo

2

2. Fundamentação Teórica

Esta seção discute os conhecimentos básicos que nortearam a elaboração deste trabalho. Aqui foi realizada uma revisão sobre computação de alto desempenho com ênfase nas aplicações com *FPGAs*, na plataforma de desenvolvimento utilizada. Também é abordada a padronização aritmética e o formato de ponto flutuante utilizado no módulo de processamento.

2.1 Computação de alto desempenho em Hardware Reconfigurável.

Dispositivos capazes de realizar computação estão cada vez mais presentes em nosso cotidiano. Esses dispositivos são gradativamente mais exigidos em termos de capacidade de processamento com o avanço natural da tecnologia e suas aplicações. Essa tendência se mantém também quando falamos em computação de alto desempenho.

A demanda por maior velocidade e eficiência em aplicações com operações massivas de dados exige cada vez mais dos equipamentos que as realizam. A computação de alto desempenho está diretamente relacionada com as aplicações que requerem máquinas com alto poder computacional e memória, tais como: biotecnologia, climatologia, economia, indústria aeronáutica, petroquímica, processamento gráfico, entre outras.

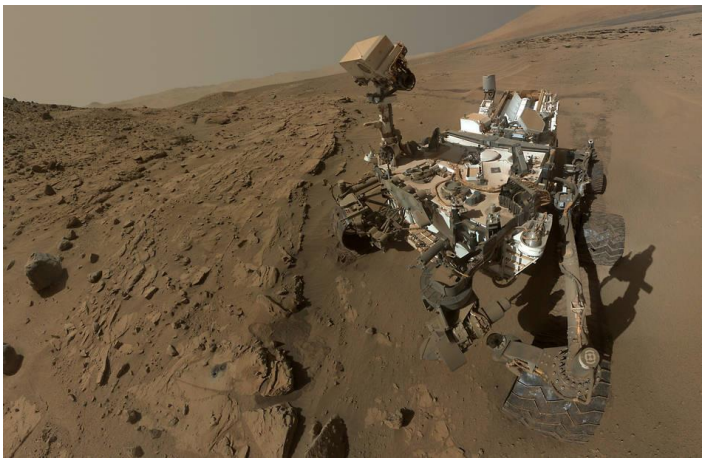
Várias arquiteturas têm sido utilizadas no desenvolvimento de soluções visando processamento de alto desempenho. Pode-se destacar os dispositivos lógicos reconfiguráveis, os *FPGAs* como uma arquitetura voltada para exploração de alto desempenho.

Os *FPGAs* possuem caráter intrinsecamente paralelo e permitem explorar uma grande capacidade computacional. Essas duas características, juntamente com a baixa frequência em que operam, permitem explorar diversas aplicações a um baixo custo energético. Contudo, o desenvolvimento de aplicações em *FPGA* ainda é custoso em termos de tempo de desenvolvimento, muitas vezes se tornando incompatível com a necessidade da indústria (MEDEIROS, 2013).

De maneira similar aos *FPGAs*, os *Application Specific Integrated Circuits (ASIC)* implementam as funções lógicas como componentes diretamente implementadas em hardware (SMITH, 1997). Os *ASICs* são dispositivos desenvolvidos e otimizados para apenas um único propósito. Em geral, estes dispositivos possuem um custo por unidade menor quando feitos em larga escala e podem atingir frequências de operação mais altas, ou seja, permitem uma maior velocidade na execução de uma tarefa específica. Eles podem ser utilizados para alguns problemas em computação de alto desempenho, levando-se em consideração que são otimizados para a execução de um conjunto específico de tarefas. Porém, justamente por serem voltados para a execução de um conjunto restrito de operações, não possuem um mínimo de flexibilidade que muitas aplicações computacionais exigem. Como exemplo de uma aplicação onde os *ASIC´s* possuem seu uso limitado, pode-se citar um veículo de

exploração de Marte denominado *Curiosity* da NASA[3]. Esse veículo possui uma demanda por alterações de programação e atualizações em nível de *hardware* mesmo depois de desembarcado em solo marciano, o que se tornaria impossível com o uso de *ASIC´s*. Com isso, o veículo foi equipado com *FPGAs* da Xilinx que podiam ser reprogramadas de acordo com a necessidade da missão (XILINX, 2015). A figura 2 a seguir mostra o veículo de exploração marciana *Curiosity*.

Figura 2 - Veículo de exploração marciana com dispositivos reconfiguráveis embarcados.



O *FPGA* permite o melhor dos dois mundos, *ASIC´s* e *CPU´s*. Eles podem ser ajustados para atender demandas específicas e também podem ser modificados para atender às variações das aplicações. Eles possuem um tempo de desenvolvimento e um *time-to-market*[4] menor quando comparado a projetos de *ASICs* (DUTRA, 2010).

Além disso, conseguem explorar o uso de operações de maneira simultânea com baixo consumo de energia quando comparados a *CPU´s*, o que provê uma boa solução para a eficiência computacional (MEDEIROS, 2013).

Novas iniciativas para desenvolvimento de padrões de linguagens que permitem explorar alto desempenho estão em andamento, como a *Open Computing Language*[5]

[3] *National Aeronautics and Space Administration* (NASA) é o órgão do governo dos Estados Unidos responsável pelo programa espacial civil, bem como aeronáutico além da investigação aeroespacial. NASA. Acesso em 31 ago. 2015. Disponível em <https://www.nasa.gov/>.

[4] Corresponde a quantidade de tempo que leva para se projetar e fabricar um produto antes que ele esteja disponível para a compra. Dictionary Cambridge. Acesso em: 31 ago. 2015. Disponível em: <http://dictionary.cambridge.org/pt/dicionario/ingles/time-to-market>.

[5] É um padrão multi-plataforma para programação paralela de processadores encontrados em computadores pessoais, servidores e dispositivos que melhora a velocidade e capacidade de resposta para um amplo espectro de aplicações. OpenCL. Acesso em: 29 jul. 2015. Disponível em: <https://developer.nvidia.com/opencl> .

(OpenCL). Esse padrão busca o desenvolvimento de uma linguagem aberta para multi-plataformas, incluindo-se *CPUs*, *GPU* e *FPGAs* (OPENCL, 2015).

2.2 FPGAs

Os *FPGAs* foram lançados no ano de 1985 pela Xilinx e podem ser descritos como dispositivos lógicos reconfiguráveis implementados em *hardware*, capazes de serem reprogramados de acordo com as necessidades do desenvolvedor (XILINX, 2015). Isso é possível devido à maneira como sua estrutura foi concebida, abrigando elementos de memória, de entrada e saída, bem como de lógica combinacional e sequencial.

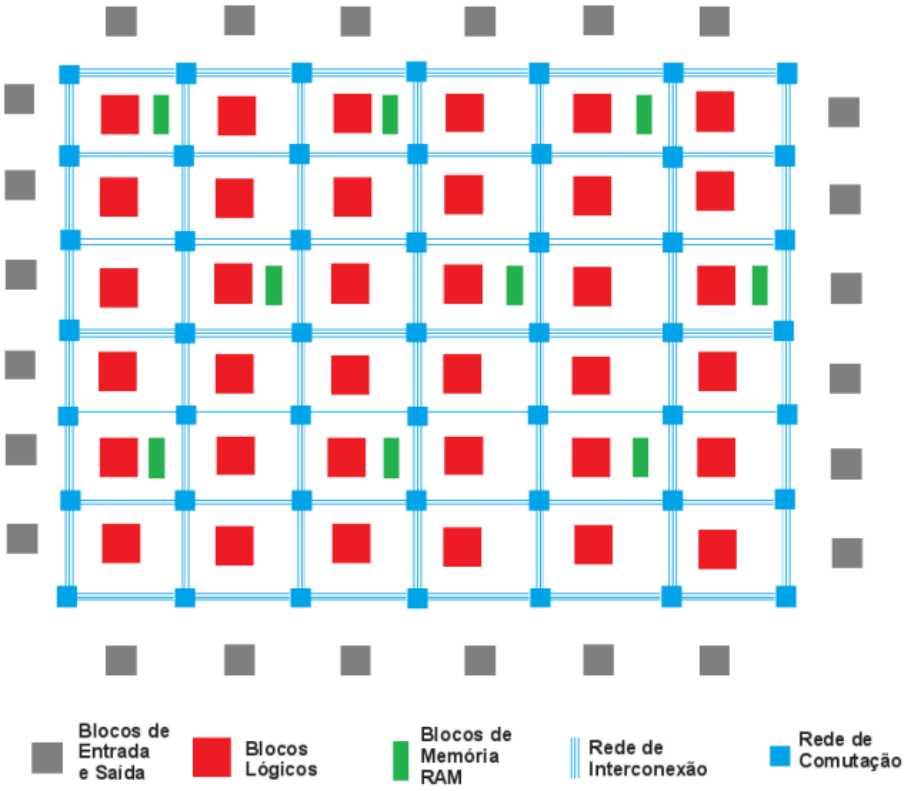
De maneira geral, uma *FPGA* moderna é formada por um array bi-dimensional de elementos computacionais. Esses elementos correspondem a blocos de lógica combinacional (*CLBs*) e sequencial (*Flip-Flops*), elementos de entrada e saída, bancos de memória, blocos DSP, *transceivers* de alta velocidade, blocos de *PLLs*, entre outros.

Além disso, uma *FPGA* possui elementos de roteamento que permitem a conexão adequada de todos estes componentes. Esses elementos tem sua funcionalidade determinada por um conjunto de bits que define como eles devem ser interconectados. A esse conjunto de bits dá-se o nome de *bitstream.* A figura 3 exibe alguns desses componentes que estão presentes no dispositivo.

Os *FPGAs* possuem algumas características que, a depender da aplicação, são tidas como vantagens sobre processadores de propósito geral. Pode-se destacar, por exemplo, que os *FPGAs* podem ser usados para aplicações customizadas dedicadas, ou seja, são otimizados para a resolução de um problema específico. Isso faz com que toda a estrutura presente no dispositivo esteja voltada apenas para a execução das tarefas para o qual foi programada. O mesmo não pode ser dito de um processador de propósito geral, pois esse implementa diversas instruções que podem não ser utilizadas por uma aplicação específica.

Outra vantagem significativa sobre os processadores de propósito geral é sua operação em frequências baixas, quando comparadas às aplicadas as *CPUs*, permitindo, assim, um baixo consumo de energia, o que, para grandes clusters de computadores, implica em uma economia considerável de energia (MEDEIROS et al., 2013).

Figura 3 – Visão geral da arquitetura de uma FPGA

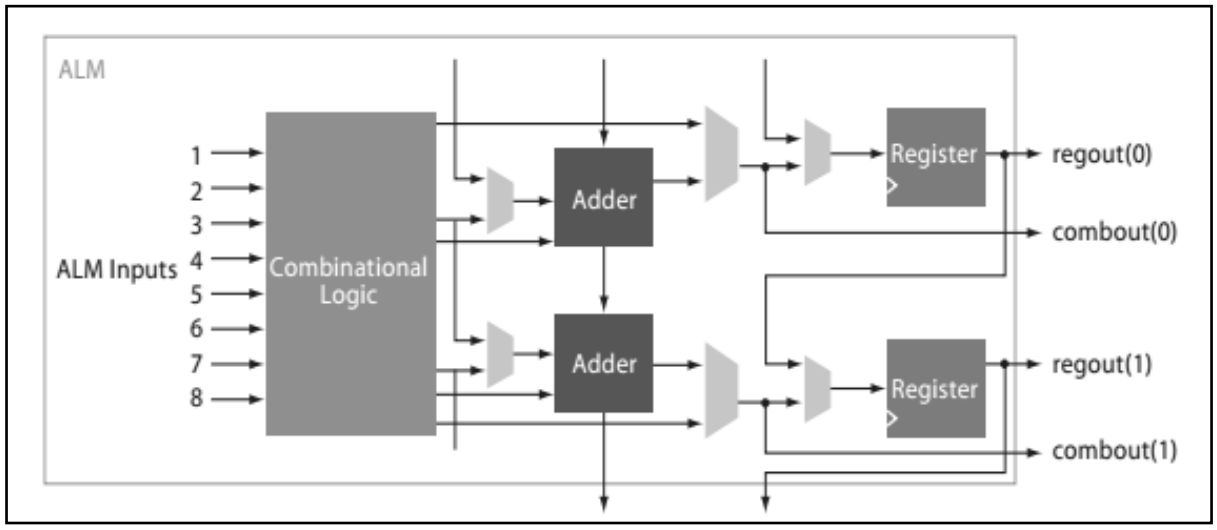


Os blocos lógicos possuem uma região para a implementação da lógica combinacional, bem como possuem *latchs* ou *flip-flops* para implementação de lógica sequencial (XILINX, 2015). A saída dos blocos lógicos pode ser selecionada entre o da lógica combinacional e o da lógica sequencial, vinda do *flip-flop* por um multiplexador de saída. Pode-se observar uma estrutura de um bloco lógico *ALM* (Adaptive Logic Module) da *Altera* na figura 4 (ALTERA, 2015).

A maneira mais comum de implementar a lógica combinacional é através do uso de *LUTs* (*look-up table*s), abordado em Altera (2015), que, conceitualmente, pode ser explicada como sendo um grande multiplexador para células de memória. Uma *LUT* é, basicamente, uma memória pré-programada que provê uma saída lógica em função do valor do vetor das variáveis de entrada. Com isso, as *LUTs* operam como uma memória de N linhas de endereço para 2^N locais de memória. As *LUTs*, por usarem tecnologia de memórias *Randomic Access Memory* (RAM), são voláteis, perdem seu conteúdo em caso de falta de energia elétrica. No entanto, é possível armazenar informações de programação do dispositivo em memórias não voláteis. Isso é feito utilizando-se, por exemplo, memórias *FLASH EEPROM* (*Electrically Erasable Programmable Read Only Memory*), capazes de carregar automaticamente as células de armazenamento ao início de sua utilização.

Os blocos lógicos da Altera, os *ALMs* possuem duas unidades somadoras que podem atuar como somadores de dois bits (ALTERA, 2015). Além disso, possuem dois registradores na saída, responsáveis pelo armazenamento dos estados dos sinais, ou seja, a parte da lógica sequencial. Para selecionar os caminhos e sinais dentro dos *ALMs*, existem diversos Multiplexadores configurados de acordo com a necessidade de uso do bloco.

Figura 4 - Estrutura de um bloco lógico ALM em um FPGA

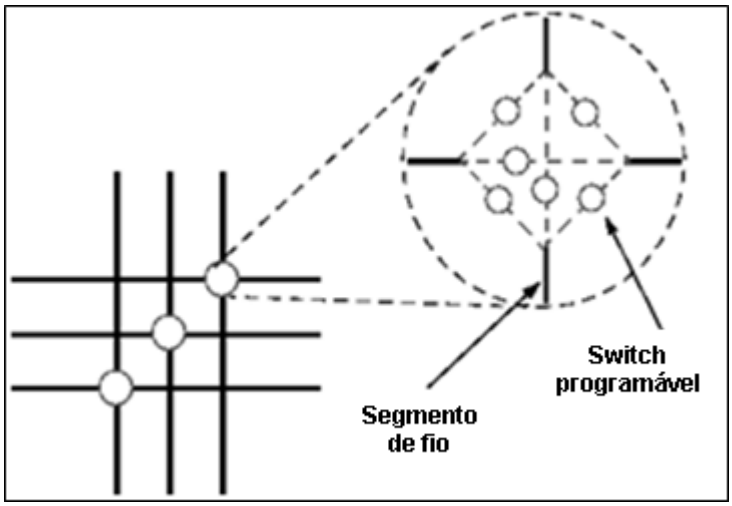


Outros módulos de interesse presentes em *FPGAs* Altera são os *DSP (*Digital Signal Processing), visto em Altera (2015). Os blocos *DSP* consistem em uma combinação de elementos dedicados que realizam multiplicação, adição, subtração, acumulação e operações dinâmicas de deslocamento (*shift)*. Esses blocos podem ser utilizados para operar números em ponto fixo. Como exemplo, podemos citar uma *FPGA StratixIII* da Altera (2015), onde seus *DSPs* dão suporte nativamente a tamanhos de palavras de 9 bits, 12 bits, 18 bits e 36 bits, além de multiplicadores de 18x18 bits, o que otimiza a realização de operações aritméticas.

Os *FPGAs* possuem elementos de roteamento chamados de *Switch Matrix*. Eles estão dispostos em forma de linhas e colunas compondo uma grande matriz de trilhas que percorrem toda a *FPGA*. Eles podem ser utilizados para a definição do roteamento de sinais pelas linhas horizontais e verticais durante a programação do circuito. Na figura 5, podemos visualizar um elemento de roteamento presente na intersecção das trilhas, chamado de matriz de roteamento. Ele possui seis chaves programáveis que, quando ativadas, permitem a passagem de corrente de uma trilha para outra. De acordo com o vetor de bits de configuração, as seis chaves podem assumir diversas combinações diferentes, provendo

versatilidade no roteamento dos sinais. Dessa maneira, é possível a propagação de um sinal de uma região para outra e se torna possível a comunicação entre blocos lógicos distantes.

Figura 5 - Matriz de roteamento, *Switch Matrix*.



Os *FPGAs* Altera possuem blocos de memória *TriMatrix* incorporados, fornecendo três tamanhos diferentes de memória *Static Randomic Access Memory* (RAM) embutidas: MLAB, M9K e M144k. As memórias MLAB são otimizadas para se implementar linhas de atraso de filtros, FIFOs pequenas e registradores de deslocamento. Os blocos M9K podem ser utilizados para aplicações de memória para fins gerais. Os blocos M144K são ideais para armazenamento de código de processamento e armazenamento de frames de vídeo. Cada bloco de memória embutida pode ser configurado independentemente para atuar como um *dual-port RAM* ou *ROM*. Além disso, vários blocos do mesmo tipo também podem ser combinados para se produzir memórias maiores. Essas memórias do tipo *SRAM* podem operar a uma frequência máxima de 600 MHz, o que representa uma ótima frequência em se tratando de *FPGAs* (ALTERA, 2015).

2.3 Aritmética de Ponto Flutuante e Ponto Fixo

Devido à presença dos números reais na computação científica e sua grande diversidade de representação, que variava conforme o fabricante no início da década de oitenta do século vinte, tornou-se necessária a criação de um padrão. Esse padrão formalizou um modelo de representação de dados a ser seguido pelos desenvolvedores de processadores.

Processadores esses que realizavam operações aritméticas sobre o conjunto dos números reais em computadores.

Com o desenvolvimento do co-processador aritmético i8087 em 1980, a *Intel* conseguiu atender a todos os requisitos de confiabilidade e portabilidade exigidos pelos programadores da época, (BARROS, 2008).

Esse componente realizava várias operações aritméticas tais como: soma, subtração, divisão, multiplicação, raiz quadrada, tangente, exponencial, entre outras. Atualmente, esse papel foi ocupado pelas *Floating Point Units* (FPUs) integradas a quase todos os processadores.

Após a formalização de um padrão para aritmética dos números reais em computadores, a computação científica ganhou replicabilidade e mais confiabilidade em seus resultados.

A seguir, será apresentado o padrão seguido pela comunidade científica e industrial para ponto flutuante com sua respectiva representação de dados e de ponto flutuante.

2.3.1 Padrão IEEE-754

Criado em 1977 e lançado em 1985, o padrão IEEE-754 surgiu para tentar solucionar os problemas causados pelas diferentes formas de representação e operação de números reais em sistemas computacionais (IEEE-754, 1985).

O IEEE-754 definiu os requisitos mínimos a serem seguidos para implementação aritmética desses números em uma base binária, padrão que se tornou o mais usado pelos computadores até os dias atuais.

Dentre seus requisitos têm-se:

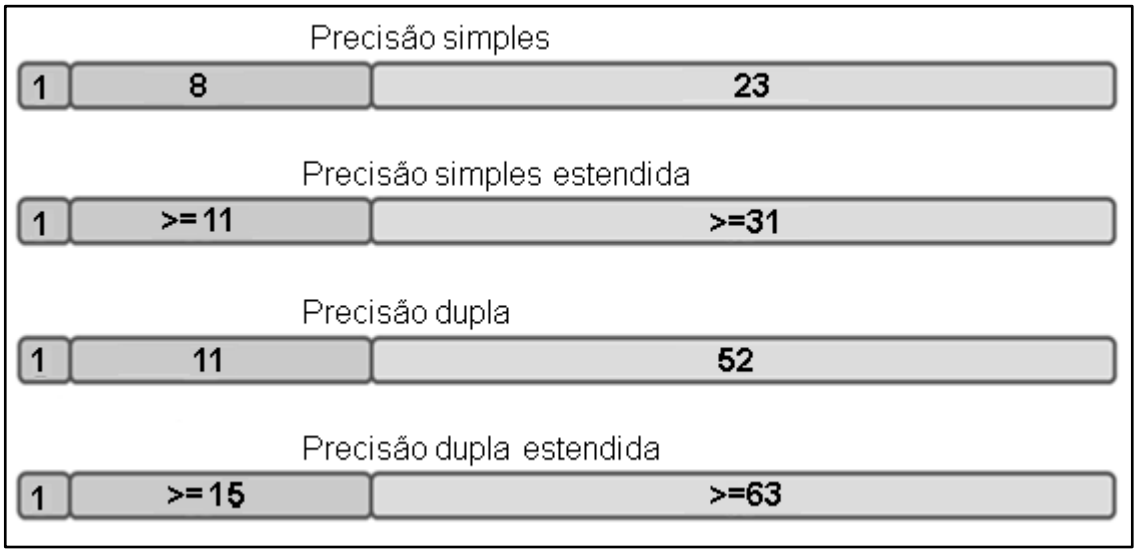
- **Formato de representação dos dados:** define quatro tipos de representação para dados, denominados *Double*, *Float, Double Extended e Float Extended.*
- **Operações aritméticas:** Devem constar as operações aritméticas de soma, subtração, multiplicação, divisão e raiz quadrada para os tipos de dados suportados em uma implementação completa do padrão.
- **Arredondamento:** Estão definidos quatro tipos de arredondamentos para as operações aritméticas: arredondamento para o mais próximo ou par (*round to nearest or even*); arredondamento para o zero (*round to zero*); arredondamento para infinito

positivo (*round to +infinity*); e arredondamento para infinito negativo (*round to –infinity)*.

- **Normalização:** Garante a coerência dos dados gerados de operações aritméticas com o próprio padrão através do ajuste dos vetores de bits, para garantir a unicidade da sua representação. Assegura que o digito mais significativo deve ser diferente de zero.
- **Exceções:** Define as exceções geradas durante o processo de soma/subtração, multiplicação e demais operações. As principais exceções definidas são: *Underflow*, *Overflow*, *NAN* (números inválidos), Infinito positivo, Infinitivo negativo e denormal.

Os números são representados como vetores de bits separados em três subconjuntos de acordo com suas características: sinal, expoente e mantissa. A extensão desse vetor depende do tipo de precisão adotada. O primeiro termo corresponde ao sinal, o segundo ao expoente e o terceiro a mantissa. Podemos observar a representação e o número de bits de cada termo dos números em ponto flutuante de acordo com sua precisão na figura 6.

Figura 6 - Representação de números em ponto flutuante padrão IEEE-754.



2.3.2 Formato de representação dos dados

Um número no padrão IEEE-754 obedece à seguinte regra de definição de parâmetros:

- **p** = Número de bits do significando.
- **Emáx** = Expoente máximo.
- **Emin** = Expoente mínimo.
- **Bias** = Valor de referência do expoente.

Dessa maneira, pode-se representar um número em ponto flutuante utilizando-se a seguinte expressão:

$$-1^{s} * \text{d.ddd} \cdots \text{d} * \beta^{e} \qquad (2.1)$$

- s : Bit de sinal.
- d.ddd...d : Os p dígitos do significando.
- β : Representa a base numérica. No caso do padrão IEEE-754 a base adotada é binária, portanto $\beta = 2$.
- e : Representa o expoente do número na base adotada. Seu valor está contido na região definida por E_{min}[6] $\leq e \leq E_{max}$[7]

2.3.3 Notação de Ponto Fixo

Nesse tipo de notação, busca-se representar tanto a parte inteira do número como sua parte fracionária. Dessa maneira, os dígitos utilizados para a representação se dividem em dois grupos separados por um "ponto separador". O ponto separador serve apenas para manter o padrão de notação dos números reais, que possuem parte inteira e parte fracionária. Os dígitos a esquerda do ponto correspondem à parte inteira do número, enquanto que os dígitos da direita representam a parte fracionária.

Para saber o valor numérico representado, basta realizar um somatório de cada dígito ponderado pela base, isso de acordo com o posicionamento relativo do dígito em sua representação. Assim, se obtém a seguinte fórmula para representação:

$$\pm \sum_{i=1}^{n} di * \beta^{i-1-k} \qquad (2.2)$$

[6] Corresponde ao expoente mínimo representado.
[7] Corresponde ao expoente máximo representado.

Nesta fórmula, **n** representa o número de dígitos, sendo **k** dígitos reservados para a representação da parte fracionária. Na notação utilizada, **d** representa o número de dígitos e **di** o dígito de posição **i.** O β representa a base adotada, no caso a binária.

A precisão de um número nessa notação depende da quantidade de bits disponíveis para representa-lo. Quanto maior a quantidade de bits disponíveis, maior a precisão, quanto menor a quantidade de bits disponíveis, menor a precisão.

Para se chegar ao nível de precisão a ser utilizado na notação de ponto fixo, foi feito um estudo do impacto da quantidade de mínima de bits necessários à operação do *RTM* em *FPGA*. Essa quantidade não podia levar a perdas significativas de qualidade do resultado do algoritmo. Como o *RTM* é utilizado para o processo de imageamento sísmico, seu resultado característico final é justamente uma imagem das camadas do solo a ser estudado.

O processo de comparação de imagens no *RTM*, apesar de levar em conta medidas como *Signal to Noise Ratio (SNR)*[8] e o *Universal Quality Index (UQI)*[9], proposta em Wang (2002), ainda apresenta um forte caráter subjetivo, a interpretação humana. Com isso, não há uma definição clara do nível de precisão numérica que os algoritmos sísmicos exigem (FU et al., 2009).

A pesquisa dessa dissertação fez parte de um projeto de pesquisa mais amplo, firmado entre o Centro de informática da UFPE e a Petrobras, com isso, a palavra final para o nível de qualidade de imagem foi dada pelos pesquisadores do próprio CENPES. A referência adotada para qualidade de imagem é dada pelo algoritmo *RTM*, utilizando um núcleo de processamento em ponto flutuante.

A representação mínima em ponto fixo que apresentou os resultados mais próximos dos resultados obtidos com a representação em ponto flutuante foi a com 9 bits na parte inteira e 27 bits na parte fracionária. A representação adotada pode ser observada na figura 7 a seguir.

Figura 7 - Representação adotada em Ponto Fixo.

[8] A relação SNR é uma relação adimensional da potência do sinal com a potência do ruído. Essa relação sinal - ruído parametriza o desempenho dos sistemas de processamento de sinais ideais. Scholarpedia. Acesso em 14 set. 2015. Disponível em: <http://www.scholarpedia.org/article/Signal-to-noise_ratio>.

[9] Métrica que busca mensurar o grau de semelhança entre imagens do ponto de vista da percepção humana do brilho, contraste e artefatos presentes nas imagens (BARROS, A. B).

Com essa notação, é possível representar números que variam de 255 a - 255 devido aos 9 bits da parte inteira, com precisão de cerca de 2^{-27}, que equivale a, aproximadamente, $7{,}45x10^{-9}$.

2.3.4 Algoritmos de Soma e Subtração em Ponto Fixo

Na representação em ponto fixo, os algoritmos de soma e subtração são mais simples dos que os realizados em ponto flutuante. Como não há a presença de um expoente na representação do número, uma soma ou subtração direta entre o conjunto inteiro de bits pode ser aplicada.

Para se realizar essa operação, é importante que o ponto separador entre a parte inteira e a parte fracionária dos dois operandos estejam alinhados. Isso implica que bits com os mesmos valores de frações de dois são operados entre si. Caso contrário, valores em partes do vetor de bits serão somados com valores errados e um resultado incorreto será gerado. No caso desta arquitetura, os tamanhos das entradas a serem somadas em ponto fixo são iguais. Ou seja, como as entradas a serem somadas possuem o mesmo número de bits total e de parte inteira, os números podem ser somados diretamente sem preocupação com alinhamentos do ponto separador.

Ao final da operação, é feito um arredondamento para garantir que não há mais bits do que o exigido pela representação adotada do número. O arredondamento feito nas operações em ponto fixo dessa arquitetura corresponde ao arredondamento para baixo (*round down)*, que será explicado nas seções a seguir.

Como exemplo, pode-se realizar a operação de soma exposta abaixo:

$$72.875 + 12.125 = 85$$

Na tabela 1 a seguir, observa-se a representação dos operandos no formato de ponto fixo. Os operandos foram nomeados como A, B e C para facilitar sua descrição.

Tabela 1 - Representação de alguns números em ponto fixo.

Operando	Valor	Parte Inteira	Parte Fracionária
A	72.875_{10}	001001000_2	$11100000000000000000000000_2$

B	12.125_{10}	000001100_2	$00100000000000000000000000_2$
C	85_{10}	001010101_2	$00000000000000000000000000_2$

Pode-se observar, na tabela 1, que o operando B já se encontra com a notação de complemento a dois e com o ponto separador alinhado, ou seja, já se encontra pronto para a soma.

Realizando-se a soma dos operandos A e B, obtém-se o seguinte resultado exposto abaixo:

$$001010101.00000000000000000000000000_2$$

Esse resultado já corresponde ao valor 85 em ponto fixo de base binária exposto na tabela 1.

2.3.5 Algoritmo de Multiplicação em Ponto Fixo

A operação de multiplicação de números em ponto fixo corresponde a uma simples operação de multiplicação em números binários, observando-se algumas exceções.

Essa operação é bem mais econômica do que a executada em números em ponto flutuante em *FPGAs*, por exemplo. Isso ocorre pois em ponto fixo os números não são representados com expoente. É justamente a presença desse expoente na notação de ponto flutuante que introduz uma maior conjunto de operações durante uma cálculo. Com isso operações aritméticas em ponto fixo são menos custosas computacionalmente do que me ponto flutuante.

O padrão de ponto flutuante necessita que módulos de hardware específicos sejam implementados. Para que esses módulos sejam gerados, recursos disponíveis de lógica serão consumidos, o que pode limitar o número de módulos aritméticos instanciados e reduzir sua frequência de operação (BARROS, 2014). Sabe-se que o desempenho em *FPGAs* está diretamente ligado ao grau de paralelismo e à frequência de trabalho dos módulos implementados. Como esses dois parâmetros dependem da quantidade de recursos disponíveis do *FPGA*, a escolha pelo padrão de ponto fixo pode significar um ganho de desempenho significativo (GUO et al., 2004).

Uma particularidade das multiplicações em ponto fixo, nesse projeto, é que elas são feitas com números constantes. Isso ocorre porque a equação utilizada no método *RTM* multiplica pesos, que correspondem a números inteiros, por alguns de seus termos a números que são somados em seguida (MEDEIROS, 2013).

A multiplicação por constantes torna possível a realização de otimizações para se economizar recursos do *FPGA*. Dessa maneira, a arquitetura desenvolvida nesse projeto não chega a realizar multiplicações de maneira esperada para números em ponto fixo. Essa operação corresponde a um conjunto de manipulações algébricas sobre o vetor de bits em notação de ponto fixo que, matematicamente, equivale a multiplicações por números constantes.

As constantes operadas dentro da arquitetura proposta em ponto fixo são: -1, 16 e -90. Algumas observações são feitas a seguir sobre essas multiplicações.

A multiplicação por -1 equivale, apenas, a uma operação de complemento a dois ao final do conjunto de soma de elementos da equação. Ou seja, não é necessário realizar, de fato, a multiplicação nesse caso.

A multiplicação por 16 também possui uma particularidade. Devido ao fato de o número ser uma potência de dois, a multiplicação se resume a uma operação de deslocamento para a esquerda (*Shift Left*). Sabe-se que cada deslocamento para a esquerda em notação binária equivale a uma multiplicação por 2. Com isso, para se multiplicar um número em ponto fixo por 16 basta realizar o deslocamento do vetor de bits quatro vezes para a esquerda. Essa operação pode ser ainda mais simplificada fazendo-se uma concatenação de quatro bits de valor zero ao final do vetor. Ou seja, assim como no caso da multiplicação por -1, a multiplicação por 16 em ponto fixo, na verdade, não corresponde a uma multiplicação de fato, e sim a uma simples manipulação de bits.

Diferentemente das duas anteriores, na multiplicação por -90, não há grandes otimizações. A multiplicação direta de uma variável de entrada por esse número é feita. Um detalhe importante ao se realizar uma multiplicação é a quantidade de bits necessários para o armazenamento da resposta sem perda de precisão. Como toda multiplicação de ‘n’ dígitos geram ‘2n’ dígitos, a cada multiplicação realizada o número de bits armazenado dobra, o que exige que um arredondamento seja feito.

2.3.6 Algoritmo de Arredondamento em Ponto Fixo

Para que as operações aritméticas encadeadas em ponto fixo não se tornem demasiadamente custosas em termos de recursos do *FPGA*, é necessário realizar arredondamentos. Esses arredondamentos limitam a quantidade de bits ocupados em *FPGA* pelos números ao final de operações aritméticas, como multiplicações e somas.

Como dito anteriormente, multiplicações entre dois números com a mesma quantidade de bits gera um número de resposta com o dobro de bits. Operações de soma entre números com ‘n’ dígitos geram números com ‘n+1’ dígitos de resposta. O que implicaria em uma constante expansão da quantidade de bits a serem utilizados para se armazenar um número a cada operação aritmética.

Por exemplo, com operações de multiplicação realizadas seguidamente, a quantidade de bits necessária para armazenar um número se tornaria proibitiva. Com isso, surge a necessidade dos arredondamentos.

O arredondamento utilizado nesta arquitetura para números em ponto fixo consiste em um arredondamento para zero, ou truncamento. Essa escolha foi feita devido à economia de recursos de hardware disponíveis em *FPGA*, visto que é a maneira mais simples de se realizar arredondamento.

2.3.7 Notação de Ponto Flutuante

A notação de ponto flutuante, apesar de ser semelhante à de ponto fixo, possui uma diferença significativa: a adição de um grupo de dígitos de expoente. A junção da parte inteira e da parte fracionária do número é denominada de significando.

Por definição os números nessa notação estão normalizados, ou seja, a parte inteira do significando nessa notação é representada por um dígito diferente de zero.

A fórmula que representa um número em ponto flutuante segue na equação 2.5:

$$\pm \sum_{i=0}^{n} di * \beta^{i-k+\,expoente} \qquad (2.3)$$

Nessa equação, *n* corresponde ao número total de dígitos utilizados para se representar um número e *k* corresponde à quantidade de dígitos reservados para a representação da parte fracionária do número.

A parte relativa ao campo de expoente pode assumir tanto valores negativos quanto positivos. É importante destacar que, no padrão de ponto flutuante IEEE-754, o expoente zero é um valor de referência denominado de *Bias*[10], onde os valores positivos de expoente se encontram acima do *Bias* e valores negativos, abaixo. Com isso, o valor real de um número deve levar em consideração a diferença entre os valores contidos no campo expoente e o valor do *Bias* (BARROS, 2014).

Devido ao acréscimo do campo de expoente, a representação de ponto flutuante tornou possível manipular a posição do "ponto separador" do significando e, dessa forma, agregou-se mais expressividade à notação anterior de ponto fixo.

2.3.8 Algoritmos de Soma e Subtração em Ponto Flutuante

Pode-se assumir que as operações aritméticas de soma e subtração possuem um mesmo algoritmo. Esse algoritmo possui os seguintes passos:

- **Preparação dos operandos:** Operação que verifica a diferença entre os expoentes e ajusta o significando de acordo com essa diferença. Nesse passo, também é feita a operação de complemento a dois.
- **Execução da Soma/Subtração:** Etapa onde ocorre a soma dos números binários.
- **Normalização:** Etapa de ajuste do resultado da soma para a representação definida pelo padrão com apenas um bit antes do ponto separador. O bit deve ser diferente de zero.
- **Arredondamento:** Etapa onde o número obtido é mapeado para o valor representável mais próximo.

A etapa de preparação dos operandos serve para alinhar os bits dos significandos entre si. Esse alinhamento garante que os operandos possuam os mesmos expoentes ao serem somados ou subtraídos. O alinhamento ocorre quando os "pontos separadores" dos dois números estão na mesma posição em seu respectivo vetores de bits. Isso é feito de maneira que a operação de soma ou subtração ocorra somente entre bits que estejam sendo associados a pesos equivalentes.

Como exemplo, pode-se realizar a operação de soma dos seguintes números: 9.807 e 3.1415, que corresponde a 12.9485, demonstrada abaixo.

[10] Para números em ponto flutuante de precisão simples o *bias* corresponde a 127.

$$9.807 + 3.1415 = 12.9485$$

A tabela 2 a seguir demonstra a representação dos operandos no formato IEEE-754. Os operandos foram nomeados como A, B e C para facilitar sua descrição.

Tabela 2 - Representação de alguns números em ponto flutuante

Operando	Notação Decimal	Notação binária padrão IEEE-754
A	9.80700_{10}	$01000000010010010000111111010100_2$
B	3.14159_{10}	$01000001000111001110100101111001_2$
C	12.94859_{10}	$01000001010011110010110101101110_2$

A tabela 3 a seguir demonstra os componentes dos números demonstrados, separados em sinal, expoente e mantissa.

Tabela 3 - Operandos em ponto flutuante separados em componentes para soma e subtração.

Operando	Sinal(*S*)	Expoente(*E*)	Mantissa(*M*)
A	0	10000010_2	1.00111001110100101111001*
B	0	10000000_2	1.10010010000111111010100*
C	0	10000010_2	1.10011110010110101101110*

*As mantissas encontram-se representadas com o bit implícito no algarismo mais significativo pois os números por padrão são representados como já normalizados.

Observa-se, na tabela 3, que os expoentes de A e B são diferentes. Portanto, para se realizar a operação de soma, é necessário que os operandos sejam manipulados de forma que estejam alinhados de acordo com a posição do ponto separador. Com isso, deve-se realizar um ajuste nas mantissas de acordo com a diferença entre os expoentes (E_A - E_B). A representação dessa equação pode ser observada a seguir:

$$E_A - E_B = 10000010_2 - 10000000_2 = 00000010_2 = 2_{10}$$

Observa-se, na equação acima, que o vetor de bits em base binária “00000010” equivale ao valor 2 em notação decimal.

Devido a essa diferença entre os expoentes, é necessário realizar dois deslocamentos para a direita do significando do número de menor expoente. No exemplo dado acima, esse número corresponde ao significando do operando ‘B’. Isso ocorre porque cada deslocamento para a direita corresponde a uma divisão pela base numérica adotada, no caso, a base binária. A cada deslocamento realizado, o expoente deve ser incrementado. Com os dois deslocamentos para a direita do significando, o seu expoente é incrementado duas vezes, tornando-se, igual ao expoente do primeiro operando, ‘A’. O resultado da respectiva operação está expresso abaixo:

$$S_B = 0.011001001000011111101012$$

Após a etapa de alinhamento dos significandos realiza-se a soma direta bit a bit dos mesmos. A representação dessa equação está exposta na equação 2.4:

$$S_A + S_B = 1.001110011101001011110012 + 0.011001001000011111101012 = 1.100111100101101011011002 \quad (2.4)$$

Ao final da soma bit a bit, o significando deve ser normalizado para que possua apenas um bit antes do ponto separador. Por convenção do padrão IEEE-754, todo número em ponto flutuante deve ser armazenado em sua notação normalizada. O padrão de normalização adotado exige que só haja um bit antes do ponto separador e que esse seja diferente de zero. A operação de ajuste do número para que se encaixe nessa situação descrita é denominada de normalização. No caso de exemplo citado, o número já está normalizado e não necessita passar por esta operação.

A etapa seguinte à normalização é a etapa de arredondamento. No caso do exemplo citado, o significando resultante possui vinte e cinco bits, dos quais vinte e três são da notação padrão para ponto flutuante e os demais resultantes da operação aritmética realizada.

Inicialmente, a expansão dos bits se faz necessária pois, como dito anteriormente, a soma de números com ‘n’ dígitos gera uma resposta com ‘n+1’ dígitos. Portanto, a resposta precisa de mais bits para não perder precisão. Porém, para se manter a notação exigida pelo padrão, a mantissa do resultado deve possuir apenas os vinte e três bits. Com isso, faz-se necessário um processo de arredondamento.

Apesar de o padrão IEEE-754 definir quatro formas de arredondamento, a mais usual é o arredondamento para o mais próximo ou par. Nesta dissertação, para garantir uma maior

precisão no resultado e uma conformidade absoluta com o que foi estabelecido pelo IEEE, esse foi o padrão utilizado.

Dessa maneira, o significando do resultado (S_R) assume a seguinte forma exposta abaixo:

$$S_R = 1.10011110010110101101110_2$$

Observa-se que, com essa operação, o bit menos significativo foi truncado. O número resultante dessa operação é o mais próximo do resultado visto na equação 2.4 para um vetor de bits de tamanho menor em um bit. Caso o número visto na equação 2.4 fosse arredondado para o seguinte vetor de bits "1.10011110010110101101111", o erro introduzido com a aproximação desse número seria maior do que o visto no significando S_R exposto acima. Com isso, o número foi "arredondado para baixo", foi truncado.

O significando resultante, já sem o bit expandido gerado pelo processo de soma, encontra-se exposto a seguir. Esse valor equivale ao significando do número "12.94859", exposto na tabela 2:

$$S_R = 1.10011110010110101101110_2$$

A última etapa envolve apenas a retirada do bit implícito para o retorno a notação de vinte e três bits de mantissa.

2.3.9 Algoritmo de Multiplicação em Ponto Flutuante

A operação de multiplicação, diferentemente da operação de soma e subtração, não exige que os operandos passem por ajustes antes de serem operados. Isso ocorre porque não há a necessidade de alinhamento dos pontos separadores dos números a serem operados. Com isso, os significandos dos números já podem ser operados diretamente.

Basicamente, o algoritmo de multiplicação segue o seguinte padrão:

- **Soma dos expoentes:** Nessa etapa, ocorre a soma dos expoentes de entrada entre si para gerar o expoente resultante.
- **Multiplicação dos significandos:** Os significandos devem passar por um processo de multiplicação normal entre si. O número de bits resultante dessa operação corresponde à soma dos números de bits de cada um dos significandos.
- **Normalização:** Etapa de ajuste do resultado da multiplicação para a representação padrão.
- **Arredondamento:** Etapa onde o número obtido é ajustado para se retornar à notação com o mesmo número de bits iniciais.

- **Definição do sinal de resposta:** O sinal resultante da operação é feito realizando-se uma operação de XOR, ou "OU EXCLUSIVO" com os sinais dos operandos.

Como exemplo, pode-se realizar a multiplicação dos respectivos números: -2.4 e 9.8, que corresponde a -23.52, demonstrada abaixo:

$$-2.4 * 9.8 = -23.52$$

A tabela 4 a seguir demonstra a representação dos operandos no formato IEEE-754. Os operandos foram nomeados como A, B e C para facilitar sua descrição.

A tabela 5 a seguir apresenta os operandos separados em sinal, expoente e mantissa.

Tabela 4 - Representação de alguns números em ponto flutuante

Operando	Notação Decimal	Notação binária padrão IEEE-754
A	-2.4_{10}	$11000000000110011001100110011010_2$
B	9.8_{10}	$01000001000111001100110011001101_2$
C	-23.52_{10}	$11000001101111000010100011110111_2$

Tabela 5 - Operandos em ponto flutuante separados em componentes para multiplicação

Operando	Sinal (*S*)	Expoente (*E*)	Mantissa (*M*)
A	1	10000000_2	$1.00110011001100110011010_2$*
B	0	10000010_2	$1.00111001100110011001101_2$*
C	1	10000011_2	$1.01111000010100011110111_2$*

*As mantissas encontram-se representadas com o bit implícito no algarismo mais significativo, pois os números por padrão são representados como já normalizados.

Como dito anteriormente, a primeira etapa corresponde à soma dos expoentes. É importante destacar que, após a etapa de soma dos expoentes, é necessário se realizar uma subtração nesse resultado. Essa subtração corresponde ao valor do *bias* para números em ponto flutuante em precisão simples. O valor do *bias* corresponde a 127 para números em ponto flutuante em precisão simples. Isso ocorre porque ambos os números a serem somados já possuem implicitamente esse *bias* já somados a seus expoentes. Portanto, quando somados, teriam seus *bias* também somados, gerando um número incorreto. Para sanar essa irregularidade, a subtração de um dos *bias* é necessária. A operação de soma do expoente do primeiro operando (E_A) e a do segundo operando (E_B) pode ser observada a seguir:

$$E_A + E_B = 10000000_2 + 10000010_2 - 01111111_2 = 10000011_2$$

Em seguida, os significandos devem ser multiplicados. Ao multiplicá-los, o número resultante terá o número de bits correspondente à soma do número de bits dos dois significandos. Esse número expandido deve ser preservado para não se perder precisão até que a devida etapa de arredondamento seja executada. Pode-se observar o resultado da multiplicação dos dois significandos (S_A e S_B) a seguir:

$$S_A * S_B = 1.00110011001100110011010_2 * 1.00111001100110011001101_2 = 1.01111000010100011110110011111010111000010100 10_2$$

Observa-se que o significando do resultado (S_R) tem a seguinte forma:

$$S_R = 1.01111000010100011110110 \quad 01111101011100001010010_2$$

O significando resultante exposto acima possui os vinte e três últimos bits separados dos vinte e quatro bits mais significativos. A opção por essa representação é meramente didática e visa simplificar o processo de entendimento do arredondamento.

Os últimos bits do significando resultante são necessários apenas para critérios de arredondamento. É feita uma verificação para se saber qual o número representável mais próximo do significando resultante com a quantidade de bits exigida pelo padrão. Com isso, o arredondamento procura minimizar o erro introduzido por esse processo. Isso implica que o bit menos significativo dos vinte e quatro bits mais significativos deve ter seu valor somado com um. Eventuais estouros resultantes dessa soma devem ser propagados ao longo dos demais bits.

O significando resultante, já com a soma gerada pelo processo de arredondamento, se encontra exposto a seguir. Esse valor equivale ao significando de -23.52 em base decimal, exposto na tabela 4.

$$S_R = 1.01111000010100011110111_2$$

A última etapa envolve apenas a retirada do bit implícito para o retorno a notação de vinte e três bits de mantissa.

2.3.10 Algoritmo de Normalização

Os números em ponto flutuante representados no padrão IEEE-754 encontram-se normalizados. Eles possuem um bit implícito, diferente de zero, na parte mais significativa de sua mantissa.

Para que um número binário em ponto flutuante esteja no padrão, é necessário que ocorra uma etapa de normalização ao final de suas operações aritméticas. Essa etapa visa ajustar o significando para que possua um único bit, que seja diferente de zero, antes do ponto separador. Isso faz com que seja necessário fazer uma verificação da quantidade de bits antes do ponto separador ao fim das operações.

Ao final de alguma operação aritmética, caso haja mais de um bit a esquerda do ponto separador, é necessário se realizar uma normalização. Se esses bits forem diferentes de zero, efetuam-se deslocamentos para direita do significando até que haja apenas um bit antes do ponto separador. Como um deslocamento à direita, nesse caso, equivale a uma divisão por dois, deve-se somar ao expoente o número de deslocamentos realizados para não ocorrer uma perda de representação.

Caso o bit antes do ponto seja igual a zero, também é necessário realizar uma operação de normalização. Nesse caso, deve-se deslocar o significando para a esquerda até que o bit antes do ponto se torne diferente de zero. Como um deslocamento para a esquerda corresponde a uma multiplicação pela base, deve-se subtrair do expoente o número de deslocamentos realizados. Isso ocorre para que não haja perda de representação numérica.

Basicamente, pode-se dizer que o algoritmo de normalização segue os seguintes passos:

1. Identificação do posicionamento do primeiro bit igual a um do significando.
2. Cálculo da distância desse bit até a posição do bit implícito. Caso esteja à esquerda, a distância é positiva; caso esteja à direita, a distância é negativa.
3. Realização do deslocamento dos bits do significando até que o bit implícito se torne diferente de zero.
4. Ajuste do expoente de acordo com o número de deslocamentos realizados no significando. Soma-se a distância obtida no segundo passo ao expoente.
5. O número obtido é verificado quanto à necessidade de arredondamento.

2.3.11 Algoritmo de Arredondamento

De acordo com BARROS (2008, p. 27):

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

Consequentemente, ao realizar operações aritméticas com números em ponto flutuante, haverá sempre a introdução de um erro inerente ao processo de arredondamento. Esse erro de arredondamento é algo característico da aritmética computacional e já gerou diversos problemas (COE, 1996).

O padrão IEEE-754 estabelece alguns algoritmos de arredondamento para seus números. Nesse caso, entende-se, por arredondamento, o processo de redução do número de bits de um número para a quantidade de bits definida pelo padrão. Naturalmente, apesar de provocar a perda de representação numérica, essa etapa se faz necessária, caso contrário, poder-se-ia ter números de representação infinita de bits, o que é absurdo, dadas as limitações dos recursos computacionais.

As formas de arredondamento são:

- **Arredondamento em direção ao infinito positivo:** Nesse caso, os números positivos devem ser arredondados para o mais próximo em direção ao infinito positivo. Os números negativos são arredondados em direção do zero.
- **Arredondamento em direção ao infinito negativo:** Nesse caso, os números negativos devem ser arredondados para o mais próximo em direção ao infinito negativo. Os números positivos são arredondados em direção do zero.
- **Arredondamento em direção ao zero:** Também denominada de truncamento, essa forma consiste apenas em arredondar o número para o mais próximo representável em direção ao zero.
- **Arredondamento em direção ao mais próximo ou par:** Esse modo arredonda o número para o mais próximo representável do valor original. Para os casos onde o nível de proximidade é o mesmo entre dois números, arredonda-se para o próximo par por convenção.

Dentre as formas de arredondamento citadas acima, a recomendação dada pelo IEEE é a última: arredondamento em direção ao mais próximo ou par. Isso ocorre porque esta forma de arredondamento apresenta a menor taxa de erro entre as técnicas citadas. Com isso, esse se tornou o modo de arredondamento mais utilizado nas implementações que usam o padrão

IEEE-754. É importante destacar que, apesar de essa ser a forma mais eficiente em termos de precisão, é a mais custosa em termos de recursos computacionais.

2.3.12 Representação de valores especiais

Os valores ditos especiais correspondem a uma série de números que, caso não sejam tratados de maneira distinta, podem gerar incongruências. Alguns, inclusive, não possuem representação numérica, apenas simbólica e conceitual. Alguns desses casos também possuem apenas uma formalização de um número que não se encaixa na representação binária utilizada.

A seguir, estão expostos os principais casos especiais:

- **Infinito:** Notação que demonstra que um determinado número é, em módulo, maior que o maior número representável pelo padrão. Surge como resultado de algumas operações na aritmética padrão, como a divisão de um número diferente de zero pelo próprio zero. Possui uma representação positiva e negativa.
- **Zero:** É representado como um conjunto de bits zero, desconsiderando-se o bit de sinal. O zero também possui sinal positivo e negativo, afim de preservar o sentido de aproximação da função.
- **NaN:** Notação que, traduzida do inglês "*Not a Number*", literalmente corresponde a um "não número". Essa representação surge em casos onde a aritmética padrão não consegue representação nos números reais. Dentre elas, pode-se citar a divisão de zero por zero, a raiz de menos um e a soma de infinito positivo com infinito negativo. Essa representação, diferentemente das anteriores, não necessita de notação de sinal, podendo assumir qualquer valor sem gerar impacto para as operações. Qualquer número operado com um *NaN* deve gerar outro *NaN*, fazendo com que o resultado da operação também seja invalidado.
    - **Denormal:** Corresponde a uma faixa de valores de números ditos muito pequenos. Como correspondem a números que estão além da precisão necessária para sua representação, não é possível representá-los devidamente em modo normalizado. Comumente, são tratados como zero para economia de recursos.

A tabela 6 a seguir expõe a representação numérica binária dos casos especiais presentes no padrão.

Tabela 6 - Representação de valores especiais no padrão IEEE-754

Tipo	Sinal	Expoente	Fração do significando
+ Infinito	0	11111111	00000000000000000000000
- Infinito	1	11111111	00000000000000000000000
Zero	+/-	00000000	00000000000000000000000
NaN	+/-	11111111	xxxxxxxxxxxxxxxxxxxxxxx
Denormal	+/-	00000000	xxxxxxxxxxxxxxxxxxxxxxx

Na tabela 6, os algarismos representados com “x” denotam uma palavra binária na qual pelo menos um dos seus bits é diferente de zero. Os casos de “zero”, “NaN” e “Denormal” podem vir sinalizados tanto com o sinal positivo quanto de negativo.

É importante destacar que, no caso das frações dos significandos dos NaN´s e dos números ditos denormais, os bits podem assumir qualquer valor após o ponto sem que isso afete sua classificação como valor especial. Porém, ao menos um desses bits tem que ser diferente de zero, caso contrário, o número cairia na representação de outro caso especial, como o de infinito, por exemplo.

2.4 Algoritmo de Análises Sísmicas do Solo - RTM

Para se realizar a exploração de camadas geológicas, faz-se necessário o uso de algum método de análise do solo, ou método sísmico. Esses métodos permitem a identificação das camadas do solo em subsuperfície.

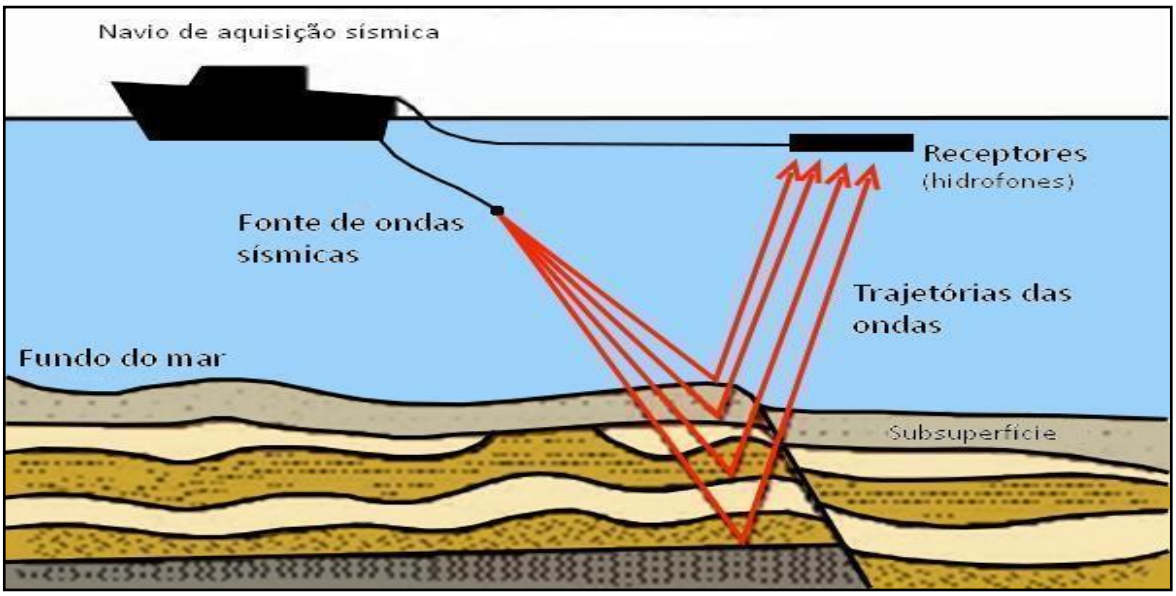
Dentre as aplicações dos métodos sísmicos, pode-se destacar a sismologia de exploração. Essa área é responsável pela exploração e produção de hidrocarbonetos em profundidades de até 10km. Essa área lida com os métodos de exploração e produção de petróleo e gás (ROCHA, 2010).

A sísmica de reflexão é a técnica de exploração geológica mais comum na indústria petrolífera. Nela, inicialmente, é escolhido um local onde será posicionada uma malha de receptores (hidrofones ou geofones). Em seguida, é posicionada e detonada uma carga capaz de gerar ondas sísmicas. Estas ondas percorrem o meio e tem parte de sua energia propagada sob forma de reflexão. Essas reflexões ocorrem com mais intensidade nas interfaces entre as camadas geológicas. Isso ocorre porque diferentes camadas de solo possuem diferentes características de propagação e densidade. Essas características provocam modificações nos

tempos de propagação e amplitude das ondas refletidas que, por usa vez, são captadas pelos receptores posicionados na superfície (ROCHA, 2010).

A figura 8 a seguir expõe, simplificadamente, como funciona o método de reflexão sísmica para uma carga no mar.

Figura 8 - Método de reflexão sísmica



O algoritmo de análises sísmicas implementado nessa dissertação foi o *RTM*. Para resolver o método de reflexão sísmica através desse algoritmo, é necessário analisar a equação de onda, visto que é através delas que é feita a análise geológica nesse caso. Para isso, pode-se fazer uso de estratégias matemáticas que solucionam essa equação.

A análise sísmica utiliza um método matemático que resolve a equação de onda assumindo que os campos de pressão podem se propagar a partir de um pulso sísmico inicial. Inicialmente, se considera que eles se propagam da fonte de ondas sísmicas para receptores como hidrofones ou geofones, etapa denominada modelagem. Em seguida, considera-se que eles se propagam dos receptores para a fonte de ondas sísmicas, etapa denominada migração (ROCHA, 2010).

A equação de onda acústica mais simples no espaço 3D é mostrada abaixo:

$$\frac{\partial^2 P(x,y,z,t)}{\partial x^2} + \frac{\partial^2 P(x,y,z,t)}{\partial y^2} + \frac{\partial^2 P(x,y,z,t)}{\partial z^2} - \frac{1}{v^2(x,y,z)} \frac{\partial^2 P(x,y,z,t)}{\partial t^2} = 0, \qquad (2.5)$$

Onde: $\frac{\partial^2 P(x,y,z,t)}{\partial x^2}, \frac{\partial^2 P(x,y,z,t)}{\partial y^2}, \frac{\partial^2 P(x,y,z,t)}{\partial z^2}$ e $\frac{\partial^2 P(x,y,z,t)}{\partial t^2}$ são, respectivamente, as derivadas de segunda ordem em relação ao eixo x, eixo y, eixo z e ao tempo t e *v*(x,y,z) representa a velocidade de propagação da onda no ponto de coordenadas x, y e z, que correspondem às coordenadas espaciais horizontal, vertical e de profundidade.

Devido ao fato da equação 2.5 ser uma equação diferencial sem uma resolução algébrica para sistemas com grandes variações de velocidade, o algoritmo *RTM* faz uso de uma resolução construída através de um operador de diferenças finitas[11] (BARROS, 2014).

A modelagem sísmica via *RTM* é feita através da resolução da equação de diferenças finitas para cada ponto do modelo ao longo de determinados espaços de tempo. Isso resulta em um modelo sísmico em três dimensões para o método de migração reversa no tempo.

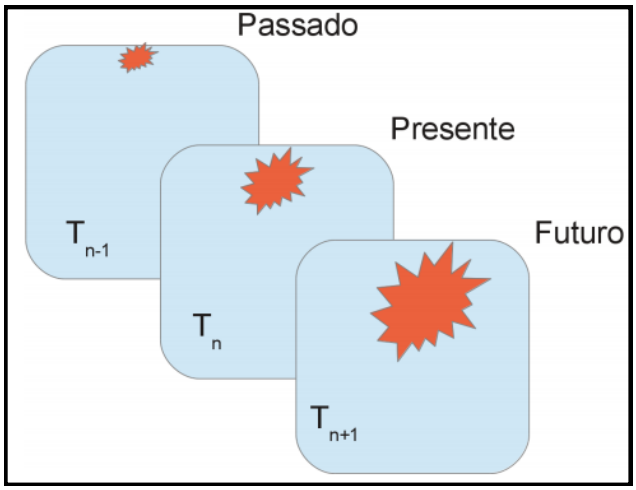
Para se resolver a equação de diferenças finitas, são utilizadas três matrizes de pressão acústica exercida por um pulso sísmico propagado por um sismograma[12]. Uma delas para modelar a pressão acústica no tempo atual, outra para modelar a pressão acústica no tempo anterior, e a terceira para modelar a pressão acústica no tempo futuro.

Em cada etapa da simulação, essas três matrizes de campos de pressão se sobrepõem umas as outras. Com isso, a matriz do tempo futuro se torna a matriz do tempo presente e a matriz do tempo presente se torna a matriz do tempo anterior. Pode-se observar o esboço dessa simulação na figura 9 a seguir.

Em seguida, substitui-se o pulso sísmico pela propagação inversa da onda, ou seja, a propagação ocorre em sentido contrário ao pulso inicialmente gerado. Ao final, os dados obtidos com essas simulações geram uma matriz que é convertida em uma imagem. Nessa imagem, espera-se que sejam destacadas as interfaces das camadas em subsuperfície (SANTOS, 2012).

[11] O método de diferenças finitas consiste na substituição das derivadas presentes na equação em estudo, por aproximações algébricas de diferenças obtidas a partir da expansão em série de Taylor. (SANTOS, 2012).
[12] Um sismograma é o registro da atividade sísmica ocorrida em uma determinada região. (BARROS, 2014).

Figura 9 - Simulação da propagação de um pulso sísmico por um campo de pressão.



Fonte: BARROS, 2014, P. 40

A equação de diferenças finitas para esse modelo é apresentada abaixo:

$$
\begin{aligned}
Cijk = \; & 2.0 * B_{ijk} * \left(Abs_i * Abs_j * Abs_k\right) + \left(Abs_i * Abs_j * Abs_k\right)^2 \\
& * \Big\{VEL * VEL * FAT \\
& * \Big[-1 * \left(B_{ij-2K} + B_{ij+2K} + B_{i-2jK} + B_{i+2jk} + B_{ijk-2} + B_{ijK+2}\right) + 16 \\
& * \left(B_{ij-1K} + B_{ij+1K} + B_{i-1jK} + B_{i+1jk} + B_{ijk-1} + B_{ijK+1}\right) - 90 * \left(B_{ijk}\right)\Big] \\
& - A_{ijk}\Big\} \qquad (2.6)
\end{aligned}
$$

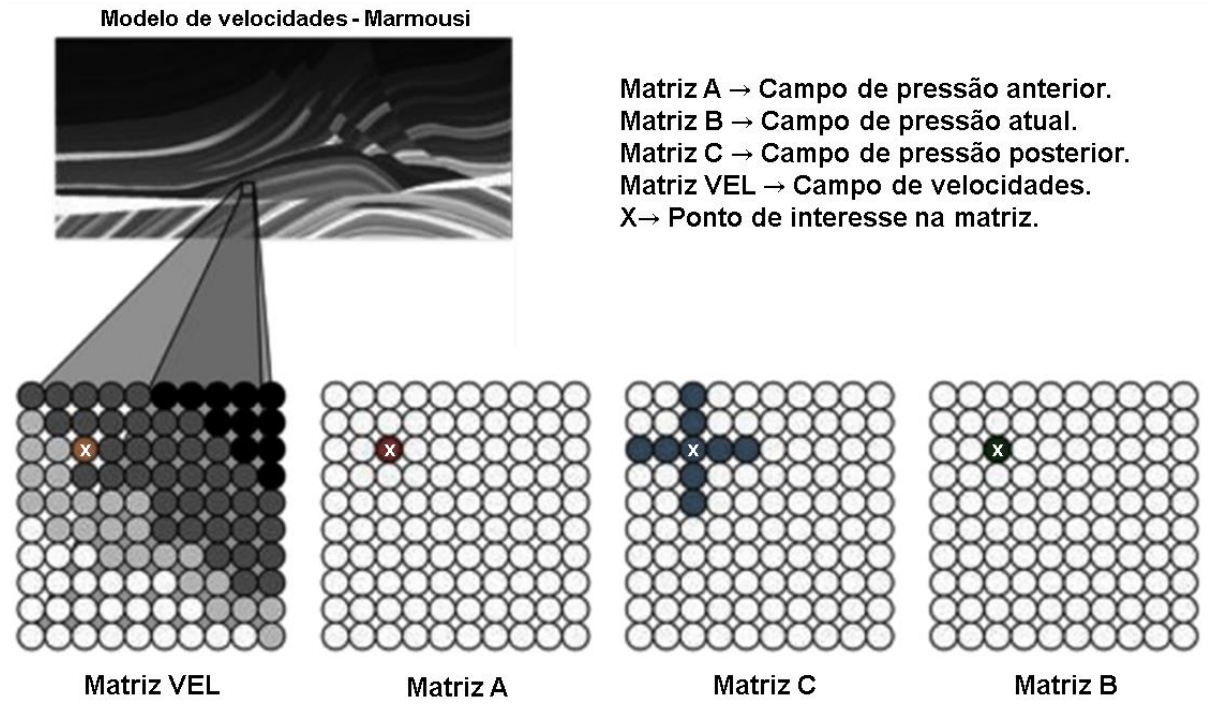
Onde A_{ijk}, B_{ijk} e C_{ijk} representam o elemento localizado na posição i, j e k das matrizes. Essas variáveis representam o campo de pressão nos tempos n-1, n e n+1 e modelam a propagação da onda acústica tanto no espaço quanto no tempo.

O termo VEL representa a velocidade da propagação do pulso sísmico da onda através do meio. Essa é uma característica das camadas geológicas que estão sendo modeladas e depende da composição do terreno.

O termo fat corresponde a uma constante de calibração da equação que é calculada levando-se em conta alguns parâmetros escolhidos na execução do *RTM*. Dentre eles, pode-se destacar o espaçamento entre os pontos da malha e a frequência de corte da onda sísmica que será propagada no modelo.

A Figura 10 a seguir ilustra a associação das matrizes utilizadas para suprir a equação de propagação de onda discretizada utilizadas pelo *RTM*. Nessa figura, observa-se o modelo de velocidades utilizado, o Marmousi (1988).

Figura 10 - Equação de propagação da onda e suas respectivas matrizes de processamento



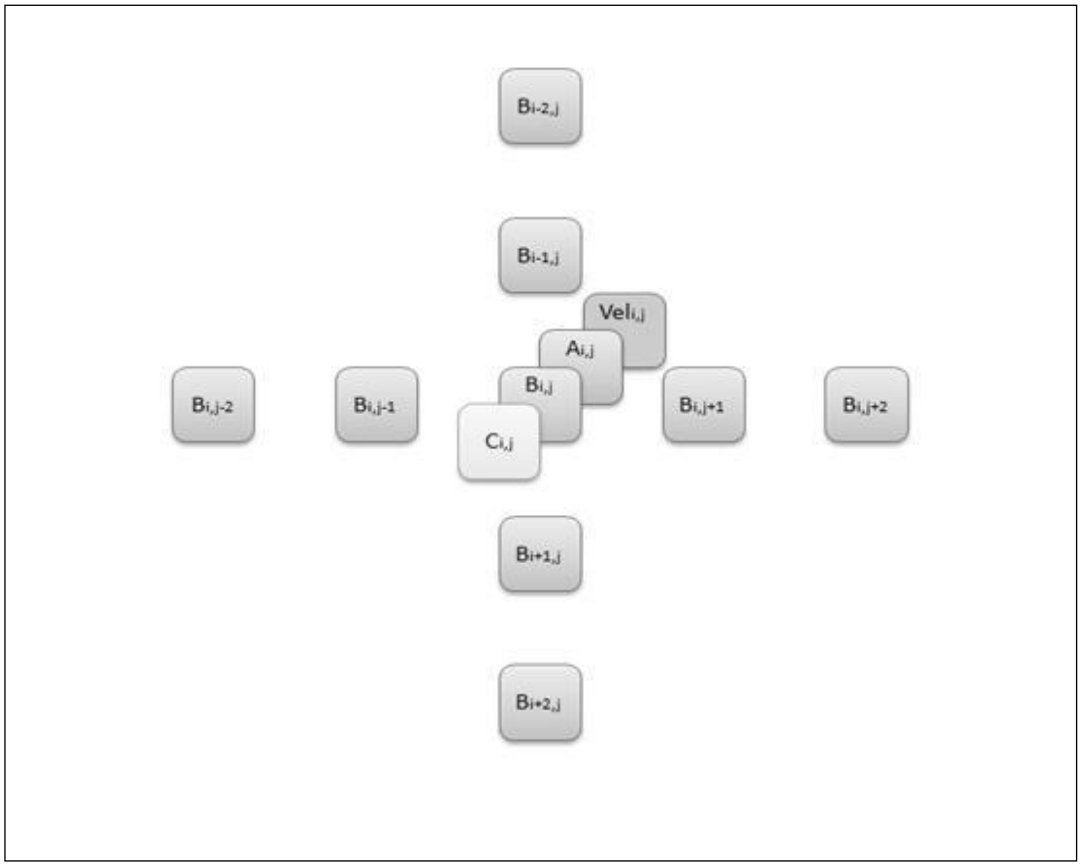
Fonte: MEDEIROS, 2013, P. 14

Para calcular o ponto $Cijk$ da matriz do campo de onda no tempo n+1, são necessários pontos das matrizes A_{ijk}, B_{ijk}, como visto na equação 2.6.

Na figura 11 a seguir, pode-se observar a estrutura e a disposição dos dados necessários para o cálculo do ponto de interesse. É importante salientar que os elementos da matriz B, ilustrados na figura 11, são vistos sob uma perspectiva de duas dimensões apenas, isso foi feito para facilitar a observação dos demais elementos.

Como exposto na equação 2.6, a matriz B possui três dimensões. Foi com base nessa representação da equação que foi elaborada a arquitetura em hardware para a solução do método sísmico.

Figura 11 - Dados de entrada da equação de diferenças finitas no RTM



2.5 Conclusões

Neste capítulo, foi feita a fundamentação de alguns dos temas abordados nesse trabalho. Essa etapa adquire grande importância quando se leva em consideração o caráter abrangente em temas da área computacional presentes nessa dissertação.

Inicialmente, foi feito um debate sobre computação de alto desempenho, fazendo-se uma breve comparação entre *ASIC´s*, *CPU´s* e *FPGA´s*. Em seguida, foi feita uma melhor explanação sobre a constituição dos *FPGA´s* e seu funcionamento.

Após essas descrições, realizou-se uma introdução à aritmética em ponto flutuante e sua presença na computação atual. Em seguida, foi debatido, de maneira introdutória, o padrão IEEE-754 para números binários em ponto flutuante. Durante essa explicação, foram abordados desde a notação utilizada até a maneira recomendada de ser realizar algumas operações aritméticas e tratar casos especiais. Essa seção mostrou-se uma das mais

importantes, visto que o foco desse trabalho está no desenvolvimento de um módulo aritmético capaz de obedecer aos critérios determinados por esse padrão.

Por fim, foi abordado o algoritmo de análises sísmicas do solo utilizado pela indústria petrolífera para descobrir poços de hidrocarbonetos. É justamente uma parte deste algoritmo, no caso o núcleo aritmético, que é o foco desta dissertação e foi prototipada para a *FPGA*.

Capítulo

3

3. Trabalhos relacionados

Esse capítulo tem, por objetivo, discutir e apresentar trabalhos relacionados ao tema desta dissertação. Dentre os tópicos a serem analisados, o foco se dá na implementação de *cores* aritméticos em *FPGA*.

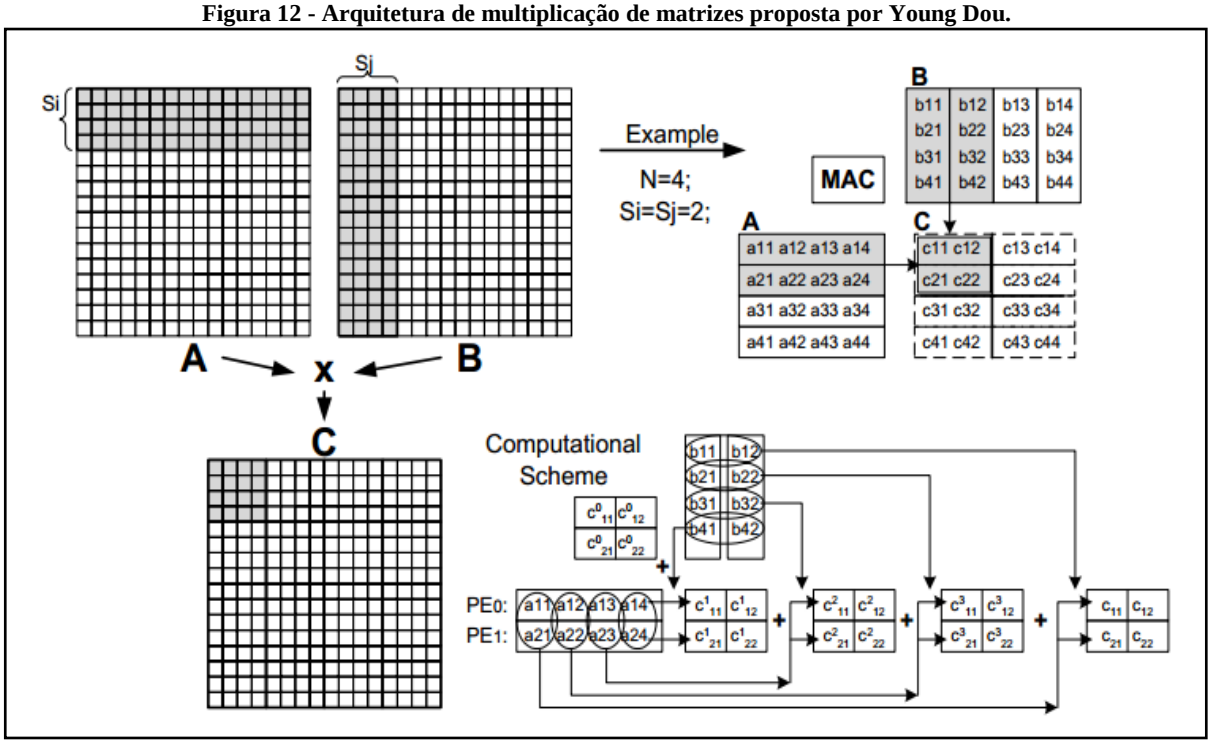
3.1 64 bit Floating Point FPGA Matrix Multiplication - Dou, Yong; Vassiliadis, S.; Kuzmanov (2005).

Neste trabalho é descrito em detalhes a implementação de um multiplicador e acumulador em ponto flutuante de precisão dupla. O foco dado por Dou (2005) está na arquitetura desenvolvida para a realização da multiplicação de matrizes densas. Essa arquitetura implementa o reuso e explora a localidade de dados, reduzindo, assim, o *throughput* de acesso à memória. Para tanto foi implementado um *pipeline*[13] de 12 estágios, bem como, uma arquitetura de controle e acesso a dados.

Nesse trabalho é proposta uma arquitetura mista do tipo *Single Program Multiple Data (SPMD)*. Para isso, um processador de propósito geral provê os dados e controle a processadores de propósito específico, no caso, os multiplicadores e acumuladores.

Pode-se observar uma visão geral da arquitetura proposta por Yong Dou na figura 12, a seguir. Nela existem 39 processadores de propósito específico, responsáveis por realizar o produto interno de uma linha, de uma matriz, pela coluna da outra matriz de entrada.

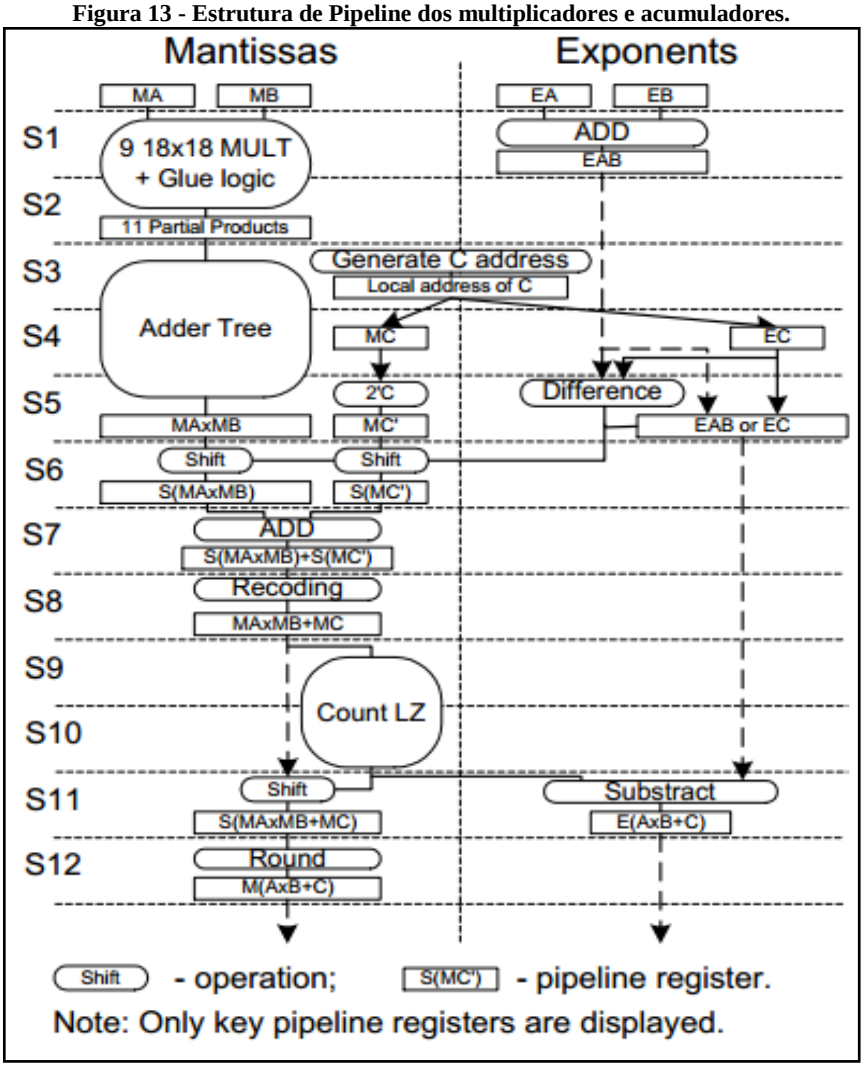
Figura 12 - Arquitetura de multiplicação de matrizes proposta por Young Dou.



[13] Técnica de hardware que dispões as instruções a serem executadas em uma fila de memória dentro do processador. Assim que uma instrução termina um estágio, segue para o estágio posterior e a próxima instrução já ocupa seu estágio.

Na figura 13 a seguir está exposta a estrutura de *pipeline* do multiplicador acumulador desenvolvido. Observa-se que as unidades de multiplicação possuem 9 multiplicadores de 18 por 18 bits, o que implica em uma lógica de particionamento do significando de acordo com os multiplicadores disponíveis na placa utilizada. Foi utilizada uma árvore de somas para as operações dos produtos parciais gerados pela unidade de multiplicação. Nota-se a presença de uma unidade de soma para lidar com o resultado das árvores de soma e com um número advindo da memória. As demais unidades incluem uma unidade de *leading zeros*, para contagem de zeros da normalização e, posteriormente, seu arredondamento.

Figura 13 - Estrutura de Pipeline dos multiplicadores e acumuladores.



Dentre os resultados obtidos por Dou, têm-se um desempenho de 15,6 *Gflops*, uma frequência de 177Mhz e a ocupação de 1419 *slices* em uma *Xilinx Virtex II* Pro XCV2P125.

3.1.1 Conclusões

O trabalho de (DOU, 2005) expõe um núcleo aritmético para *FPGA* no padrão IEEE-754. Apesar da aplicação e da precisão adotadas por Dou (2005), serem diferentes da aplicação proposta para esse trabalho, ambos fazem uso de operações aritméticas padrão IEEE-754 para *FPGA´s*.

O particionamento das operações aritméticas ao longo de uma estrutura de *pipeline* feito por DOU (2005), permitiu o aumento da frequência de operação de seu núcleo aritmético. A estratégia de quebra de operações em etapas menores e o armazenamento em registradores de *pipeline* foi a mesma utilizada no núcleo aritmético desta dissertação, apesar das diferenças de implementação.

O trabalho de Dou, assim como está dissertação, também faz uso de uma unidade de contagem de zeros (*leading zeros*) durante a etapa de normalização. Porém, (DOU, 2005) opta por particionar o processo de contagem em quatro estágios de *pipeline* para ganho de frequência de operação. Enquanto o núcleo aritmético apresentado nesta dissertação faz uso de uma etapa de lógica combinacional (multiplexador[14]) e uma de lógica sequencial (case) para a contagem dos zeros. O algoritmo de contagem (DOU, 2005) se mostra mais eficiente em *FPGAs* com menos recursos disponíveis como era seu caso com uma *Xilinx Virtex II* Pro XCV2P125. No entanto, para *FPGAs* mais modernas e com bem mais lógica disponível, o gasto com a lógica combinacional e mais vantajoso devido ao ganho na frequência de operação.

3.2 Implementação em FPGA de um modulo multiplicador aritmético de alto desempenho para números de ponto flutuante de dupla precisão, padrão IEEE-754 – Barros A. C., Barbosa J. e Lima M. E (2008)

Este trabalho aborda o desenvolvimento de um módulo que atua como coprocessador de aplicação específica para multiplicação de matrizes densas. O trabalho apresenta uma unidade aritmética do tipo multiplicador e acumulador (*MAC*). Essa unidade foi implementada em *FPGA* e faz uso do padrão IEEE 754 para números em ponto flutuante de dupla precisão. A arquitetura proposta foca na completa aplicação desse padrão IEEE. Para

[14] Dispositivo que seleciona dados de duas ou mais fontes de entrada em um único canal de saída.

tanto, estão presentes as etapas de operações aritméticas, de normalização, arredondamento, bem como, os tratamentos das exceções previstos pelo citado padrão IEEE.

Como se pode observar na figura 14 a seguir, o *MAC* foi dividido em 11 estágios de *pipeline*. Ao longo de seu *pipeline*, destacam-se algumas estruturas, como uma unidade de soma para a acumulação dos produtos parciais da multiplicação e unidades de comparação de expoente, necessárias para realização dos algoritmos de soma e multiplicação. Além disso, estão presentes unidades para deslocamento ou *shifts* e complemento a dois, para modificação dos números de entrada que estão na notação de sinal, expoente e mantissa. Observa-se ao longo de todo o *pipeline*, uma unidade para tratamento de exceções. Esta unidade trata as exceções como a representação para números infinitos e *NaN´s*. Isso é feito através da análise dos números na entrada do *MAC*, bem como, da verificação dos sinais gerados pelas demais estruturas presentes na arquitetura do *pipeline*.

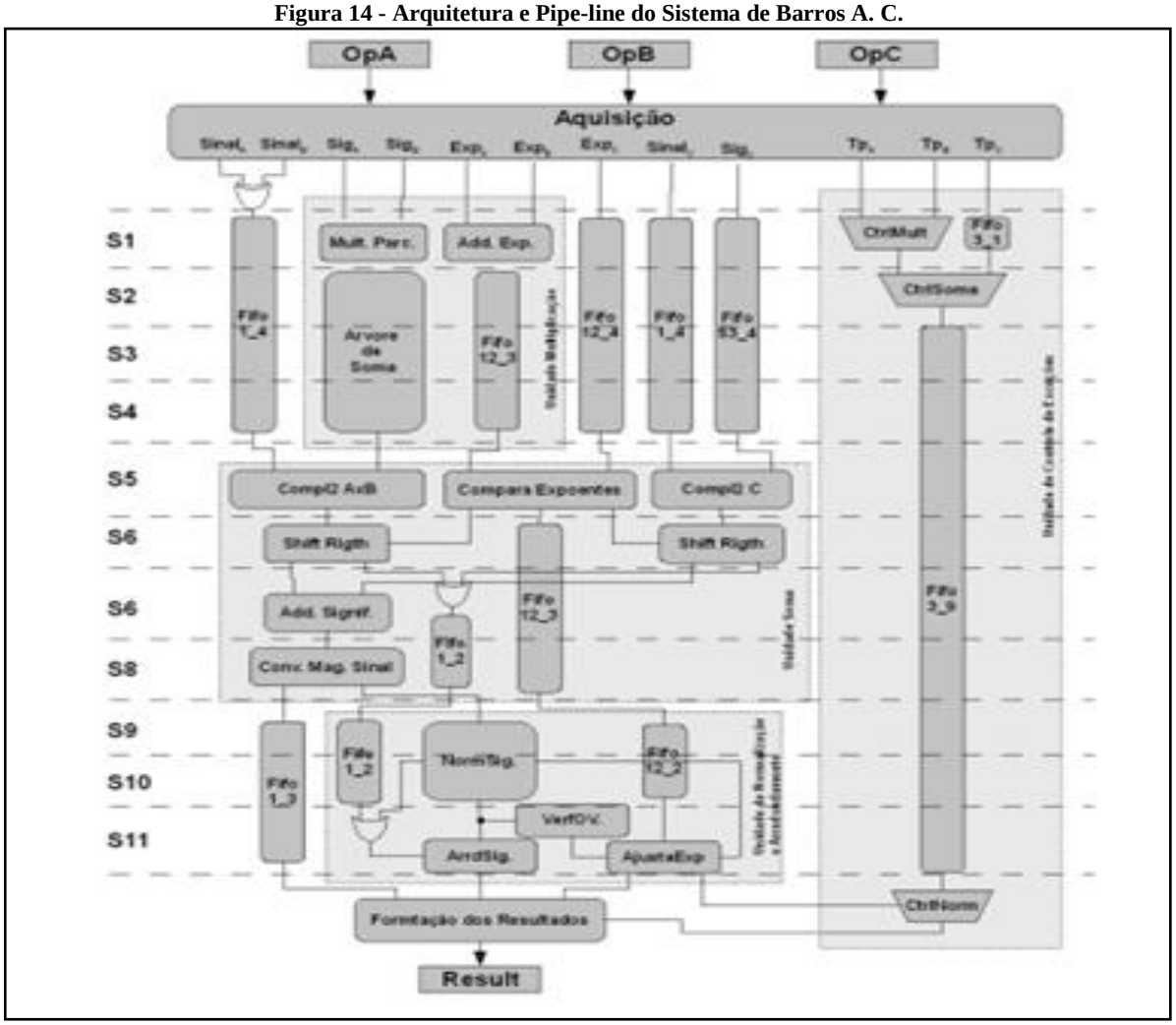
Esse projeto foi realizado para se aproveitar o paralelismo entre as operações aritméticas. Com isso, operações como a de soma parciais da multiplicação ocorrem simultaneamente a soma de expoentes. Isso permite um maior número de operações por ciclo de relógio.

O algoritmo de normalização utilizado foi *leading zeros*, que conta o número de zeros antes do primeiro bit ‘1’ encontrado. Esse número de zeros é utilizado para se realizar o deslocamento do vetor a ser normalizado. Caso o deslocamento seja realizado para a esquerda é feita uma subtração do número de zeros encontrados no expoente. Caso o deslocamento seja realizado para a direita é feita uma soma do número de zeros encontrados no expoente.

O algoritmo de arredondamento utilizado foi o arredondamento para o mais próximo ou par (*round to nearest)*. Esse arredondamento apesar de mais custoso apresenta um menor erro em relação ao número original. Com isso, a precisão utilizada nessa operação foi a máxima prevista pelo padrão IEEE-754.

O projeto do *MAC* foi sintetizado para um dispositivo da *Xilinx*, denominado *Virtex II* XC2V6000. Dentre os resultados obtidos, pode-se citar uma frequência de operação de 69.44 Mhz. O *MAC*, juntamente com uma interface de barramento *PCI*, ocuparam cerca de 14% da lógica disponível do dispositivo. Com isso, o autor espera que o dispositivo suporte até 7 unidades de ponto flutuante de 64 bits semelhantes na *FPGA* utilizada.

Figura 14 - Arquitetura e Pipe-line do Sistema de Barros A. C.



3.2.1 Conclusões

O trabalho proposto por Barros (2008) mostrou a implementação de um multiplicador e acumulador (MAC) padrão IEEE-754 para *FPGAs*. Ao contrário desta dissertação que tem uma aplicação para o algoritmo de *RTM*, o MAC foi desenvolvido para trabalhar na multiplicação de matrizes. No entanto, por também ser um núcleo aritmético padrão IEEE-754 implementado em *FPGA*, compartilha uma mesma temática.

A divisão das operações aritméticas de maneira a se explorar o paralelismo entre as intruções permitiu a Barros o melhor aproveitamento dos recursos disponíveis no *FPGA*. Essa estratégia foi a mesma utilizada no núcleo aritmético desta dissertação, o que permitiu uma melhoria na execução do algoritmo.

Em Barros (2008), assim como nesta dissertação, se faz uso do algoritmo de arredondamento recomendado pelo IEEE, para o mais próximo ou par. Para tanto, fez-se uso de uma operação de lógica de "OU" para determinação do Stick bit[15], que foi a mesma estratégia adota nesta dissertação para a implementação do arredondamento em *FPGA*.

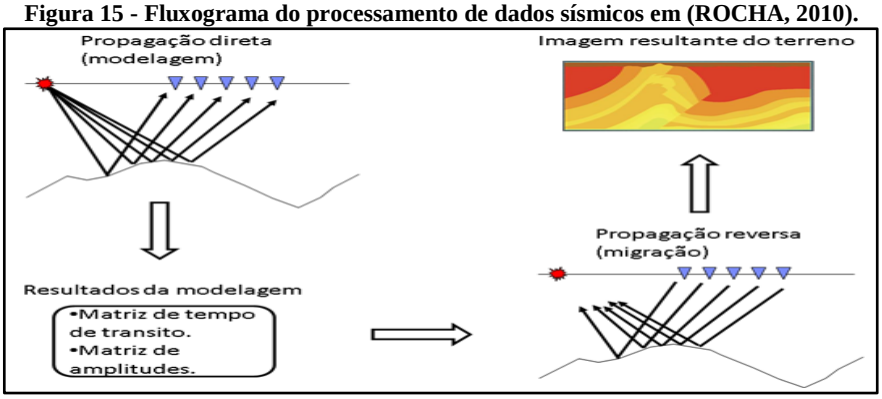
3.3 Modelagem de uma Plataforma Reconfigurável para modelagem 2D, em Sísmica, Utilizando FPGA - Rocha, Rodrigo C. F. (2010).

A pesquisa feita por Rocha (2010) realiza a implementação do problema de modelagem sísmica 2D em *FPGA* . Para tanto, desenvolveu-se uma plataforma reconfigurável baseada em *FPGA* que utiliza uma plataforma da GiDEL, denominada PROCe-III (GIDEL, 2015). O trabalho de Rocha adota um modelo co-design, que utiliza uma unidade de software representada por uma CPU e, um *FPGA*, representando o componente de hardware, como um coprocessador.

Também é exposto no trabalho de Rocha (2010) os processos envolvidos nas etapas de modelagem e migração presentes no algoritmo de *RTM*.

Na figura 15 a seguir observa-se o fluxograma do processamento de dados sísmicos utilizado. Inicialmente é feita a etapa de modelagem e os dados resultantes desse passo são armazenados. Em seguida, após a execução da modelagem, a migração se inicia e cada passo de processamento dos dados gerados pela modelagem são correlacionados com os dados que são gerados pela migração. Por fim, o resultado da etapa de migração corresponde a uma imagem do terreno de interesse e suas camadas de subsuperfície.

Figura 15 - Fluxograma do processamento de dados sísmicos em (ROCHA, 2010).



Fonte: (ROCHA, 2010, P. 12.)

[15] Bit utilizado para representar se um determinado conjunto de bits é maior do que zero.

A equação de diferenças finitas utilizada por Rocha (2010) que faz uma aproximação da equação de onda acústica pode ser vista na equação 3.1 a seguir. Ela corresponde à resolução da equação de diferenças finitas, para cada ponto do modelo, ao longo dos passos de tempo (ou passos de processamento).

$$\begin{aligned} C_{i,j} = {} & 2 * B_{i,j} - A_{i,j} \\ & - \left\{ Vel_{i,j}^2 * fat \right. \\ & * \left[16 * \left(B_{i,j+1} + B_{i,j-1} + B_{i+1,j} + B_{i-1,j} \right) - 1 \right. \\ & \left. \left. * \left(B_{i,j+2} + B_{i,j-2} + B_{i+2,j} + B_{i-2,j} \right) - 60 * B_{i,j} \right] \right\} \end{aligned} \qquad (3.1)$$

Onde os elementos $A_{i,j}$, $B_{i,j}$ e $C_{i,j}$ representam, respectivamente, a localização da posição i,j nas matrizes, que representam o campo de onda, no tempo n-1, n e n+1. O termo $Vel_{i,j}$ corresponde a velocidade da propagação da onda na posição i,j da matriz que representa o modelo do terreno. O termo fat corresponde a constante que é calculada em fator de parâmetros escolhidos para a execução do *RTM*, como o espaçamento entre os pontos da malha e outros.

Para se realizar essa operação Rocha (2010), fez uso da implementação do operador de diferenças finitas presente em Fernandes (2010). O operador faz uso do padrão IEEE-754 para números em ponto flutuante para realizar suas operações aritméticas. Ambos os trabalhos foram desenvolvidos dentro de um projeto de convênio de cooperação entre o Centro de Informática da UFPE e a Petrobras.

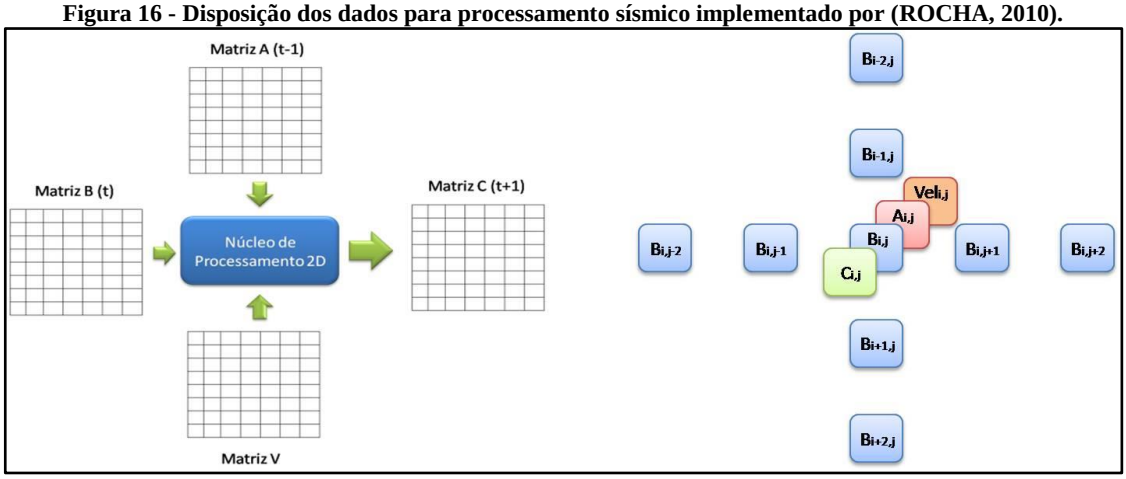
O núcleo aritmético implementado lida com o ruído gerado quando a propagação da onda sísmica atinge as bordas do modelo. Para isso uma nova versão do operador de diferenças finitas foi utilizada.

A versão do operador de diferenças finitas utilizada por Rocha (2010) e implementada por Fernandes (2011) introduziu novos coeficientes (Bl e Bi) que simulam a continuidade da onda quando o processamento dos dados alcançar as bordas da matriz. A equação 3.2 expõe esses novos coeficientes.

$$. C_{i,j} = 2 * (Bl * Bi) * B_{i,j} - (Bl * Bi)^2 * A_{i,j} - (Bl * Bi)^2 * \{Vel_{i,j}^2 * fat * [16 * (B_{i,j+1} + B_{i,j-1} + B_{i+1,j} + B_{i-1,j}) - 1 * (B_{i,j+2} + B_{i,j-2} + B_{i+2,j} + B_{i-2,j}) - 60 * B_{i,j}]\} \quad (3.2)$$

Os valores de Bl e Bi estão dispostos em matrizes e são fornecidos durante o processamento sísmico. A figura 16 a seguir expõe a disposição dos dados de entrada para o núcleo aritmético utilizado por Rocha (2010). A matriz 'A' contém os dados calculados no tempo t-1, a matriz 'B' contém os dados presentes no tempo 't' e a matriz C contém os dados no tempo t+1. A matriz 'V' possui as informações sobre a velocidade de propagação nas camadas do subsolo.

Figura 16 - Disposição dos dados para processamento sísmico implementado por (ROCHA, 2010).

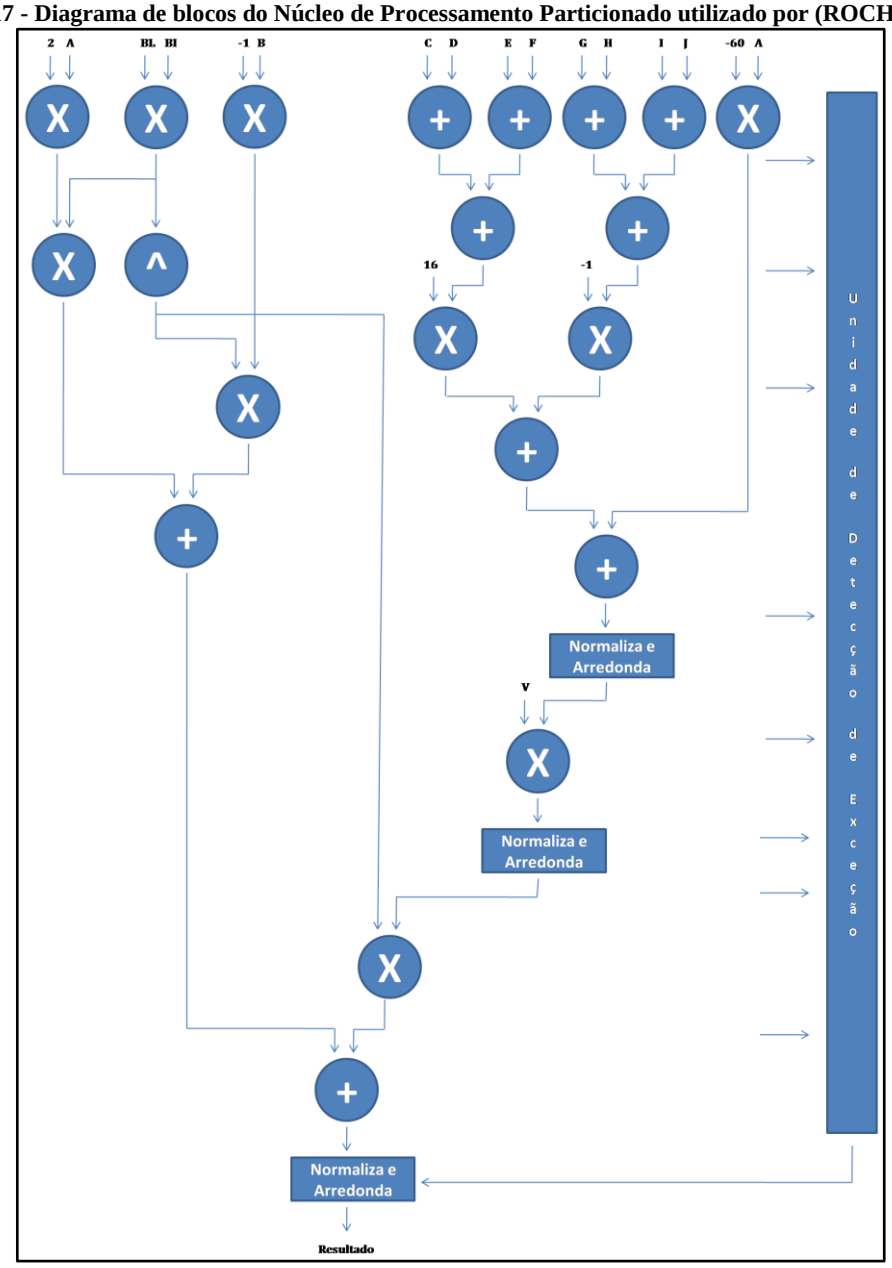


Fonte: (FERNANDES, 2010, P. 22.)

A versão particionada do núcleo aritmético com 36 estágios de *pipeline* utilizado por Rocha pode ser observada na figura 16 a seguir. As letras de 'A' até 'V' colocadas como entrada dos primeiros blocos adicionadores e multiplicadores representam os números presentas na equação (3.2), assim como os coeficientes Bl e Bi na figura apresentados.

Observa-se na figura 17 a opção por usar uma unidade de tratamento das exceções previstas no padrão IEEE-754 que percorre em paralelo a execução de todo o *pipeline*.

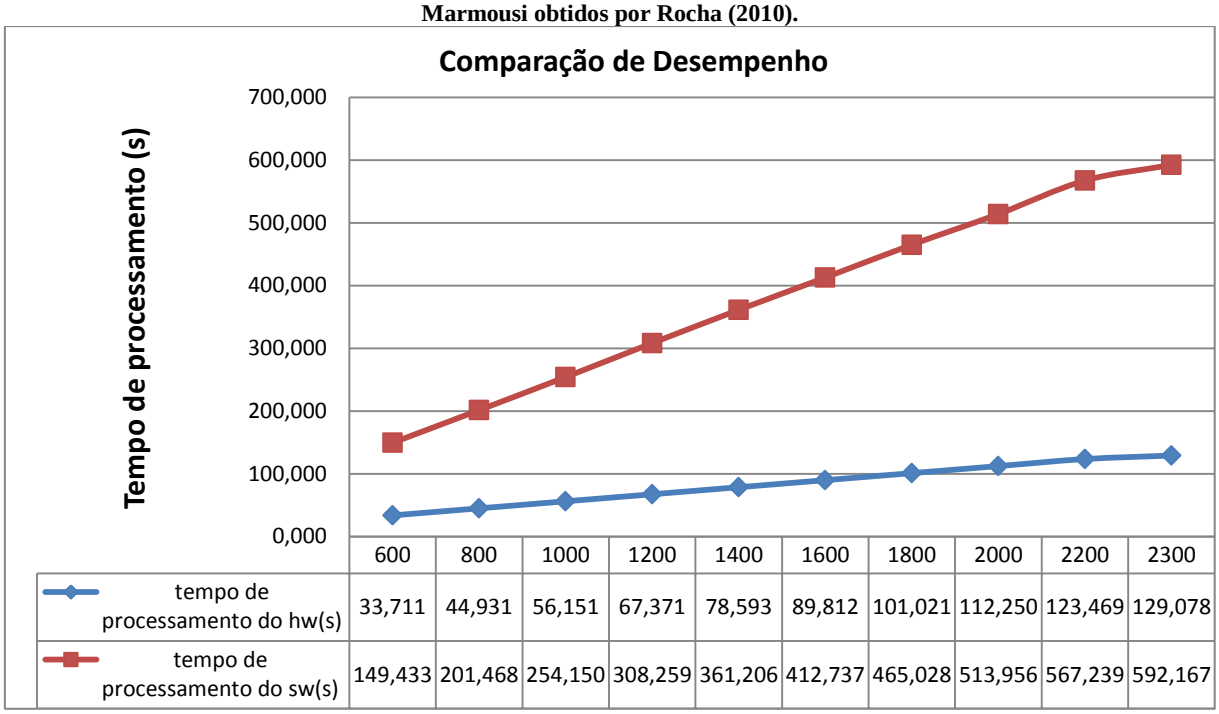
Figura 17 - Diagrama de blocos do Núcleo de Processamento Particionado utilizado por (ROCHA, 2010).



Fonte: (FERNANDES, 2010, P. 52.)

Um dos casos de uso utilizados por Rocha (2010) utiliza o modelo de velocidades de Marmousi (1988). A figura 18 a seguir ilustra um comparativo dos tempos de processamento da plataforma proposta por (2010) e de um modelo de referencia em software. Para tanto, variou-se o número de colunas na matriz de Marmousi e mediram-se os tempos de execução.

Figura 18 - Tempo de processamento da plataforma e do software compilado no Visual Studio 2008 para o modelo de Marmousi obtidos por Rocha (2010).



	600	800	1000	1200	1400	1600	1800	2000	2200	2300
tempo de processamento do hw(s)	33,711	44,931	56,151	67,371	78,593	89,812	101,021	112,250	123,469	129,078
tempo de processamento do sw(s)	149,433	201,468	254,150	308,259	361,206	412,737	465,028	513,956	567,239	592,167

Fonte: (ROCHA, 2010, P.106.)

Observa-se na figura 18 que os tempos de processamento em *hardware* foram menores do que os em *software*. O ganho de velocidade média de processamento foi de cerca de 4,5 vezes.

3.3.1 Conclusões

O trabalho de Rocha (2010) mostrou a implementação de uma plataforma capaz de realizar a etapa de modelagem do algoritmo de *RTM* em *FPGA*. Para tanto fez uso da implementação do núcleo aritmético implementado em Fernandes (2010) que opera com números em ponto flutuante no padrão IEEE-754.

Esse trabalho juntamente com o núcleo aritmético de Fernandes (2011) serviu como uma das principais referências por abordar a implementação do *RTM* em *FPGA*.

A disposição das operações aritméticas de maneira a se aproveitar o paralelismo de execução ocorreu de maneira semelhante à feita nesta dissertação. Esta dissertação também faz uso de uma arquitetura em *pipeline* para ganho de desempenho.

É importante destacar, que o núcleo aritmético implementado por (FERNANDES, 2010) resolve a equação de diferenças finitas para uma modelagem em duas dimensões apenas. Isso difere da equação utilizada nesta dissertação que possui três dimensões espaciais.

Outro ponto de diferença no núcleo aritmético utilizado em (ROCHA, 2010) e esta dissertação é a maneira como ocorre o tratamento de exceções. No utilizado por (ROCHA, 2010) há uma unidade de tratamento de exceções que percorre todo o *pipeline* realizando verificações paralelamente a execução. Nesta dissertação, o tratamento de exceções é feito internamente aos módulos aritméticos antes de se realizar a operação. Isso foi feito para se economizar recursos utilizados no processo de roteamento dos dados dos módulos.

3.4 Trabalhos relacionados gerais.

Além dos trabalhos já citados, outros também contribuíram, ainda que em menor grau, para o desenvolvimento dessa dissertação.

Os trabalhos de Wilson et al. (2014), Ramesh et al. (2013), Amaricai et al. (2013) abordam implementações de operações aritméticas com números em ponto flutuante em *FPGA*. Pode-se observar a implementação de operações de multiplicação, soma, raiz quadrada e no caso de Amaricai (2013) de números em notação de ponto fixo. Esses trabalhos contribuíram para essa dissertação na metodologia de desenvolvimento de alguns módulos da arquitetura e disposição das operações ao longo da execução do *pipeline*.

Em Govindo et al. (2005) é exposta uma biblioteca no padrão IEEE-754 contendo as operações de soma, subtração, divisão, multiplicação e raiz quadrada para números em ponto flutuante. Os diagramas funcionais com a indicação dos módulos internos são apresentados e bem descritos, porém não apresentados detalhes internos de implementação.

O trabalho presente em Barros (2014) provê um extenso estudo da precisão numérica necessária à implementação do algoritmo de *RTM*. Ele contribuiu com esta dissertação durante a etapa de desenvolvimento dos núcleos em ponto fixo e na definição do número de bits necessários para sua representação.

A pesquisa desenvolvida por Santos (2012) compara as etapas de modelagem da equação de onda e a migração reversa no tempo (*RTM*), utilizando os métodos das diferenças finitas (MDF) e elementos finitos (MEF). Esse trabalho provê fundamentação matemática para a aproximação da equação de onda pelo método de diferenças finitas.

3.5 Conclusões

Este capítulo apresentou alguns trabalhos que possuem relação com essa dissertação.

A estratégia de quebra do fluxo de operações ao longo de um *pipeline* presente em Dou (2005) permitiu o aumento da frequência de operação do núcleo aritmético. Também foi utilizado nessa dissertação um núcleo de contagem de zeros durante a normalização, apesar de o algoritmo utilizado por Dou (2005) ter sido diferente.

Nessa dissertação utilizou-se a mesma estratégia de arredondamento e divisão das operações aritméticas ao longo de um *pipeline* feitas por Barros (2008). Elas foram implementadas de maneira a se explorar o paralelismo entre as instruções o que permitiu uma melhoria na execução do algoritmo.

O núcleo aritmético desenvolvido em Fernandes (2010) e implementado por Rocha (2010) serviu com inspiração para o desenvolvimento do núcleo aritmético deste trabalho. No entanto, a aproximação da equação de onda desta dissertação é uma extensão da utilizada por Rocha (2010), visto que trabalha com as três dimensões espaciais ao invés de apenas duas.

Os trabalhos Wilson et al. (2014), Ramesh et al. (2013), Amaricai et al. (2013), Govindo et al (2005), Barros (2014), Santos (2012) também possuem relação com as temáticas de aritmética de ponto flutuante e *RTM* e apresentaram contribuição para essa dissertação.

Apesar dos diversos trabalhos correlatos a esta dissertação, não foram encontrados projetos que abordassem a implementação de um núcleo aritmético, padrão IEEE-754, capaz de resolver o operador de diferenças finitas do algoritmo *RTM* em três dimensões para *FPGA*s.

Capítulo

4

4. Plataforma para modelagem 3D do algoritmo de RTM em FPGA

Esse capítulo descreve a plataforma desenvolvida pelo projeto HPCIn em parceria com o CENPES no Centro de Informática da UFPE para a etapa de modelagem do algoritmo de análises sísmicas, RTM 3D, para hardware reconfigurável, *FPGA*.

4.1 Ambiente de implementação

Os módulos aritméticos desenvolvidos nesse trabalho foram utilizados para resolver a equação de ondas sísmicas que compõem o método de *RTM*. Para solucionar o método de avaliação do solo por essa técnica, foi feita uma arquitetura capaz de funcionar em *FPGA*.

A etapa de modelagem do método de *RTM* foi escolhida como exemplo por apresentar menor complexidade de implementação do que a etapa de migração.

Na etapa de modelagem ocorre o processamento direto da equação de onda através de um campo de velocidades do solo. Este implementado em parceria com o CENPES no projeto HPCIn do CIn da UFPE (2014).

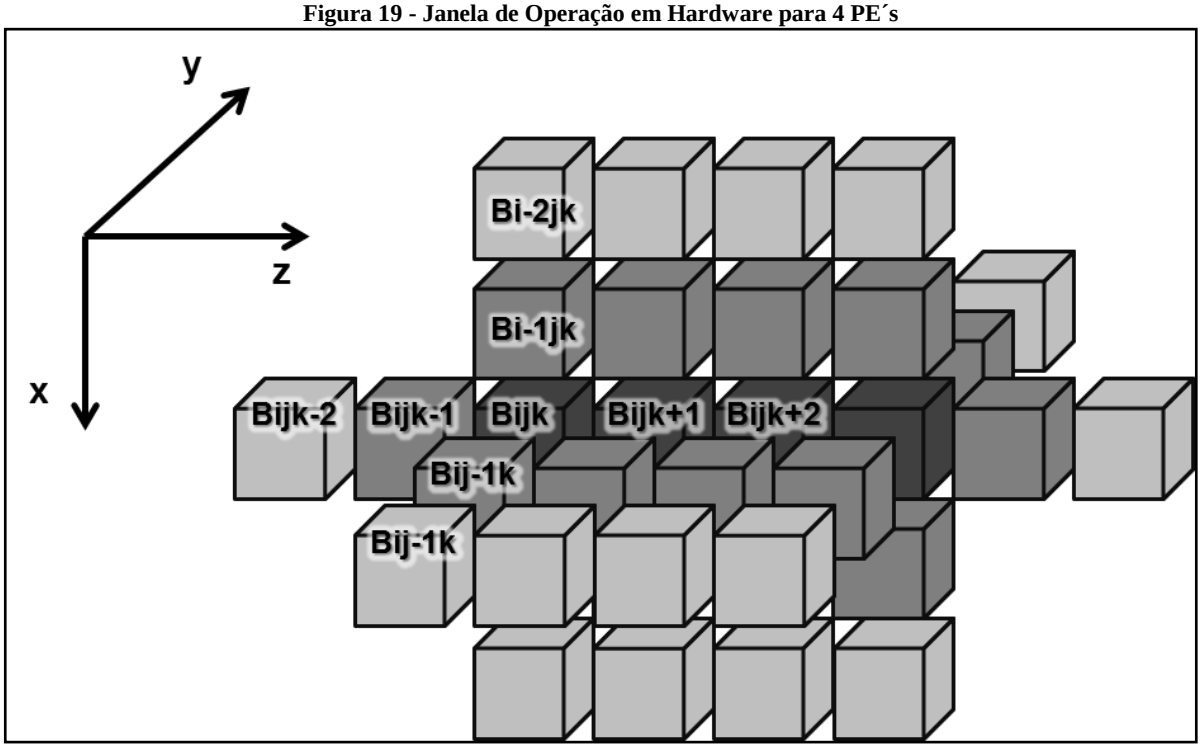
A arquitetura desenvolvida pode ser observada no anexo III. Nela estão presentes os módulos que fazem comunicação com os dados nas memórias externas ao *FPGA*, registradores de deslocamento, módulos de leitura e escrita, de geração de pulso, entre outros. Além desses módulos, estão presentes componentes de lógica de controle e até componentes lógicos comuns, como multiplexadores e portas lógicas.

Essa arquitetura em hardware foi desenvolvida para se explorar o paralelismo das operações. Em um primeiro momento, foi explorado o paralelismo que trata do número de instruções simultâneas executadas, ou seja, quantas janelas de processamento são executadas em paralelo. A janela de processamento utilizada foi implementada com o *PE* e a arquitetura desenvolvida dá suporte a 4 desses PE´s. Com isso, têm-se 4 operações realizadas por ciclo de relógio.

Inicialmente, foi utilizado o PE Float como parâmetro de comparação durante a execução do fluxo da arquitetura. Em seguida, o mesmo foi feito com o PE Híbrido. Ao final, o PE Híbrido foi escolhido para integrar a arquitetura, pois o mesmo gera uma economia de blocos lógicos do *FPGA.*

Na figura 19, pode-se observar a maneira como os 4 *PEs* operam. Eles percorrem uma matriz tridimensional do campo de pressão atual. Primeiramente, a matriz é percorrida no sentido da coordenada z, depois no sentido da coordenada y, e por último, no sentido do eixo x.

Figura 19 - Janela de Operação em Hardware para 4 PE´s



Pode-se observar, também, na figura 19, os termos de operação citados na arquitetura do *PE*. Esses termos correspondem às entradas para o núcleo aritmético desenvolvido, sendo *Bijk* o termo central, *Bijk-1*,*Bij-1k* e *Bi-1jk* os termos diretamente adjacentes, e os termos *Bijk-2, Bij-2k* e *Bi-2jk* com distância 2 do termo central. Os quatro *PEs* presentes na figura 19 compõem um *time step*, ou passo temporal de execução.

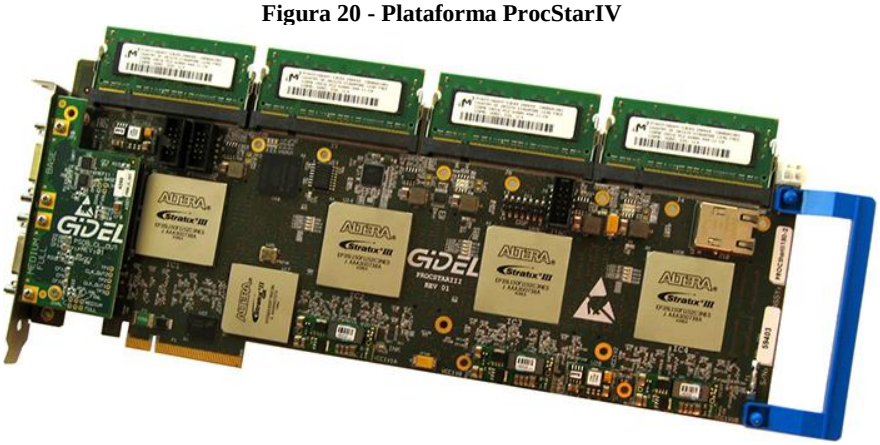
Nessa arquitetura implementada em *FPGA*, também é explorado um paralelismo temporal de operações. Cada grupo de 4 *PEs* opera sobre um *time step* gerando resultados que são armazenados em *FIFOs* internas. Essas *FIFOS* contêm, então, os dados que servem de entrada para o segundo grupo de 4 *PEs* e assim por diante. Esse fluxo segue até se chegar ao quarto grupo, onde sua saída passa a ser escrita na memória. Com isso, nesse algoritmo está presente um conjunto com 16 *PEs*, que processam ao mesmo tempo 4 *time steps*, cada passo processando 4 janelas de processamento.

A plataforma de desenvolvimento em *FPGA* corresponde a *ProcStarIV*, da *Gidel* (2014), que é equipada com 4 *FPGAs Stratix IV* da *Altera* (Altera, 2015). Cada *FPGA* é alimentada por um módulo de memória com 512MB *DDR2*, além de dois *slots* para memórias *DDR2* que podem chegar até a 4GB. No caso de uso, utilizou-se dois módulos de 4GB para cada um dos *FPGA*s. Para a geração de componentes de integração na placa foi utilizada a ferramenta *ProcWizard* (Gidel, 2015).

O sismograma gerado ao final dessa operação opera com matrizes de tamanho máximo 2048x2048x2048, fornecendo assim, 32 GB de números em ponto flutuante de precisão simples. Devido a grande quantidade de dados, o problema foi dividido em partições que cabiam na *FPGA* utilizada, de tamanho 2048X48X48. Ao final de execução de uma partição desse tamanho, uma nova fatia da matriz de entrada de mesmo tamanho é executada. Isso ocorre até que todas as partições da matriz sejam executadas.

Na figura 20 a seguir, pode-se observar a plataforma de desenvolvimento utilizada para a aplicação do caso de uso em *FPGAs*.

Figura 20 - Plataforma ProcStarIV



Capítulo

5

5. Um Núcleo aritmético para execução de um operador de diferenças finitas em FPGA

Nesta seção aborda-se o desenvolvimento de um núcleo aritmético capaz de calcular o operador de diferenças finitas presente no algoritmo de RTM.

5.1 Visão Geral

Nessa seção está exposta a arquitetura desenvolvida para se calcular o operador de diferenças finitas para a equação de onda presente no mapeamento sísmico do solo via *RTM*.

As soluções desenvolvidas recebem, como entrada, números em formato de ponto flutuante padrão IEEE-754 e geram, em sua saída, números no mesmo formato.

Inicialmente, foi feita uma versão que soluciona a equação do operador de diferenças finitas, para números em ponto flutuante no padrão IEEE-754. Essa versão foi denominada de PE Float..

Em seguida, foi feita outra versão visando à redução da quantidade de elementos lógicos utilizados em *FPGA* para a implementação do módulo. Essa versão foi denominada de PE Híbrido. Esse módulo opera tanto com números em ponto flutuante quanto em ponto fixo.

Isso foi feito porque a notação em ponto fixo é mais econômica, tanto em custos de área, quanto em complexidade de cálculos. No entanto, isso só foi possível devido à natureza dos números utilizados em algumas das variáveis de entrada, pois a notação em ponto fixo é bem mais limitada em relação ao alcance de representação numérica. Esse módulo aritmético foi otimizado em relação à área utilizada sem ter impactos significativos quanto à precisão esperada para o resultado.

Ambos os PE´s desenvolvidos fazem uso de uma arquitetura em *pipeline*. Isso foi feito tanto para se aumentar a frequência de operação do núcleo aritmético quanto para que houvesse a execução de mais de uma instrução simultaneamente.

O PE Float e o PE Híbrido compartilham diversos módulos. Ambos são divididos em três grandes blocos, denominados de Bloco1, Bloco2 e Bloco3.

Os Bloco1 e Bloco3 são idênticos em ambas as arquiteturas, pois esses módulos não sofreram modificações de notação da representação adotada. Com isso, a diferença entre o PE Float e o PE Híbrido reside, basicamente, no Bloco2, também denominado de árvore de somas. No PE Float, esse bloco opera com os números em ponto flutuante, já no PE Híbrido, o bloco opera com números representados em ponto fixo.

Com a finalidade de simplificar esse documento, o Bloco1 e o Bloco3 são explicados apenas na parte que aborda o PE Float, visto que são iguais nas duas arquiteturas. No entanto, o Bloco2 é explicado tanto na descrição do PE Float, quanto na do PE Híbrido, devido às suas diferenças em cada PE.

A estrutura dos PEs foram desenvolvidas para se calcular a equação do operador de diferenças finitas descrita abaixo:

$$Abs\_i * Abs\_j * Abs\_k * (2.0 * B_{ijk} + Abs_i * Abs_j * Abs_k * (VEL * VEL * FAT * (-1 * (B_{ij-2K} + B_{ij+2K} + B_{i-2jK} + B_{i+2jk} + B_{ijk-2} + B_{ijK+2}) + 16 * (B_{ij-1K} + B_{ij+1K} + B_{i-1jK} + B_{i+1jk} + B_{ijk-1} + B_{ijK+1}) + (-90 * B_{ijk})) - A_{ijk})) \quad (4.1)$$

Essa equação teve sua estrutura alterada para operar de maneira mais eficiente em hardware. Essa manipulação matemática permite reuso de operações e abre caminho para otimizações. A equação assume, então, a seguinte forma representada abaixo:

$$(2.0 * B_{ijk} * Abs_i * Abs_j * Abs_k) + ((Abs_i * Abs_j * Abs_k)^2 * (VEL * VEL * FAT * (-1 * (B_{ij-2K} + B_{ij+2K} + B_{i-2jK} + B_{i+2jk} + B_{ijk-2} + B_{ijK+2}) + 16 * (B_{ij-1K} + B_{ij+1K} + B_{i-1jK} + B_{i+1jk} + B_{ijk-1} + B_{ijK+1}) + (-90 * B_{ijk})) - A_{ijk})) \quad (4.2)$$

Os PE´s implementam a equação 4.2, com isso eles só precisam realizar a multiplicação entre as variáveis Abs_i, Abs_j, Abs_k uma única vez antes de sua resposta seguir pelo *pipeline*. Seu resultado é propagado através de registradores de deslocamento.

Com a manipulação da equação 4.1 para a 4.2 introduziu-se uma operação de quadrado com o termo $(Abs_i * Abs_j * Abs_k)^2$. Isso permitiu que mais operações fossem feitas em paralelo a um custo relativamente baixo, pois a operação de quadrado é uma operação mais simples do que uma multiplicação padrão em ponto flutuante.

A figura 21 a seguir ilustra em alto nível a arquitetura dos PE´s implementados. Pode-se observar a presença dos três grandes blocos que compõem todo o *pipeline* da arquitetura.

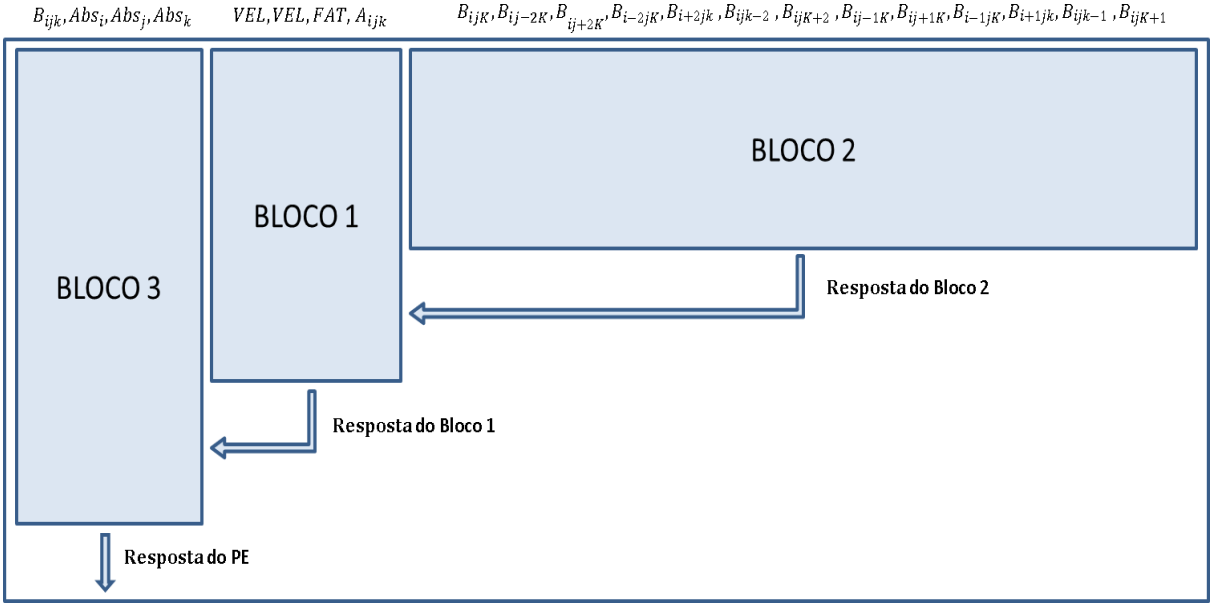
Observa-se na figura 21 que o Bloco2 recebe como entradas os termos: $B_{ijK}, B_{ij-2K}, B_{ij+2K}, B_{i-2jK}, B_{i+2jk}, B_{ijk-2}, B_{ijK+2}, B_{ij-1K}, B_{ij+1K}, B_{i-1jK}, B_{i+1jk}, B_{ijk-1}$ e B_{ijK+1}.

Pode-se observar também na figura 21 que o Bloco1 opera com os termos: VEL, VEL, FAT e A_{ijk}.

A figura 21 ilustra também que o Bloco3 opera com os termos: B_{ijk}, Abs_i, Abs_j e Abs_k.

O Bloco1 também recebe como entrada o resultado advindo do Bloco2. O Bloco3 recebe também como entrada o resultado advindo do Bloco1. A saída final do núcleo de processamento em ambos os PE´s (PE Float e PE Híbrido) se dá através da saída do Bloco3.

Figura 21 - Visão geral da arquitetura dos PE´s.



Toda a arquitetura esta disposta em um grande *pipeline*. Com isso, os três blocos podem operam simultaneamente enquanto processam seus dados. A sincronização dos dados entre os blocos é feita através de registradores de deslocamento presentes dentro da estrutura do *pipeline*.

Como o PE Float e o PE Híbrido possuem a mesma estrutura dos Bloco1 e Bloco2, seus subcomponentes básicos serão explicados logo nesta seção. Os subcomponentes que são exclusivos do PE Híbrido, ou seja, os que integram o Bloco2 serão descritos na seção que expõe a arquitetura do PE Híbrido.

A seguir são descritos alguns dos módulos internos básicos que são instanciados inúmeras vezes por blocos maiores tanto do PE Float quanto PE Híbrido.

5.1.1 Conversor

Esse componente é responsável por ajustar os números do formato IEEE-754 de trinta e dois bits, para uma notação interna com mais bits. Essa notação aplica um bit implícito na parte mais significativa da mantissa e ajusta o expoente para ser operado internamente. O bit implícito é adicionado pois o padrão assume que os números em ponto

flutuante estão normalizados. Para se realizar as operações aritméticas subsequentes com um número em ponto flutuante o mesmo deve ser operado já com o bit implícito, caso contrário a operação é tida como incorreta

Esse módulo possui um único estágio de *pipeline*.

5.1.2 Somador

Esse componente realiza uma operação de soma para números em uma notação estendida. Essa notação é resultado de operações prévias no PE, em que foi necessário o aumento do número de bits para se preservar a precisão da operação. Esse fenômeno acontece, por exemplo, após uma operação de multiplicação, que possui a resposta com uma quantidade de bits equivalente as somas das quantidades de bits das variáveis de entrada. No caso da soma, o significando de resposta deve possuir uma quantidade de bits equivalente à quantidade de bits dos significandos de entrada mais um.

No primeiro estágios do *pipeline* dessa unidade, é feita uma verificação das exceções previstas no padrão IEEE. Isso se faz necessário, pois a operação com números desta natureza gera um resultado inconsistente sob o ponto de vista do padrão. As verificações feitas são quanto a infinitos positivos e negativos, *NANs* e zeros. Além desses, é verificada a presença de números *denormais*. No entanto, para fins de economia de recursos de hardware, esses números, que, por natureza, são muito pequenos, foram arredondados para zero. Após essa etapa, segue-se com o algoritmo padrão de soma para números com notação de sinal e magnitude.

As exceções geram um código interno de exceção no somador que ao longo dos demais estágios do *pipeline* são verificados para não serem operados da mesma maneira que um número normal. Por exemplo, ao se identificar um NaN no primeiro estágio esse deve ser propagado ao longo dos demais estágios de *pipeline,* pois qualquer soma de um número com um não válido gera como resposta o próprio número não válido (NaN).

No segundo estágio é feita a conversão para uma notação de complemento a dois para que os números sigam para a etapa seguinte para serem somados.

Ainda no segundo estágio é feito o cálculo da diferença entre os expoentes de entrada paralelamente a operação de complemento a dois.

No estágio seguinte realiza-se um deslocamento dos números complementados a dois de acordo com a diferença obtida entre os expoentes no estágio anterior. Esse deslocamento é feito para a direita no significando do operando que tem o menor expoente. Essa etapa

acontece para que os pontos separadores estejam alinhados e bits de mesma magnitude sejam somados entre si.

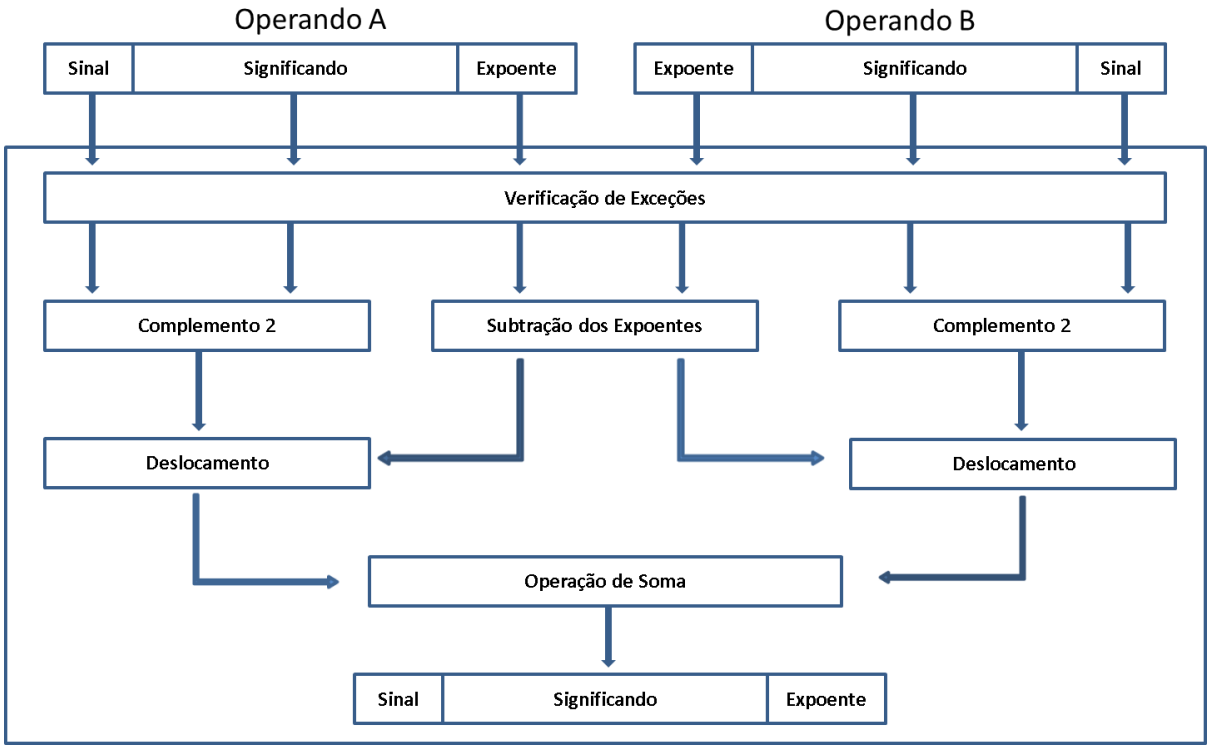
A operação de deslocamento de um número à direita equivale a dividi-lo pela base numérica adotada. A fim de manter o valor originalmente representado pelo significando que está sendo deslocado, a cada deslocamento deve-se incrementar o valor do expoente associado.

No último estágio realiza-se a operação de soma entre os significandos que foram convertidos para a notação de complemento a dois nos estágios anteriores.

Essa unidade ao todo possui quatro estágios de *pipeline*.

Pode-se observar o fluxo de execução das operações de soma e subtração na figura 22 a seguir. Nesta figura não estão indicados os registradores de *pipeline* para facilitar seu entendimento. Porém, os mesmos estão presentes e o leitor deve estar consciente de sua existência.

Figura 22 - Fluxo de execução das operações de soma e subtração do PE



A etapa de normalização e arredondamento são feitas em um outro módulo que é descrito mais adiante.

5.1.3 Multiplicador

Esse componente realiza uma operação de multiplicação de acordo com o padrão IEEE-754, em ponto flutuante. Essa unidade recebe dois números na notação de sinal e magnitude, e gera uma resposta com uma notação de bits expandida.

Assim como descrito no somador, é necessária a verificação de números que representam exceções no início de suas operações. Esses números são identificados, e a operação entre eles é ajustada para se gerar um resultado esperado pelo padrão. Após a etapa de verificação inicial de exceções, segue-se com o algoritmo de multiplicação.

No estágio seguinte os expoentes são somados entre si e em paralelo é feita a operação de multiplicação dos significandos de entrada.

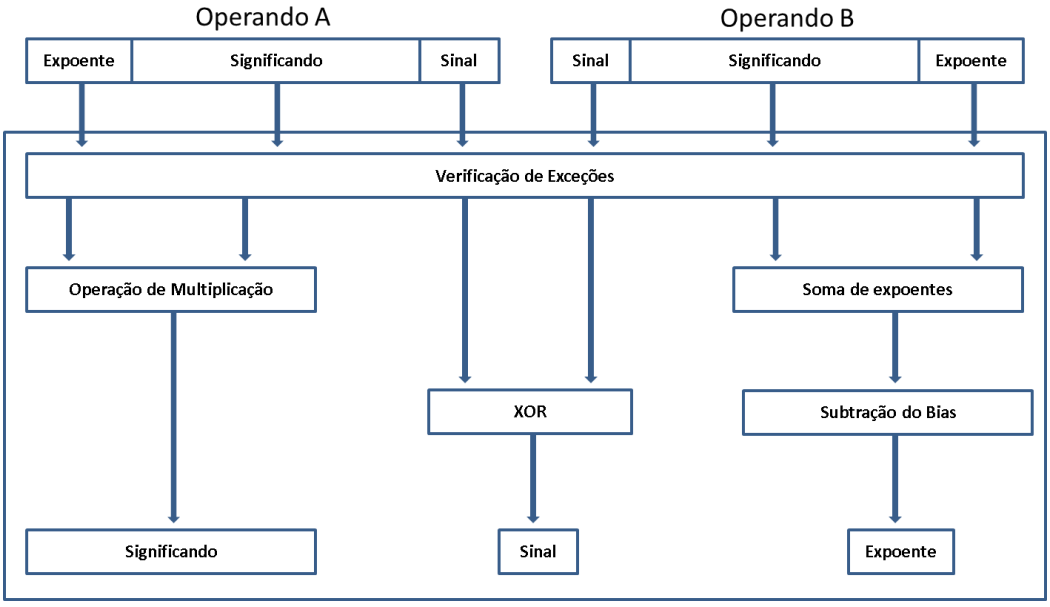
Após esse estágios o *bias* é subtraído do resultado da operação de soma de expoentes. Os expoentes já estão cada um com um *bias* e, portanto, quando somados, geram um número que possui dois *bias*, o que seria um resultado incorreto.

Em paralelo a subtração do bias é feita uma operação lógica de XOR (*exclusive OR)* entre os sinais de entrada para a geração dos sinais de saída.

Essa unidade ao todo possui três estágios de *pipeline*.

Pode-se observar o fluxo de execução da operação de multiplicação na figura 23. Nesta figura não estão indicados os registradores de *pipeline* para facilitar seu entendimento. Porém, os mesmos estão presentes e o leitor deve estar consciente de sua existência.

Figura 23 - Fluxo de execução da operação de multiplicação do PE.



Assim como no somador, a etapa de normalização e arredondamento é feita em um outro módulo que é descrito mais adiante.

5.1.4 Normalizador_Arredondador

Essa unidade é responsável por realizar a normalização dos números depois das operações aritméticas e, em seguida, realizar seu arredondamento.

Uma das operações que esse módulo precisa realizar para se normalizar um número é uma contagem de zeros na parte mais significativa do vetor de bits. Essa etapa identifica a quantidade de deslocamentos que serão necessárias para se normalizar o número.

O deslocamento é feito de maneira que, ao final, haja um número ‘1’ na primeira posição a esquerda do ponto separador. Caso essa distância esteja à esquerda da posição esperada a distância é considerada positiva, caso esteja a direita a distância é considerada negativa. É importante destacar que qualquer deslocamento feito na mantissa, seja ele para esquerda ou direita, implica em uma alteração no valor armazenado no expoente.

A etapa seguinte ao processo de normalização, a ser realizada nesse módulo, é o arredondamento. Os bits que excedem a notação de 22 bits de mantissa prevista pelo IEEE devem ser retirados do significando.

O arredondamento utilizado foi para o próximo ou par. Caso alguns dos bits excedentes do significando apresentem valor diferente de zero, o número deve ser arredondado para o próximo. Isso é feito somando-se ‘1’ ao significando.

Se nenhum valor diferente de zero é encontrado nos bits excedentes do significando, o número pode ser simplesmente truncado, ou seja, é realizado um arredondamento para zero.

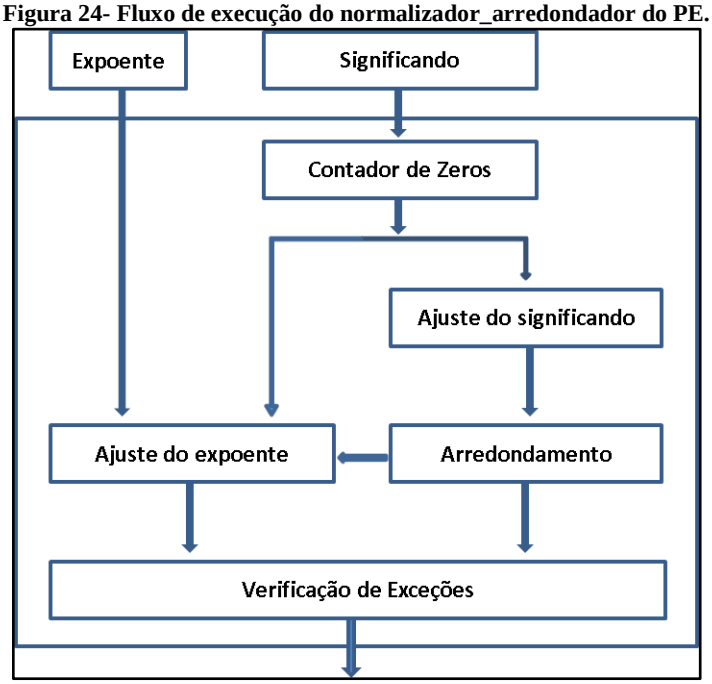
A etapa final é a de verificação de exceções. Essa etapa serve para sinalizar as exceções previstas no padrão IEEE para os próximos módulos do PE.

Ao final das operações o número estará na notação de ponto flutuante 32 bits prevista no padrão IEEE-754,

Esse módulo apresenta quatro estágios de *pipeline*.

Pode-se observar o fluxograma geral de execução do normalizador e arredondador na figura 24 a seguir. Os deslocadores de deslocamento (FIFOs) responsáveis pela sincronia dos sinais internos foram omitidos para simplificação da ilustração.

Figura 24- Fluxo de execução do normalizador_arredondador do PE.



5.1.5 Quadrado

Este módulo é responsável por realizar a multiplicação de um número por ele mesmo, ou seja, a operação de quadrado.

Para ser realizar a operação de quadrado, é feita, inicialmente, uma verificação de exceções no primeiro ciclo do modulo. Esses números são identificados para não interferirem nos resultados esperados pelo módulo. O caso do zero, por exemplo, exige que a resposta gerada pelo módulo seja também zero, não sendo necessário o cálculo para se obter essa resposta.

No segundo estágio é feito o cálculo do expoente de resposta. Para tanto, seria necessário a realização de uma soma de expoentes, seguindo o algoritmo de multiplicação normal. Contudo, como têm-se uma soma de expoentes iguais, a operação na verdade equivale a dobra-se o valor desse expoente. A multiplicação por dois em base binária equivale a um deslocamento para a esquerda (*shift left*) de uma posição no vetor de bits que representa o expoente. Para simplificar ainda mais esse processo, o deslocamento para a esquerda foi substituído por um mapeamento dos bits mais significativos do expoente de tamanho ‘n’ em um expoente de resposta de tamanho ‘n+1’. Com isso um bit zero é adicionado na parte menos significativa desse novo expoente.

O mapeamento do expoente foi incorporado ao primeiro ciclo de *pipeline* do módulo quadrado. Essa estratégia equivale a se deslocar o vetor de bits para a esquerda sem perda de bits no deslocamento. Esse expoente com mais bits passa pelo módulo de quadrado sendo tratado fora dele.

É importante destacar que a soma de dois expoentes exige uma subtração do *bias*. Isso é necessário pois os dois expoentes somados já tem o *bias* embutido em cada um de seus valores e sua soma geraria uma número com dois bias embutido, o que não é previsto pelo padrão IEEE-754.

O processo de subtração do *bias* do expoente é feito no segundo estágio desse módulo.

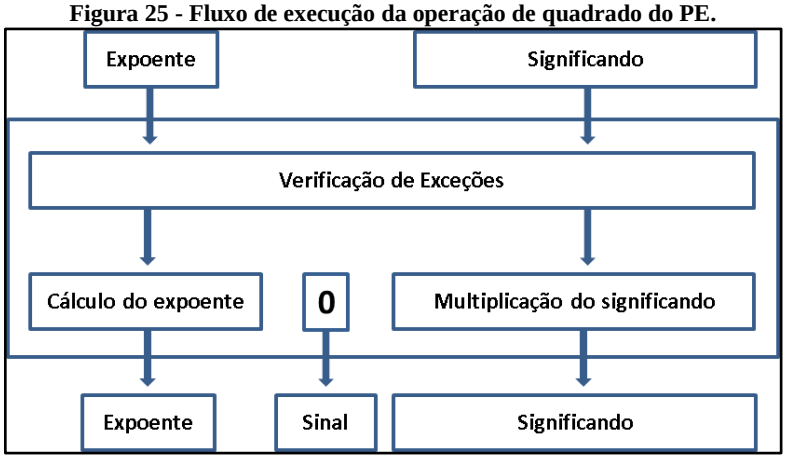
No segundo estágio também ocorre em paralelo a multiplicação dos significandos. A quantidade de bits resultante dessa operação, especificamente, deve ser o dobro da quantidade de bits do significando inicial. Isso se faz necessário porque a operação de multiplicação precisa de mais bits de resposta para não perder precisão. O significando de resposta sai com essa notação de bits expandida. O processo de arredondamento desse significando é tratado fora desse módulo em um módulo denominado de arredondador.

A definição do sinal no módulo quadrado é uma constante e possui valor zero. Isso se torna evidente porque, devido à natureza da operação de quadrado, os resultados serão sempre positivos ou nulos.

Essa unidade possui apenas dois estágios de *pipeline*.

Pode-se observar o fluxo de execução da operação de quadrado na figura 25 a seguir.

Figura 25 - Fluxo de execução da operação de quadrado do PE.



5.1.6 Arredondador

Essa unidade é responsável por arredondar os valores que saem do módulo quadrado. O arredondamento utilizado nessa unidade é o arredondamento para o próximo ou par definido pelo padrão IEEE-754.

No primeiro estágio de *pipeline* desse módulo verifica-se se os bits excedentes do significando são maiores do que zero. Entende-se por bits excedentes aqueles que estão além da representação de 23 bits do significando normalizado definido no padrão IEEE.

Os bits excedentes quando maiores do que zero implicam que o significando deve ser arredondado para cima, pois essa operação introduz um menor erro do que o gerado caso o número fosse arredondado para baixo. Esse arredondamento para cima é feito realizando-se a soma do significando com ‘1’.

O segundo estágio faz as verificações e ajustes no expoente.

É feito no segundo estágio o processo de arredondamento do expoente de entrada para a diminuição de um bit em sua notação. Para tanto é tratado se esse processo estoura a capacidade máxima de representação do expoente de 8 bits, padrão para o ponto flutuante de precisão simples IEEE-754.

5.1.7 FIFOs

As unidades denominadas de FIFOs (*First in, Firs Out*) são apenas registradores de deslocamento ao longo do *pipeline*.

O tamanho das FIFOs varia de acordo com o módulo no qual ela está inserida.

Essas unidades estão presentes ao longo de todo o *pipeline* dos PE´s e servem para garantir a devida sincronia entre seus sinais.

5.2 PE Float

O PE Float opera com números em ponto flutuante no padrão IEEE-754. As operações aritméticas implementadas nesse PE obedecem às normas previstas no padrão. Dentre essas normas destaca-se o tratamento de exceções, arredondamento e normalização.

Apesar de operar com números em ponto flutuante, o PE Float internamente modifica essa representação para que as operações aritméticas sejam realizadas. Cada termo do número em ponto flutuante, sinal, expoente e mantissa são operados separadamente. Ao

final de suas respectivas operações aritméticas os números retornam a notação padrão em ponto flutuante.

Para que o grande número de operações presentes na equação 4.2 fosse realizado, foi necessária a repartição das operações em módulos menores. Essa divisão em componentes obedeceu aos critérios de similaridades das suboperações da equação. Isso foi feito procurando-se preservar um subconjunto de somas em um mesmo módulo e um subconjunto de multiplicações em outro módulo. Com isso, a equação de diferenças finitas pode ser implementada em hardware de maneira mais eficiente.

Pode-se observar na figura 53 do anexo II, o diagrama em blocos com a estrutura geral do *PE3D Float*.

Como dito anteriormente, o PE Float está dividido em três módulos principais denominados Bloco1, Bloco2 e Bloco3. Os componentes desses módulos principais estão divididos em blocos menores para operações específicas e são descritos a seguir.

5.2.1 Bloco1

O Bloco1 é responsável por realizar o seguinte trecho da equação de diferenças finitas.

$$\left(VEL * VEL * FAT * (Bloco2_{out}) - A_{ijk}\right) \qquad (4.3)$$

Esse bloco recebe, como entrada, os números VEL, FAT, $Bloco2\_out$ e A_{ijk}. Todas as suas entradas, com exceção da advinda do Bloco2, são de regiões externas ao *PE*. Com exceção do temo VEL, todas as entradas estão na representação padrão IEEE-754 de 32 bits.

A entrada *Bloco2_out* representa a saída do Bloco2, que é necessária para a operação do Bloco1. Nesta versão da arquitetura, o PE Float, essa saída encontra-se na notação de ponto flutuante padrão IEEE-754 com 32 bits, visto que, após a operação no Bloco2, os números são normalizados e arredondados para a notação adotada.

Para se operar com os números que representam a velocidade de propagação da onda acústica no meio, o *VEL*, é necessária uma etapa de conversão. Essa etapa é importante, pois o número está em notação de *half-precision*, ou seja, com apenas 16 bits, um número menor do que os 32 bits previstos no padrão.

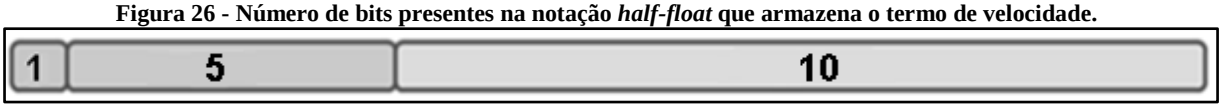
Os números em notação de *half-precision* possuem os mesmos termos de um número em ponto flutuante de 32 bits, sinal, expoente e mantissa. A diferença entre as duas

notações se encontra apenas no número de bits utilizados, o que modifica o alcance de representação numérica e a precisão dessa representação.

Na figura 26 a seguir pode-se observar a distribuição dos bits na notação *half-float*. Essa notação possui ‘1’ bit para sinal, ‘5’ bits para expoente e ‘10’ bits para mantissa.

Com essa notação é possível se representar números com valores até 2^{15}, ou seja, 32.768 com uma precisão máxima de 2^{-10}, ou seja, 0,0009765625.

Figura 26 - Número de bits presentes na notação *half-float* que armazena o termo de velocidade.



A opção pela representação do termo VEL em notação *half-float* foi motivada devido a menor variação de amplitude e precisão deste número em relação aos demais termos da equação 4.2. Nos casos de uso fornecido pelo CENPES (PETROBRAS, 2015) para o grupo HPCIn (HPCIN, 2014) os campos de velocidade não ultrapassavam o limiar de velocidade máxima de 2^{15}. Tampouco os limiares de velocidade ultrapassavam a precisão de 2^{-10}. Com isso, foi possível representar os números do termo VEL da equação 4.2 com metade dos bits dos números em ponto flutuante padrão de 32 bits.

Essa foi uma exigência ditada pela arquitetura de controle e comunicação presentes no algoritmo de *RTM* que foi implementado em *FPGA* pelo grupo HPCIn, mas que fogem ao escopo desta dissertação.

Devido à presença do *half-float* foi necessário o acréscimo de um conversor para a notação de 32 bits na entrada do Bloco2 para o termo VEL. Com isso, após essa etapa o número se torna um ponto flutuante de 32 bits e segue sendo operado normalmente. Os detalhes sobre essa operação serão discutidos a seguir, na descrição do módulo responsável por essa função.

O Bloco1 é composto por quatro grandes unidades: BlocoMultiplica1, BlocoVelFat, BlocoMultiplica, BlocoSomadorNormalizador. Esses módulos são descritos a seguir.

Podse-se observar a arquitetura do Bloco1 na figura 31.

5.2.1.1 BlocoMultiplica1

Esse módulo é responsável pela multiplicação do termo A_{ijk} por ‘-1’.

Essa operação exige apenas uma simples mudança de sinal do número. Isso é feito invertendo-se o primeiro bit do número de entrada do módulo, bit esse que corresponde ao sinal.

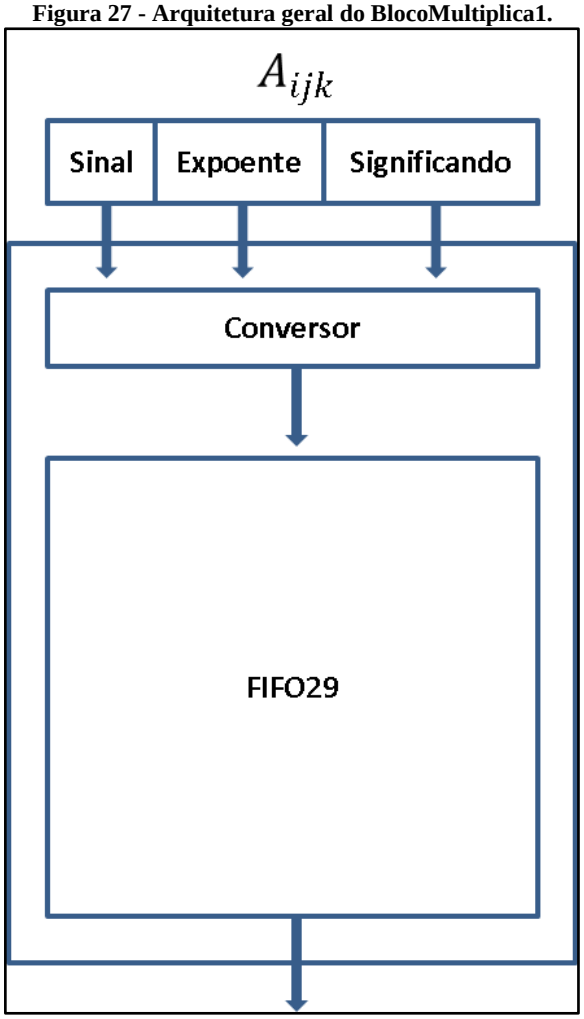
O BlocoMultiplica1 apresenta no início um módulo conversor, já descrito anteriormente. Ele acrescenta o bit implícito no número de entrada, garantindo que o significando esteja na representação prevista no padrão IEEE-754.

O outro componente presente nesse módulo é um registrador de deslocamento (FIFO) que tem a função de propagar o número ao longo dos estágios de *pipeline* para garantir a sincronia dos sinais do módulo.

Esse módulo apresenta 30 estágios de *pipeline* ao todo.

Pode-se observar a estrutura geral do BlocoMultiplica1 na figura 27 a seguir:

Figura 27 - Arquitetura geral do BlocoMultiplica1.



5.2.1.2 BlocoVelFat

Esse bloco é responsável por realizar a operação de multiplicação do termo FAT, pelo quadrado do termo VEL.

Para realizar essas operações os números são inicialmente modificados no módulo Conversor para receberem o bit implícito e possuírem a quantidade de bits esperada para o significando.

O termo VEL segue então para o módulo Quadrado para ser operado e logo em seguido para o módulo Arredondador para ser ajustado.

Como o termo VEL passou pelo arredondamento e retornou a notação de 32 bits, ele necessita passar novamente pelo conversor para ganhar o bit implícito e compor o significando.

Em paralelo as operações do VEL, o FAT é propagado ao longo do *pipeline* por registradores de deslocamento (FIFO) para então receber o bit implícito no módulo conversor.

Em seguida é feita a operação de multiplicação dos termos VEL^2 e FAT. Ao final dessa operação a resposta segue para ser normaliza e arredonda no módulo seguinte.

A resposta dessa operação já normalizada e arredondada é propagada pelo *pipeline* por um registrador de deslocamento (FIFO) de 13 estágios. Isso é feito para garantir a sincronia do sinal com a saída do Bloco2, que também é uma das entradas do Bloco1.

Ao todo esse módulo possui 26 ciclos.

Pode-se observar a estrutura geral do módulo BlocoVelFat na figura 28.

5.2.1.3 BlocoMultiplica

Esse módulo é responsável por realizar a multiplicação da saída do BlocoVelFat com a saída advinda do Bloco2.

Em sua entrada há dois blocos conversores para adicionar os bits implícitos aos dois operandos a serem multiplicados.

Em seguida, os operandos seguem para o bloco Multiplicador para serem multiplicados entre si.

O BlocoMultiplica está presente em outras etapas de processamento do PE, visto que é o responsável pela operação de multiplicação, que é bastante presentes na equação 4.2. Essas etapas serão descritas nos demais blocos a seguir.

Esse módulo possui ao todo 4 estágios de *pipeline*. Pode-se observar sua estrutura geral na figura 29.

Figura 28 - Arquitetura geral do BlocoVelFat.

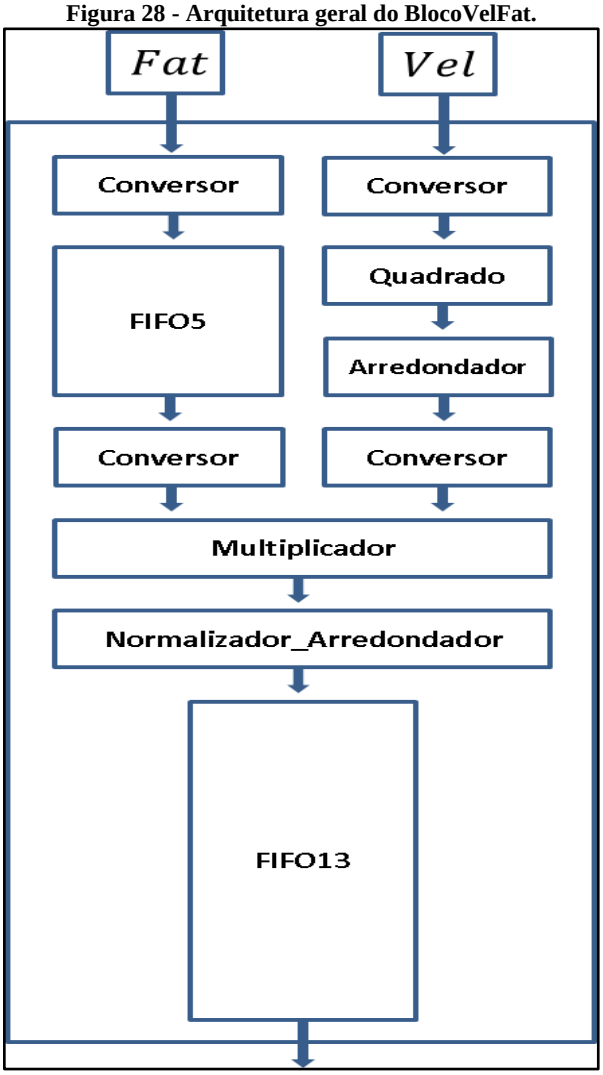
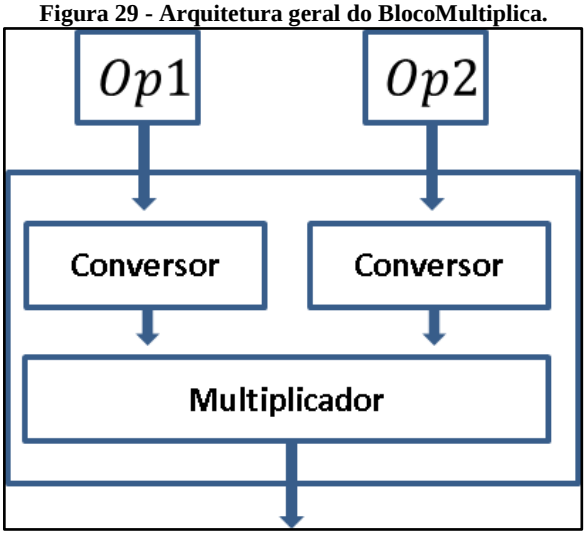


Figura 29 - Arquitetura geral do BlocoMultiplica.



5.2.1.4 BlocoSomadorNormalizador

Esse módulo é responsável realizar uma operação de soma seguida de normalização e arredondamento.

No Bloco1 essa unidade é a encarregada por realizar a soma dos termos $-A_{ijk}$ por $VEL * VEL * FAT * (Bloco2_{out})$.

Inicialmente é feita a operação de soma dos termos de entrada no bloco Somador. Esse módulo já foi descrito na seção 4.1.2. Em seguida a resposta segue para o módulo Normalizador_Arredondador que é a saída do Bloco1, também descrito anteriormente na seção 4.1.4..

O BlocoSomadorNormalizador também é instanciando em outros pontos da arquitetura geral dos PE´s. Isso acontece porque essa unidade é a responsável pelas operações de soma ao longo do *pipeline* do PE e como observado na equação 4.2 a soma é uma operação comum dentro do operador de diferenças finitas.

Ao final de todas as operações a saída desse módulo já está na notação de magnitude e sinal de ponto flutuante de 32 bits definido no padrão IEEE-754.

Ao todo esse bloco possui 8 estágios de *pipeline*. Sendo 4 estágios para o bloco somador e 4 estágios para o bloco Normalizador_Arredondador.

Sua estrutura geral pode ser observada na figura 30 a seguir.

Figura 30 - Arquitetura geral do BlocoSomadorNormalizador.

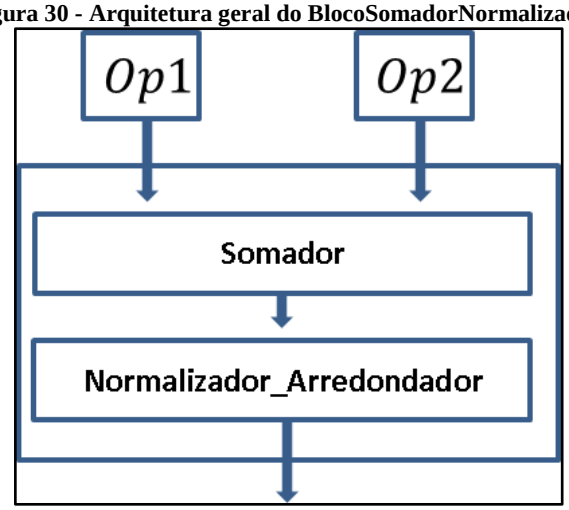
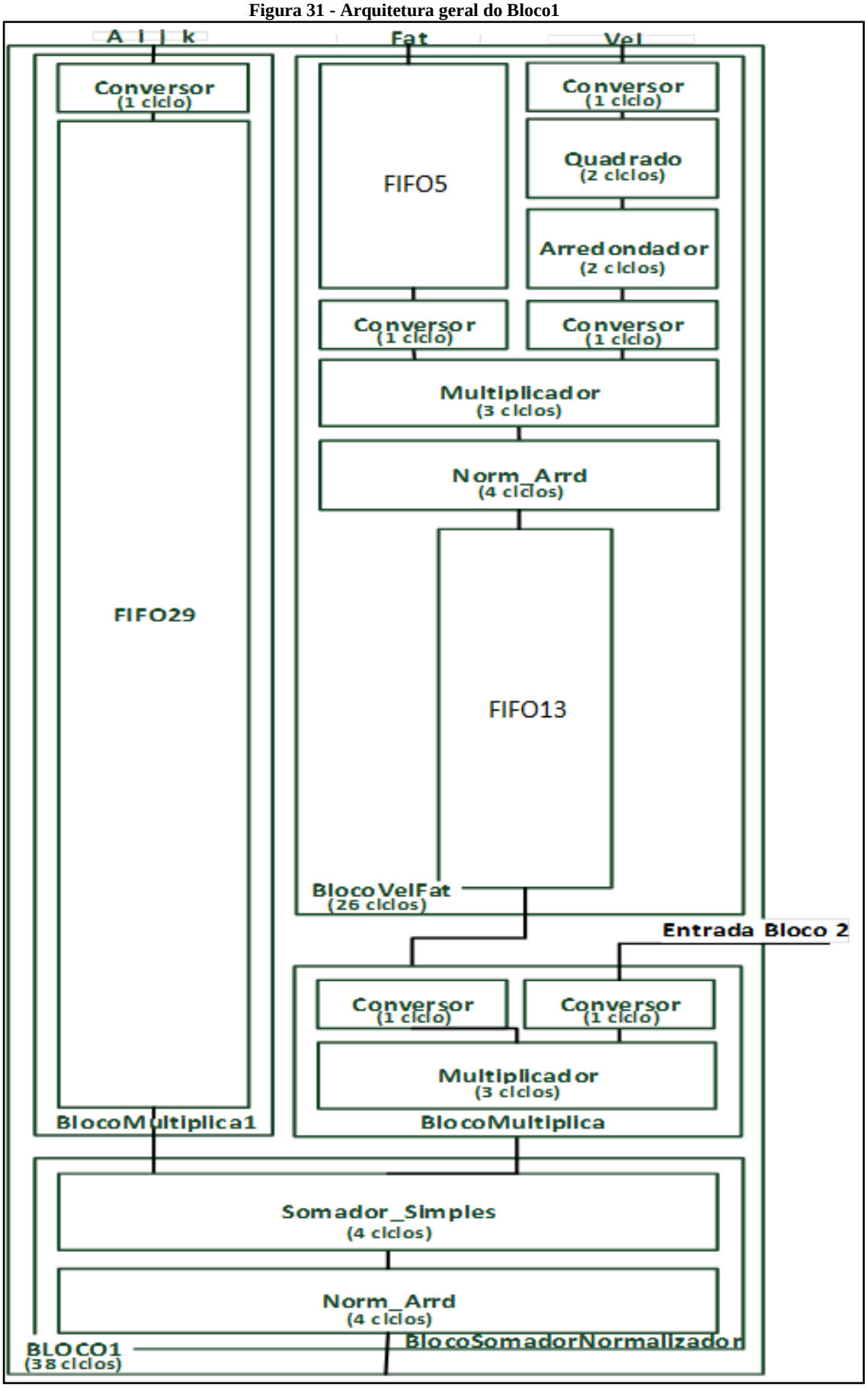


Figura 31 - Arquitetura geral do Bloco1



5.2.2 Bloco2

Este bloco é responsável por realizar o seguinte trecho da equação de diferenças finitas:

$$-1 * (B_{ij-2K} + B_{ij+2K} + B_{i-2jK} + B_{i+2jk} + B_{ijk-2} + B_{ijK+2}) + 16 * (B_{ij-1K} + B_{ij+1K} + B_{i-1jK} + B_{i+1jk} + B_{ijk-1} + B_{ijK+1}) + (-90 * B_{ijk}) \quad (4.4)$$

Este módulo, diferentemente dos outros dois grandes módulos, recebe como entrada, apenas números advindos de uma região externa da arquitetura. Os números recebidos estão todos no padrão IEEE-754 precisão simples.

Basicamente, esse bloco opera com os números da matriz B do operador de diferenças finitas, descrito na equação 4.2, que representam o campo de pressão atual no algoritmo de *RTM*.

Dentre essas entradas, estão o ponto central B_{ijk}, e os pontos vizinhos deste, com distância de um ($B_{ij-1K} + B_{ij+1K} + B_{i-1jK} + B_{i+1jk} + B_{ijk-1} + B_{ijK+1}$) e com distância de dois ($B_{ij-2K} + B_{ij+2K} + B_{i-2jK} + B_{i+2jk} + B_{ijk-2} + B_{ijK+2}$). Os números são multiplicados pelos valores -90, 16 e -1, respectivamente de acordo com o operador de diferenças finitas utilizado no algoritmo de *RTM* (SANTOS, 2012).

Os valores multiplicados decaem conforme se distanciam do valor central B_{ijk}. Com isso, a equação se torna uma soma ponderada, onde os valores a serem multiplicados decaem em módulo conforme se afastam do valor central B_{ijk}.

O Bloco2 é composto dos seguintes componentes: BlocoSomadorBijk, BlocoSomadorBijk2, BlocoMultiplica90, BlocoSomadorNormalizador.

Pode-se observar sua arquitetura geral na figura 35.

5.2.2.1 BlocoSomadorBijk

Esse bloco é o responsável por realizar a soma de seis números recebidos na entrada entre si.

No Bloco2 esse módulo é instanciado duas vezes. Uma vez para se realizar a soma dos termos $B_{ij-1K} + B_{ij+1K} + B_{i-1jK} + B_{i+1jk} + B_{ijk-1} + B_{ijK+1}$ e um outra vez para se

realizar a soma dos termos $B_{ij-2K} + B_{ij+2K} + B_{i-2jK} + B_{i+2jk} + B_{ijk-2} + B_{ijK+2}$ da matriz B.

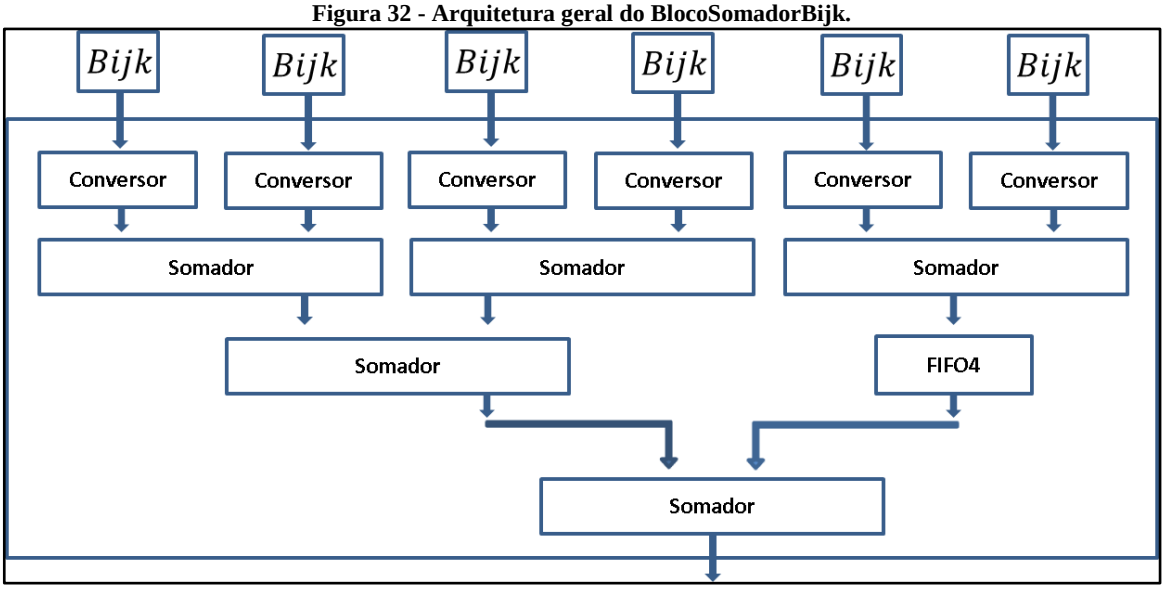
Esse bloco é composto das seguintes unidades: Conversor, Somador e registradores de deslocamento (FIFOs).

Inicialmente é adicionado os bits implícitos dos números a serem operados no bloco Conversor. Em seguida é feita a soma dois a dois dos termos, sendo realizada três somas dois a dois entre os seis termos, em seguida uma soma com o resultado da duas primeiras e por fim uma soma com esse resultado e o dos dois últimos termos de entrada.

Com isso ao final da operação do módulo têm-se os seis números iniciais somados.

O BlocoSomadorBijk possui ao todo 13 estágios de *pipeline* e pode se observador na figura 32 a seguir.

Figura 32 - Arquitetura geral do BlocoSomadorBijk.



5.2.2.2 BlocoSomadorBijk2

Esse módulo realiza a multiplicação das entradas por -1 e por 16 e em seguida realiza uma operação de soma entre os resultados dessas operações.

No Bloco2 esse módulo é responsável por realizar a multiplicação por -1 com resultado da operação de soma dos termos $B_{ij-1K} + B_{ij+1K} + B_{i-1jK} + B_{i+1jk} + B_{ijk-1} + B_{ijK+1}$, assim como a multiplicação por 16 com o resultado da operação de soma dos termos $B_{ij-2K} + B_{ij+2K} + B_{i-2jK} + B_{i+2jk} + B_{ijk-2} + B_{ijK+2}$, ambos advindos do BlocoSomadorBijk.

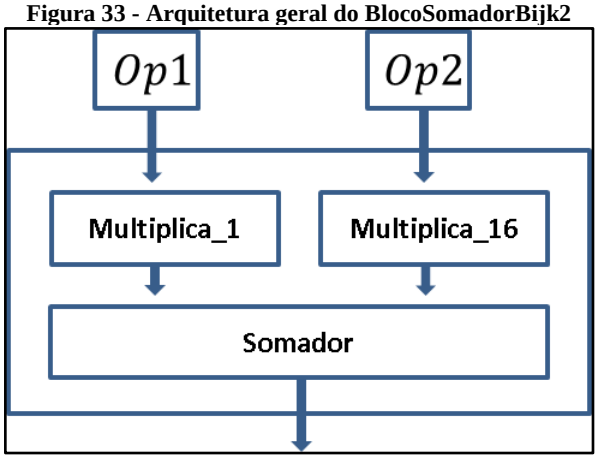
Inicialmente o bloco denominado Multiplica_1 realiza a multiplicação de sua entrada por -1. Para tanto, faz apenas uma inversão do sinal de entrada, não realizando de fato uma operação de multiplicação sobre o número.

Em paralelo a multiplicação por -1 é feita a multiplicação do outro operando de entrada por 16 no módulo Multiplica_16. Para realizar essa operação esse módulo realiza apenas a soma do expoente do número de entrada com 4. Essa operação equivale a quatro operações de deslocamentos para a esquerda (*shift left*), que por sua equivalem a operação de multiplicação de um número de base binária por 16. As exceções são verificadas na entrada desse módulo para que não sejam tratadas como números normais.

Em seguida, é feita uma operação de soma entre as saídas dos dois módulos de multiplicação descritos no parágrafo anterior. Ao final dessa operação o número resultante segue para a saída do BlocoSomadorBijk2.

Esse bloco possui 5 estágios de *pipeline* e sua arquitetura geral pode ser observada na figura 33 a seguir.

Figura 33 - Arquitetura geral do BlocoSomadorBijk2



5.2.2.3 BlocoMultiplica90

Esse módulo realiza a multiplicação do número de entrada por -90. No Bloco2 esse módulo recebe o termo B_{ijk}

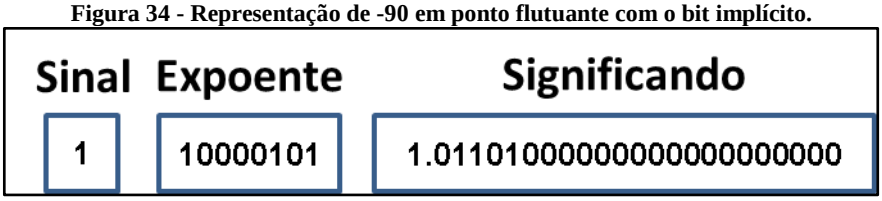
Inicialmente o número de entrada é propagado por 16 estágios de *pipeline* através de registradores de deslocamento (FIFOs). Isso é feito para se garantir a devida sincronia das operações entre os outros módulos do PE.

O número segue então para o módulo Multiplica_90 para ser operado. Esse é o módulo que vai multiplica-lo por -90.

O primeiro estágio do Multiplica_90 faz uma verificação de exceções. Essa etapa é importante para garantir que números tidos como exceções pelo padrão IEEE não sejam operados como números comuns. As exceções são então detectadas e devidamente sinalizadas.

O segundo estágio do Multiplica_90 realiza a operação aritmética esperada para o módulo. Para isso realiza-se a soma do expoente do número menos -90 com o expoente do termo de entrada do módulo e realiza-se a multiplicação do significando de -90 também pelo significando do termo de entrada. Pode-se observar a representação de -90 na notação de ponto flutuante, já com o bit implícito, na figura 34 a seguir.

Figura 34 - Representação de -90 em ponto flutuante com o bit implícito.



É importante destacar que a soma dos expoentes feita no módulo não ocorre propriamente com o expoente do número -90, que equivale a 133 em notação decimal. Isso não pode ocorrer pois o expoente da figura 34 está com o *bias* já adicionado, o que geraria a necessidade da subtração de um *bias*, visto que o expoente do número de entrada também já possui o *bias* embutido. Com isso, o termo correto a ser adicionado ao expoente é 133 menos 127 (*bias*), o que equivale a 6.

A multiplicação entre os significandos ocorre em paralelo a operação com os expoentes.

Ao final do segundo estágio do Multiplica_90 já se têm a resposta do BlocoMultiplica90.

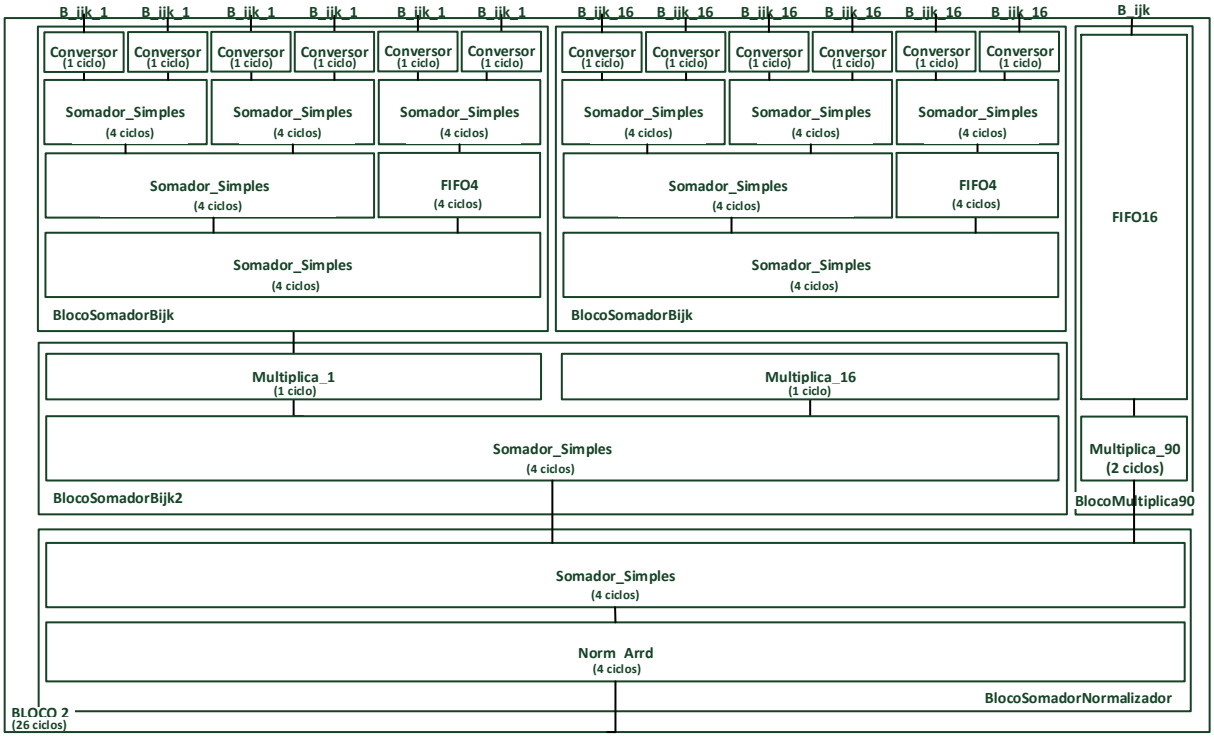
O BlocoMultiplica90 possui ao todo 18 estágios de *pipeline*.

5.2.2.4 BlocoSomadorNormalizador

O Bloco2 possui internamente um BlocoSomadorNormalizador para a soma seguida de normalização dos resultados advindos dos BlocoSomadorBijk2 e BlocoMultiplica90.

Esse bloco já foi descrito na seção 4.2.1, portanto não será detalhado novamente nesta seção.

Figura 35 - Arquitetura geral do Bloco2 no PE Float



5.2.3 Bloco3

As operações realizadas no Bloco3 correspondem ao seguinte trecho da equação de diferenças finitas:

$$(2.0 * B_{ijk} * Abs_i * Abs_j * Abs_k) + ((Abs_i * Abs_j * Abs_k)^2 * (Bloco1)) \qquad (4.5)$$

Esse bloco realiza a multiplicação entre os três valores de Abs, B_{ijk} e dois. Além disso, Além disso, realiza a operação de quadrado do resultado da multiplicação dos termos Abs.

Nesse modulo, também é feita a multiplicação do resultado da operação de quadrado com os valores de Abs e o resultado advindo do Bloco1. Nessa etapa de operação da equação de diferenças finitas, todos os seus números de entrada estão no padrão de ponto flutuante de trinta e dois bits do padrão IEEE.

O Bloco3 é composto pelos seguintes blocos: BlocoMultiplica2, BlocoMultiplicaAbs, BlocoMultiplica2Abs, BlocoQuadradoAbs, BlocoMultiplicaAbs, BlocoMultiplica.

Pode-se observar a arquitetura geral do Bloco3 na figura 41 a seguir.

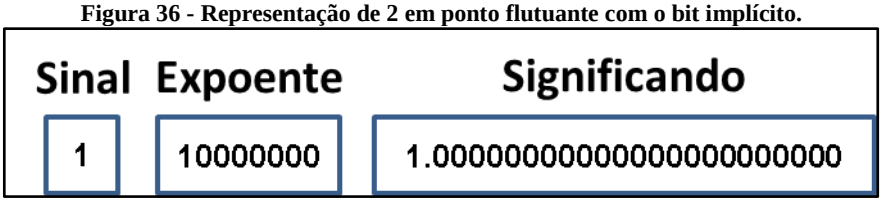
5.2.3.1 BlocoMultiplica2

Esse bloco é responsável por realizar a multiplicação de sua entrada por dois. No Bloco3 ele recebe o termo B_{ijk}como entrada.

Inicialmente é feita a multiplicação do operando de entrada por dois no bloco Multiplica_2. Para tanto é feita uma verificação das exceções de entrada no primeiro ciclo deste módulo, caso o operando de entrada não seja caracterizado como exceção é feita sua multiplicação. Essa operação também é feita no primeiro ciclo desse módulo.

A representação do número dois em notação de ponto flutuante já com o bit implícito pode ser observada na figura 36 a seguir.

Figura 36 - Representação de 2 em ponto flutuante com o bit implícito.



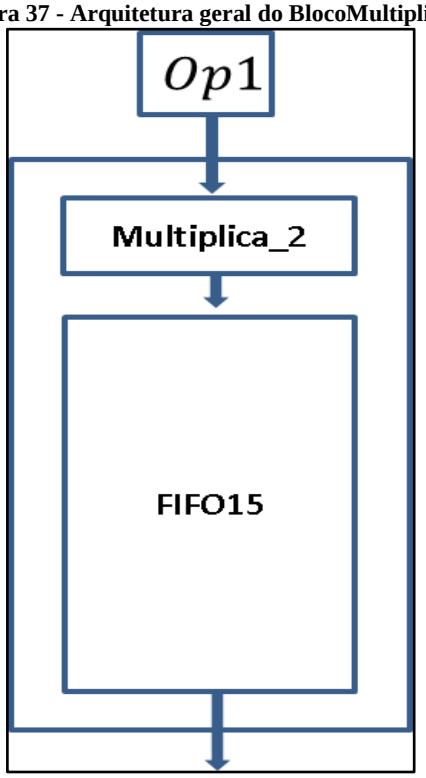
A multiplicação de um número por dois equivale a soma do expoente do operando de entrada pelo expoente do termo dois em ponto flutuante. Essa soma deve desconsiderar o *bias* presente no expoente na representação do número dois em ponto flutuante. Com isso tem-se o expoente 128 menos o 127 (bias), que equivale e 1.

A operação de multiplicação a ser realizada no módulo se torna uma soma de uma constante no expoente, o que economiza os recursos disponíveis em *FPGA* caso uma multiplicação padrão fosse feita.

A resposta do Multiplica_2 segue para registradores de deslocamento (FIFOs) que garante a sincronia desse sinal com os demais sinais presentes ao longo do *pipeline* do PE. A FIFO no BlocoMultiplica2 possui 15 estágios de *pipeline*.

Ao todo, o BlocoMultiplica2 possui 16 estágios de *pipeline*. Pode-se observar a arquitetura desse bloco na figura 37 a seguir.

Figura 37 - Arquitetura geral do BlocoMultiplica2.



5.2.3.2 BlocoMultiplicaAbs

Esse módulo realiza a multiplicação de três operandos recebidos como entrada entre si. No Bloco3 ele realiza a multiplicação dos termos Abs_i, Abs_j e Abs_k entre si.

Inicialmente é feita a conversão dos termos Abs_j e Abs_k para a notação com o bit implícito. Em seguida seus resultados seguem para o bloco Multiplicador para serem operados. O resultado segue então para o bloco Normalizador_Arredondador.

Em paralelo a multiplicação dos termos Abs_j e Abs_k o termo Abs_i é propagado ao longo de um registrador de deslocamento (FIFO) de 8 estágios de *pipeline*. Essa FIFO garante a sincronia entre os sinais de resposta da multiplicação de Abs_j e Abs_k com o termo Abs_i. Ao sair da FIFO é feita a conversão desse termo para a notação com o bit implícito.

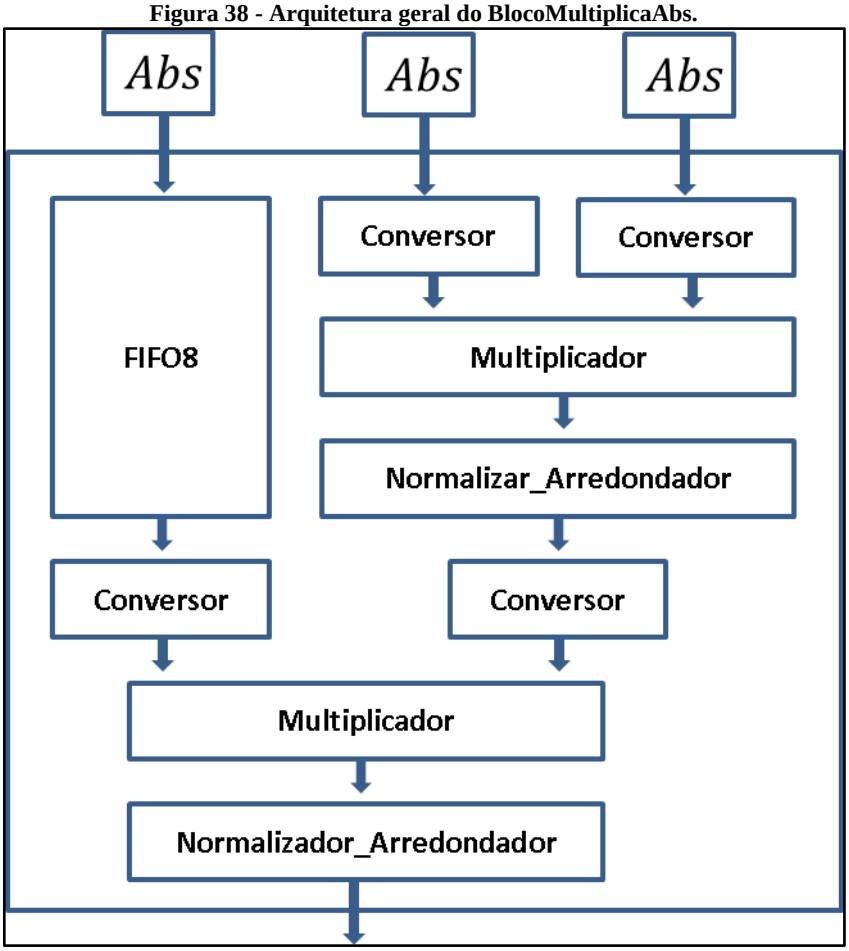
Seguindo o fluxo do *pipeline* desse modulo tem-se então a multiplicação de Abs_i com o resultado da multiplicação das outras duas entradas desse módulo. Assim como descrito anteriormente, essa operação é feita no módulo Multiplicador e seu resultado segue para o módulo Normalizador_Arredondador.

Após a etapa de normalização e arredondamento o número sai do BlocoMultiplicaAbs para entrar no seguinte e seguir o *pipeline* do PE.

Esse bloco possui ao todo 16 ciclos de *pipeline*.

Pode-se observar a arquitetura geral do BlocoMultiplicaAbs na figura 38 a seguir.

Figura 38 - Arquitetura geral do BlocoMultiplicaAbs.



5.2.3.3 BlocoMultiplica2Abs

Esse bloco é responsável por realizar a multiplicação dos dois operandos recebidos como entrada.

No Bloco3 esse módulo é responsável por realizar a multiplicação dos termos advindos do BlocoMultiplica2 e do BlocoMultiplicaAbs.

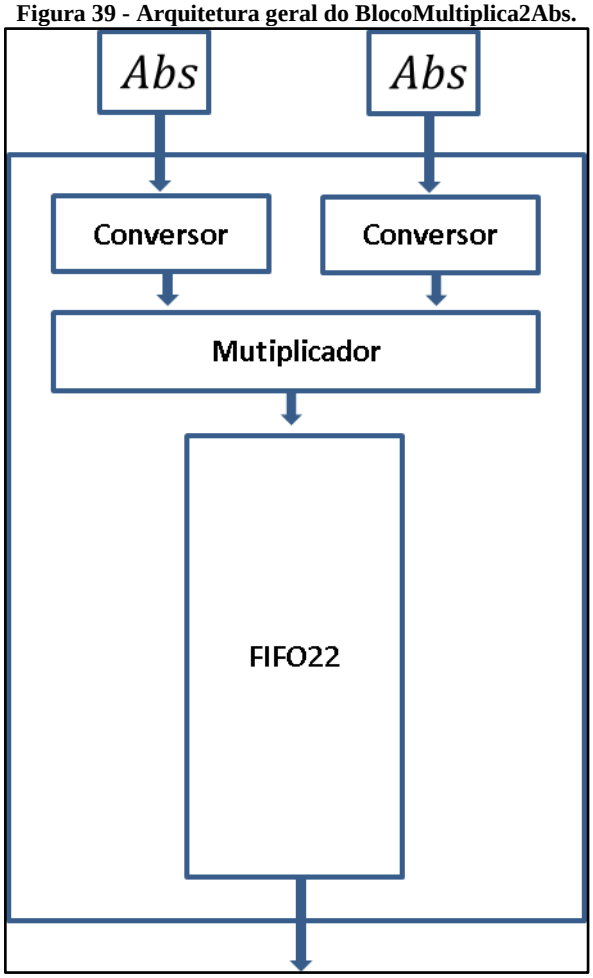
Os dois termos de entrada são convertidos para a notação com o bit implícito logo no primeiro estágio desse bloco. Em seguida, os termos seguem para serem operados no bloco Multiplicador.

A saída do bloco Multiplicador é propagada ao longo de uma cadeia de registradores de deslocamento (FIFO) por 22 estágios de *pipeline*. Isso garante a sincronia da saída do Multiplicador com o restante dos módulos do PE.

Ao todo esse módulo possui 26 estágios de *pipeline.*

Pode-se observar a arquitetura geral do BlocoMultiplica2Abs na figura 39 a seguir.

Figura 39 - Arquitetura geral do BlocoMultiplica2Abs.



5.2.3.4 BlocoQuadradoAbs

Nesse módulo é feita a operação de quadrado com o operando recebido como entrada. No Bloco1 a entrada desse bloco corresponde a saída advinda do BlocoMultiplicaAbs.

Inicialmente é feita a propagação do operando de entrada ao longo de registradores de deslocamento (FIFOs) para devida sincronia dos sinais dentro do PE. Logo em seguida é feita a conversão do número recebido como entrada para a notação interna com bit implícito no módulo Conversor.

O número já notação com bit implícito segue para o módulo Quadrado para ser operado. O bloco Quadrado foi descrito na seção 4.1.5 e não será novamente descrito nesta seção.

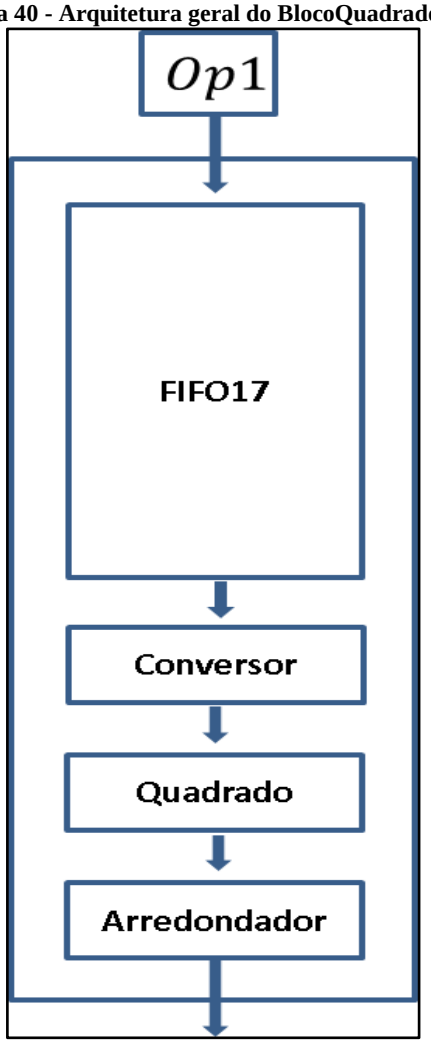
A saída do bloco Quadrado segue para o bloco Arredondador onde terá seu número de bits reduzido de acordo com a operação de arredondamento para o próximo ou par recomendada pelo IEEE. O módulo Arredondador foi descrito na seção 4.1.6.

A saída do Arredondador já corresponde a saída do BlocoQuadradoAbs e se encontra na notação de ponto flutuante de 32 bits.

Ao todo esse módulo possui 22 ciclos de *pipeline*.

Pode-se observa a arquitetura geral do BlocoQuadradoAbs na figura 40 a seguir.

Figura 40 - Arquitetura geral do BlocoQuadradoAbs.

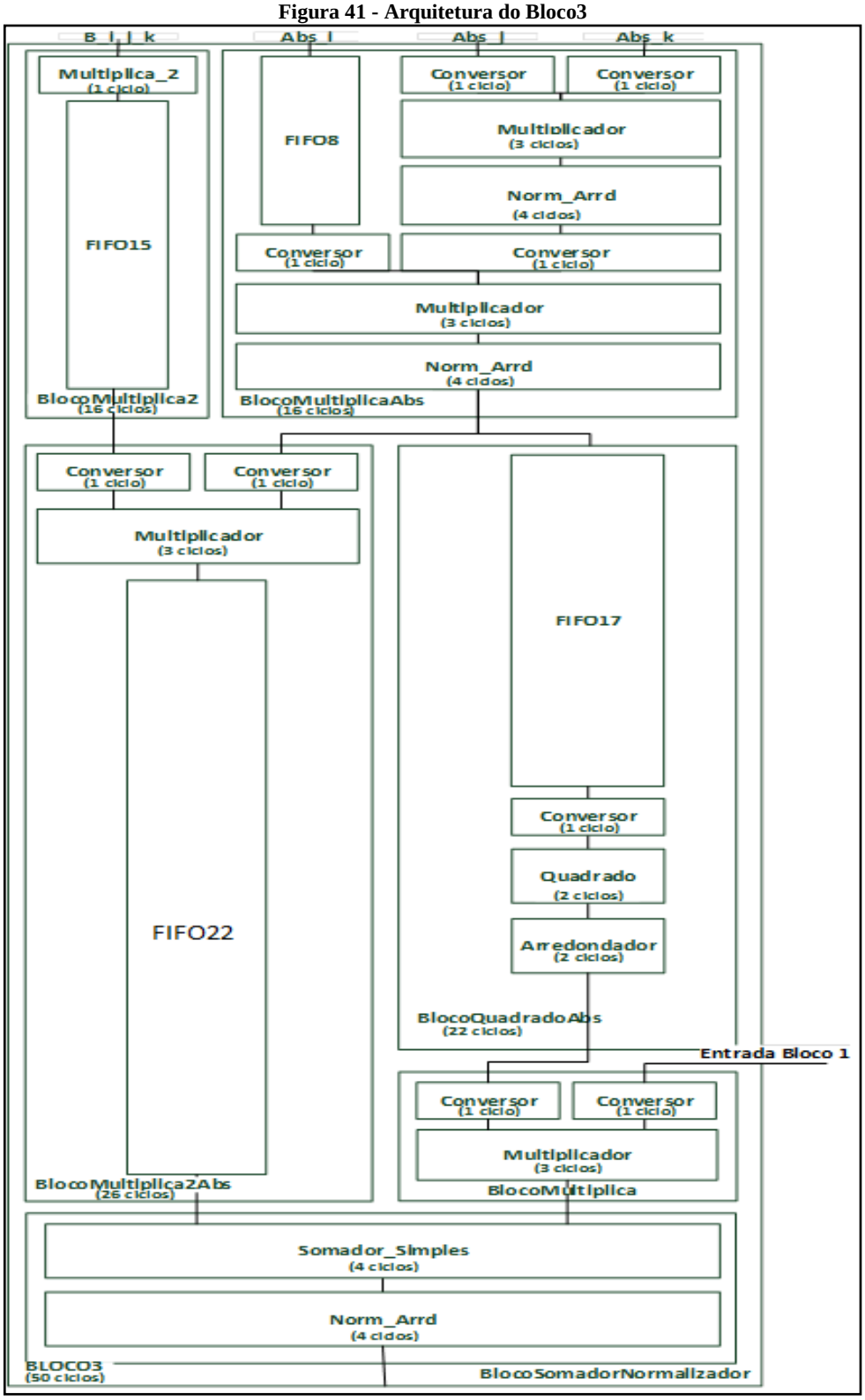


5.2.3.5 BlocoMultiplica

Esse bloco, como descrito na seção 4.2.1.3 deste documento, realiza a multiplicação entre dois números de entrada.

No Bloco3 ele recebe como entrada a saída do BlocoQuadradoAbs e a saída advinda do Bloco1. A saída do Multiplicador presente nesse bloco corresponde já a saída do BlocoMultiplica.

Figura 41 - Arquitetura do Bloco3



5.2.3.6 BlocoSomadorNormalizador

Esse é o módulo que realiza a soma dos resultados advindos do BlocoMultiplica2Abs e do Bloco Multiplica e em seguida faz uma normalização e arredondamento de seu resultado.

Esse bloco já foi descrito na seção portanto não será descrito novamente nesta seção.

A saída do BlocoSomadorNormalizador presente no Bloco3 corresponde a saída final do PE Float, ou seja, corresponde ao resultado do operador de diferenças finitas.

A saída do Bloco3 se encontra na notação de ponto flutuante de 32 bits padrão IEEE-754, mantendo assim a congruência com a representação dos números de entrada do PE Float.

5.3 PE Híbrido

O PE Híbrido possui, basicamente, a mesma estrutura do PE Float apresentado anteriormente. Pode-se observar sua estrutura geral na figura 51 no anexo I deste documento.

Esse *PE* se encontra dividido nos mesmos três módulos que o PE Float está dividido. Essa divisão se deu da mesma maneira que o PE Float, e foi feita de acordo com a ordem das operações e simplificação da equação utilizada.

Portanto, o PE Híbrido possui as mesmas funcionalidades do PE Float e realiza a mesma operação, gerando os mesmos resultados matemáticos, observada a precisão esperada.

O PE Híbrido surgiu da necessidade de redução da área ocupada na *FPGA* pelo PE Float. Isso foi feito para que fosse possível se instanciar mais núcleos aritméticos na *FPGA* e, portanto, aumentar a quantidade de dados possíveis de serem operados.

Essa arquitetura faz uso de uma conversão entre a representação em ponto flutuante para a representação em ponto fixo. Apesar da perda de precisão inerente a essa conversão, a margem de erro obtida não gerou impacto significativo na arquitetura[16]. Por outro lado, o ganho de área obtido foi significativo.

O PE Híbrido ocupa uma menor área da *FPGA*, pois as operações de soma, em ponto fixo, ocorrem de maneira bem mais simples do que em ponto flutuante. Isso implica que menos módulos são necessários para se realizar a mesma operação e, consequentemente, menos hardware é ocupado.

O módulo que realiza uma grande sequência de somas e multiplicações por constantes é o Bloco2. Nesse bloco os números em ponto flutuante são convertidos para ponto

[16] Os resultados obtidos operam dentro da precisão da ordem de 10^{-6} exigida para o modelo do algoritmo RTM pelo Centro de pesquisa da Petrobrás (CENPES) no projeto HPCIN, do Centro de Informática da UFPE

fixo, são operados, e ao final são convertidos para a notação de ponto flutuante novamente. Como a saída desse bloco é um número em ponto flutuante, ela se encontra equivalente à saída do Bloco2 do PE Float. Desse momento em diante no fluxo de execução, os PE´s desenvolvidos são iguais.

O Bloco1 e Bloco3 foram explicados na seção anterior, portanto não serão explicados novamente nessa seção, apesar de também fazerem parte do PE Híbrido. Com isso, a seguir, encontra-se apenas a explicação do Bloco2.

5.3.1 Bloco2

Como dito anteriormente, este bloco é responsável por realizar o trecho da equação de diferenças finitas presente na equação 4.4 deste capítulo

Esse bloco recebe, como entrada, números que são advindos apenas de regiões externas ao PE e estão na notação padrão do IEEE com trinta e dois bits de representação. No entanto, no PE Híbrido, esses números são convertidos em uma notação de ponto fixo antes de sua operação. Isso é feito para economizar a área ocupada em *FPGA* pelo *PE*. Após essa etapa de conversão na entrada, os números podem ser operados normalmente. Ao final de todas as operações desse bloco, os números devem ser convertidos novamente para notação de ponto flutuante para seguirem com as demais operações.

Pode-se notar através da análise desse trecho da equação, que a maior parte das operações realizadas nesse módulo são de soma. Apesar de haver três multiplicações, essas são feitas com números que são constantes, ou seja, não são entradas do módulo e, portanto, podem ser otimizados na operação. Com isso, o Bloco2 é também denominado de árvore de somas.

O Bloco2 no PE Híbrido é composto por cinco grandes blocos: M1Soma, M16Soma, M90, MSoma1_16_90, Conversor_Fixo_Float, Conversor_Float_Fixo.

A seguir são detalhadas suas arquiteturas.

5.3.1.1 Conversor_Float_Fixo

Essa unidade é responsável pela conversão dos dados recebidos em formato de ponto flutuante de 32 bits IEEE-754 para a notação de ponto fixo utilizada no PE Híbrido. A notação padrão adotada para a representação do número em ponto fixo é de 9 bits para parte inteira e 27 para a parte fracionária, como debatido do capítulo 2 desta dissertação.

O primeiro ciclo de *pipeline* desse módulo apenas verifica as exceções previstas no padrão IEEE e as sinaliza para as demais operações.

O segundo ciclo desse módulo faz uma operação sobre o expoente do número de entrada. Para tanto, é necessário se subtrair o *bias* presente no expoente do número em ponto flutuante. Como dito no capítulo 2, o bias nesse caso possui valor de 127.

O valor obtido após a subtração do *bias* corresponde a medida de deslocamento para o valor da mantissa ainda em *float*.

Números menores do que 2 em ponto flutuante possuem o seu expoente igual ou menor ao valor do *bias*. Com isso, a subtração do expoente com o *bias* nesses casos gera zero ou um número negativo. Já os números maiores do que 2, possuem o expoente maior do que o bias, portanto a subtração por 127 gerará sempre um número positivo.

Essa condição é tratada de maneira que a subtração pelo bias sempre gera um resultado positivo. Isso é feito apenas realizando-se uma comparação do expoente com 127. Caso seja maior, é feita a subtração e deve-se realizar um deslocamento para a direita com o significando pelo números de vezes obtido com a subtração. Caso seja menor, é feita a subtração e deve-se realizar um deslocamento para a esquerda com o significando pelo números de vezes obtido com a subtração. Caso seja igual, não é necessário se realizar deslocamentos no significando.

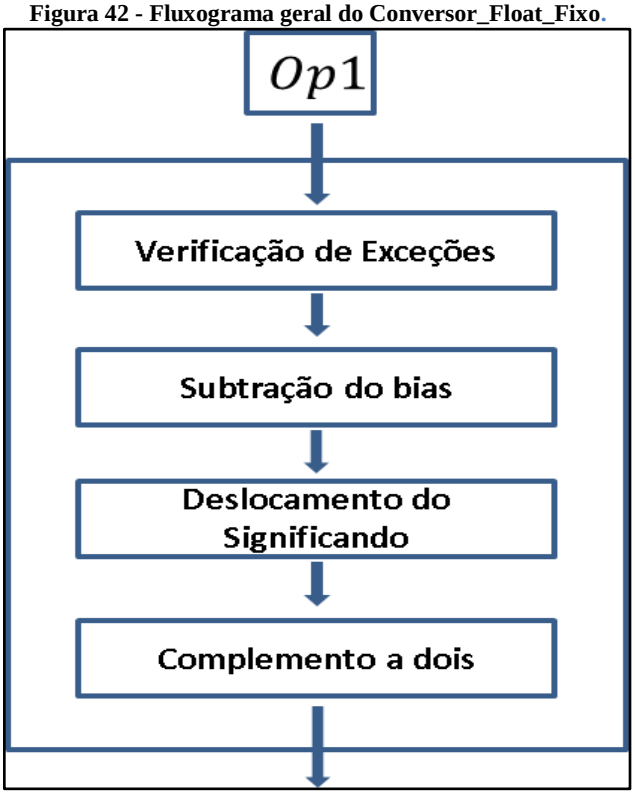
O terceiro ciclo desse módulo recebe os valores e a direção dos deslocamentos a serem realizados no significando do número e apenas realiza tal operação.

O quarto ciclo, faz uma operação de complemento a dois do número, invertendo todos os bits do significando e somando-se 1.

Ao final dessa operação têm-se o número em representação de ponto fixo e em notação de complemento a dois já preparado para se realizar as operações aritméticas no PE Híbrido.

Pode-se observar o fluxograma geral do Conversor_Float_Fixo na figura 42 a seguir. Esse módulo possui ao todo quatro estágios de *pipeline*.

Figura 42 - Fluxograma geral do Conversor_Float_Fixo.



5.3.1.2 M1Soma

Esse bloco é responsável por realizar o seguinte trecho da equação do operador de diferenças finitas na notação de ponto fixo:

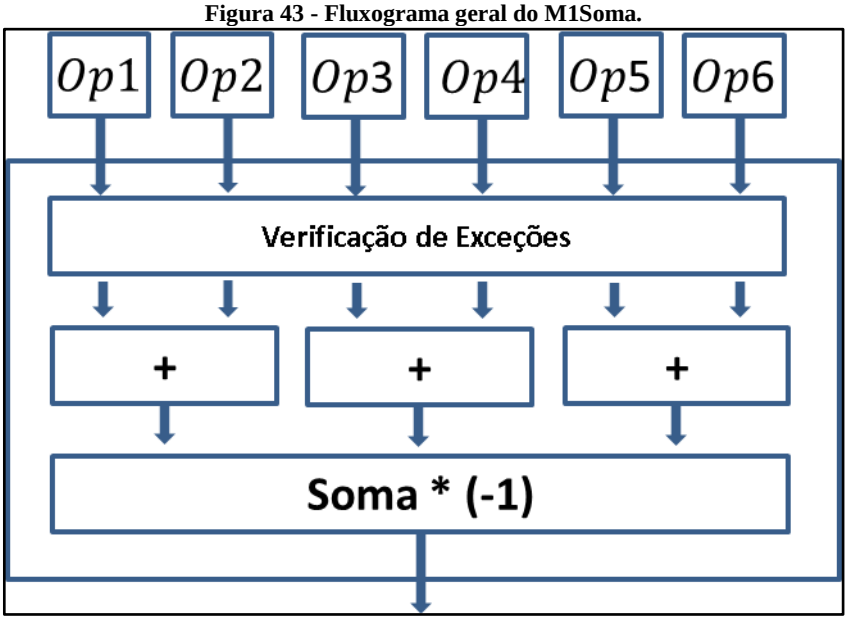
$$-1 * (B_{ij-2K} + B_{ij+2K} + B_{i-2jK} + B_{i+2jk} + B_{ijk-2} + B_{ijK+2}) \qquad (4.6)$$

Para se realizar essa operação, as somas foram separadas em somas de dois termos, apenas. Com isso, foram somadas as duplas B_{i-2jK} e B_{i+2jk}, B_{ij-2K} e B_{ij+2K}, e também a dupla B_{ijk-2} e B_{ijK+2}. Essas três somas foram feitas em paralelo em apenas um ciclo de relógio.

Após a primeira rodada de somas, é feita a soma de seus resultados entre si. Essa operação é feita no terceiro e último ciclo desse módulo.

Também no último ciclo desse módulo é feita uma operação equivalente a multiplicação por -1 ao resultado da soma final. Essa operação equivale a um complemento a dois, uma operação de inversão dos bits e em seguida a soma de 1.

Figura 43 - Fluxograma geral do M1Soma.



5.3.1.3 M16Soma

É o componente do Bloco2 que é responsável por realizar o seguinte trecho da equação do operador de diferenças finitas, em ponto fixo:

$$16 * (B_{ij-1K} + B_{ij+1K} + B_{i-1jK} + B_{i+1jk} + B_{ijk-1} + B_{ijK+1}) \qquad (4.7)$$

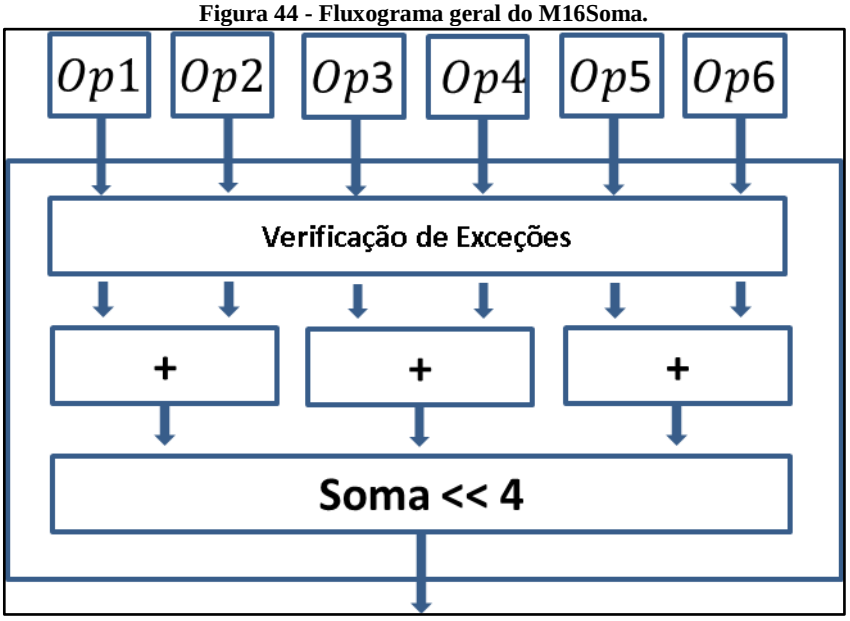
Assim como o bloco *M1Soma* descrito anteriormente, esse trecho de equação foi separado em três somas de dois elementos. Com isso, foram somadas as duplas B_{i-1jK} e B_{i+1jk}, B_{ij-1K} e B_{ij+1K}, e também a dupla B_{ijk-1} e B_{ijK+1}. Essas três operações são feitas em paralelo em um único ciclo.

No ciclo seguinte, seus valores são adicionados e a multiplicação por 16 é realizada. Uma otimização foi feita para se realizar essa multiplicação. Como o número a ser multiplicado é constante e corresponde a 2^4, significa que a operação pode ser substituída por um deslocamento para a esquerda (*shift left)* de quatro posições.

Pode-se observar o fluxograma geral do M16Soma na figura 44 a seguir.

Essa unidade possui ao todo três estágios de *pipeline*.

Figura 44 - Fluxograma geral do M16Soma.



5.3.1.4 M90

É o componente do Bloco2 do PE Híbrido que é responsável por realizar o seguinte trecho da equação do operador de diferenças finitas em ponto fixo:

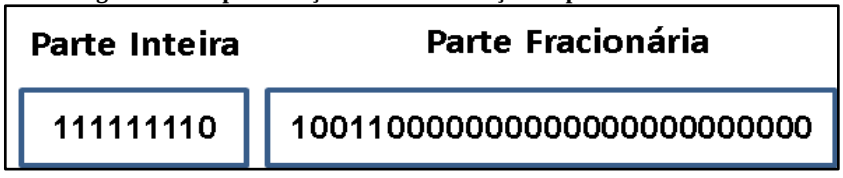
$$-90 * B_{ijk}$$

Inicialmente é feita a identificação de exceções do número recebido. Essa operação é feita no primeiro ciclo de *pipeline* do módulo

Diferentemente dos subcomponentes do Bloco2 descritos anteriormente, essa unidade não possui somas, mas apenas uma multiplicação por uma constante. Pode-se observar a representação de -90 na notação de ponto fixo utilizada na figura 45.

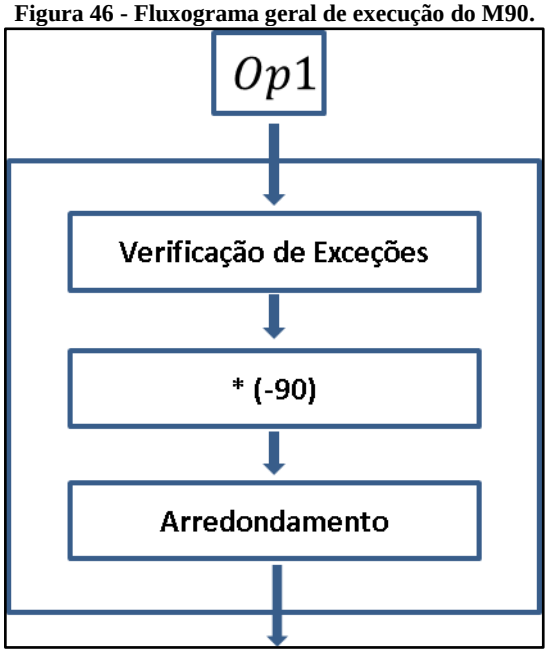
No segundo ciclo, é feita a operação de multiplicação e, no seguinte, uma operação de arredondamento. Por fins de economia de recurso de hardware, optou-se por se realizar um truncamento do resultado, sem acarretar em perdas significativas de precisão.

Figura 45 - Representação de -90 na notação de ponto fixo utilizada.



Pode-se observar o fluxo geral de execução do módulo M90 na figura 46 a seguir. Ao todo, o bloco M90 possui 3 estágios de *pipeline*.

Figura 46 - Fluxograma geral de execução do M90.



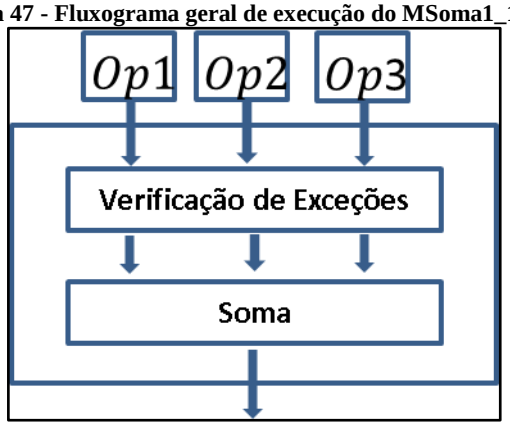
5.3.1.5 MSoma1_16_90

É a unidade integrante do Bloco2 responsável por realizar a soma dos resultados advindos dos blocos: M1Soma, M16Soma, M90, em formato de ponto fixo.

Esse módulo basicamente realiza as duas somas entre essas três entradas. Essa operação é feita em um ciclo de relógio, apenas. Especificamente, essa operação é feita no segundo estágio do *pipeline* do módulo.

O primeiro estágio do *pipeline* desse módulo é responsável pela verificação de exceções. Observa-se o fluxograma de execução desse módulo na figura 47.

Figura 47 - Fluxograma geral de execução do MSoma1_16_90.



5.3.1.6 Conversor_Fixo_Float

Esse componente realiza a conversão de números no formato de ponto fixo para ponto flutuante IEEE-754.

Inicialmente, no primeiro ciclo de *pipeline* é feita a verificação quanto as exceções previstas pelo padrão IEEE.

Em seguida, no segundo ciclo, é feita uma operação de complemento a dois, pois o número em ponto fixo precisa ser representado na notação de magnitude e sinal.

Após essa etapa, é feita uma contagem de bits da esquerda para a direita, até o primeiro bit ‘1’ ser encontrado. Essa contagem serve indica a quantidade de deslocamentos a serem realizados no número, em ponto fixo.

Para a correta representação em ponto flutuante o número deve estar normalizado, ou seja, só deve haver um único bit antes do ponto separador e esse deve ter valor 1.

Caso a quantidade de deslocamentos seja maior do que o tamanho da parte inteira do número em ponto fixo, no caso 9, significa que o número em ponto fixo é menor do que 1 e, portanto, só possui parte fracionária. Portanto, deve ter sua parte inteira e fracionária deslocadas para a esquerda pelo número de vezes equivalente a subtração da contagem do estágio anterior por 9.

Caso a quantidade de deslocamentos seja menor do que 9, significa que o número é maior ou igual a 1 e possui parte inteira diferente de 0. Portanto, deve ter sua parte inteira e fracionária deslocadas para a direita pelo número de vezes equivalente a subtração de 9 pelo resultado da contagem no estágio anterior.

Através dessa quantidade de deslocamento é feita uma lógica para se adicionar ou subtrair um determinado valor ao *bias*, gerando assim o expoente em ponto flutuante do número.

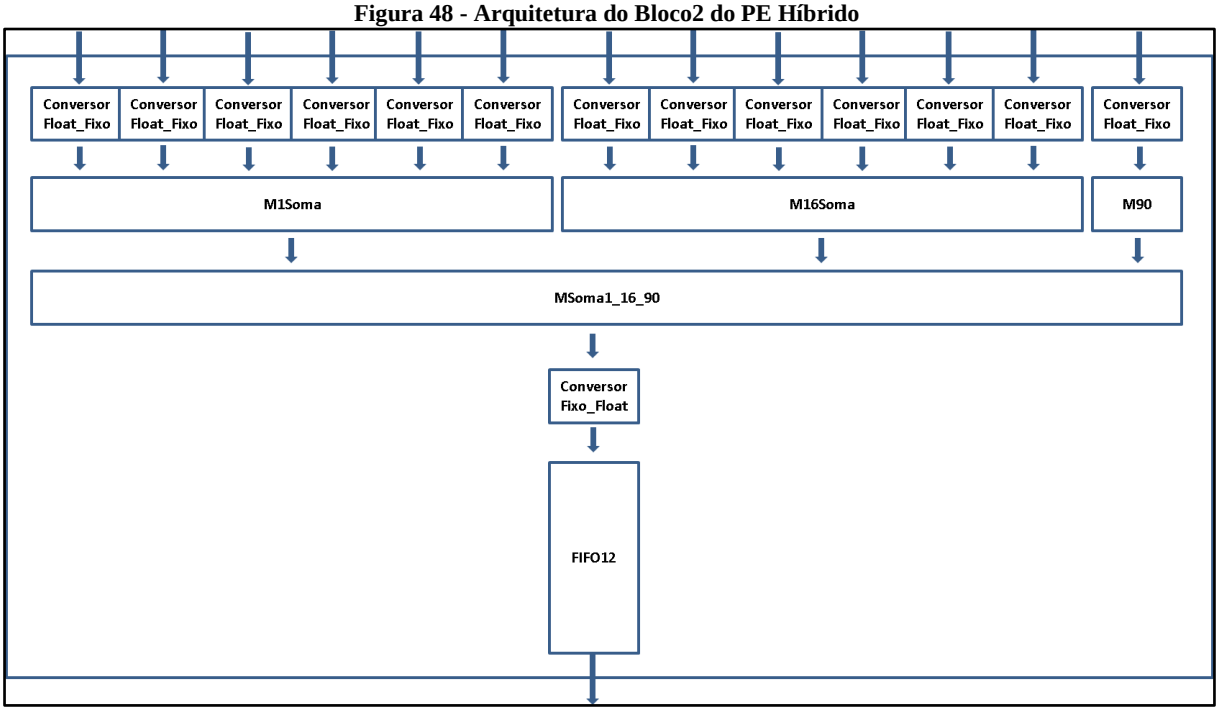
Subtrai-se então do bias (127) o valor encontrado, caso tenha ocorrido um deslocamento para a esquerda ou soma-se o valor encontrado, caso tenha ocorrido um deslocamento para a direita.

Casos especiais devem ser tratados a parte, como é o caso do zero, identificado ao longo das operações dentro do módulo, para que ocorra sua correta representação no padrão IEEE.

Os subcomponentes presentes nesse bloco serão descritos a fundo no decorrer deste capítulo.

Pode-se observar a estrutura do *Bloco* 2 na figura 48 a seguir.

Figura 48 - Arquitetura do Bloco2 do PE Híbrido



5.4 Metodologia de Desenvolvimento

Para o desenvolvimento de um módulo aritmético capaz de calcular o operador de diferenças finitas utilizado no *RTM* em hardware reconfigurável, foi necessária a realização de algumas etapas prévias. Essas etapas de desenvolvimento incremental definiram, também, metodologia de verificação dos módulos desenvolvidos.

Devido à complexidade da equação e dos algoritmos aritméticos em ponto flutuante, foi necessária a criação de versões de soluções da equação em diferentes níveis de abstração de programação. Essa abstração das camadas de hardware, objetivo final deste trabalho, seguiu do alto nível até chegar, finalmente, na versão definitiva em *FPGA*.

A criação de modelos em mais alto nível do módulo se fez necessária por permitir um estudo prévio das estratégias de implementação. Essa etapa envolve tanto a escolha do padrão de representação numérica a ser adotado, quanto a verificação dos efeitos desta escolha sobre o resultado do processamento.

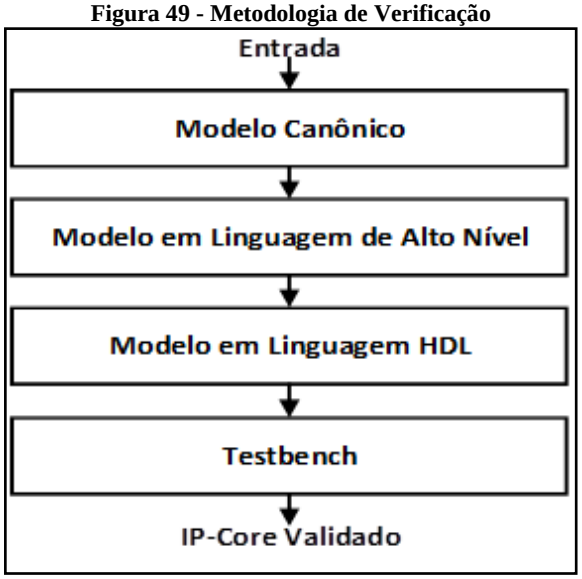
Outro ponto significativo nesta etapa é a geração de modelos de referência confiáveis para a verificação e validação dos módulos aritméticos em desenvolvimento. Em especial quando tais módulos devem operar sobre tipos de dados que não possuem representação direta nas linguagens de descrição de hardware. Com isto, fez-se necessária a adoção de uma metodologia que permita a criação de modelos de referência capazes de mapear os resultados obtidos em software com os resultados obtidos no padrão de representação numérica adotado em hardware.

A metodologia de desenvolvimento obedeceu à seguinte ordem:

1. Geração de um Modelo Canônico.
2. Geração de um Modelo de Alto Nível em Software.
3. Conversão do Modelo em Alto Nível em Software para um Modelo em linguagem de descrição de hardware.
4. *Testbench*.

Pode-se observar na figura 49 o fluxograma do desenvolvimento adotado. A seguir, são explicados cada uma das etapas citadas.

Figura 49 - Metodologia de Verificação



5.4.1 Modelo Canônico

Esse modelo foi desenvolvido com o objetivo de gerar um conjunto de dados de referência que reflete o comportamento esperado do núcleo aritmético em uma situação real de uso.

Para tanto, uma versão em software, implementada em C++, foi desenvolvida. Essa versão opera com entradas geradas aleatoriamente pelo algoritmo que então calcula as saídas. Essas respostas são armazenadas para serem utilizadas em etapas posteriores de verificação.

O gerador de entradas impõe limites ao alcance dos números. Isso é feito para que os números gerados estejam próximos dos que podem ser encontrados em situações de processamento real. A variação média dos números coletados pelo algoritmo de análises sísmicas foi validada pelo Cenpes (2015) em parceria com o projeto HPCIn (2014) da UFPE.

Como exemplo, pode-se citar a variação das velocidades de propagação dos campos de pressão que não excedem a velocidade de 2^{15} km/h nos modelos adotados. Essa característica da velocidade, juntamente com a dos demais parâmetros do operador de diferenças finitas, foi simulada pelo gerador de entradas que alimenta a versão do núcleo aritmético em alto nível.

A versão em software possui etapas de conversão de números inteiros para a representação em magnitude e sinal dos números em ponto flutuante. Isso gera os números já com os componentes de sinal, expoente e mantissa. Esses elementos são operados separadamente, já seguindo o fluxo de operação do padrão IEEE-754 e, ao final, tem seu resultado reagrupado para a versão de saída do formato desejado.

No modelo canônico a equação do operador de diferenças finitas do *RTM* é realizada de maneira mais direta possível, sem o particionamento da equação em blocos. Dessa maneira, os números gerados aleatoriamente apenas alimentam a equação para se conhecer e guardar a saída correspondente do operador para aquele determinado conjunto de dados de entrada.

O objetivo desse modelo também é ser de natureza simples, tanto de implementação, quanto de verificação. Caso contrário, um novo sistema de verificação teria que ter sido implementado para esse módulo.

5.4.2 Modelo de Linguagem de Alto Nível

O objetivo desse modelo é a implementação de uma versão de mais baixo nível do modelo canônico, utilizando a camada de mais alto nível da linguagem de *System Verilog*[17]. Com isso, uma camada intermediária entre a versão em software e a versão em *hardware* foi gerada.

[17] System Verilog. Acesso em: jul. 2015. Disponível em: <http://www.eda.org/sv/SystemVerilog_3.1a.pdf>

Foi nesse modelo em alto nível que se iniciou o particionamento do módulo aritmético em blocos. Esse particionamento foi feito de acordo com as dependências dos números no fluxo de execução do operador de diferenças finitas no *RTM*. Isto é, procurou-se agrupar em um bloco os números que podiam ser melhor operados de maneira paralela. Isso gerou a reorganização da equação do operador de diferenças finitas que levou a divisão do PE em três grandes blocos.

Assim como no modelo canônico, os blocos operam internamente com os números já na notação de sinal, expoente e mantissa. No entanto, diferentemente do modelo canônico que não esmiúça as operações, o modelo em alto nível realiza as operações aritméticas em várias etapas. Essas etapas são as correspondentes aos passos descritos no capítulo dois dessa dissertação para a soma, subtração e multiplicação dos números em ponto flutuante. Com isso os números são operados de maneira semelhante a como são operados diretamente no *hardware*.

Isso permitiu realizar as operações aritméticas com as restrições impostas pelo padrão IEEE-754 com os números no *software*. Devido a isso, foi possível fazer a simulação das etapas das operações aritméticas em alto nível, permitindo a identificação de erros de implementação antes da aplicação direta em *hardware*.

O modelo em alto nível recebe como entrada os números gerados no modelo canônico. Esses números são operados pelos blocos e ao final seus resultados são comparados com os resultados do modelo canônico previamente armazenados. O modelo canônico atua como uma referência para a corretude dos algoritmos do modelo em alto nível. Portanto, os dois modelos devem apresentar as mesmas respostas dadas as mesmas entradas.

Nesta etapa do projeto também foram definidos os padrões de representação numérica adotados, ponto flutuante e ponto fixo, bem como, a precisão numérica dos operandos, dos operadores aritméticos utilizados e dos resultados.

5.4.3 Modelo em Linguagem *HDL*

Nessa fase da metodologia, o modelo obtido na etapa anterior é, então, traduzido da linguagem de alto nível para uma linguagem de descrição de hardware (*HDL*).

Para tanto, é mantida a estrutura de grandes blocos e módulos menores definidos nas etapas anteriores para simplificar o desenvolvimento do projeto.

Nessa fase, o código em linguagem de hardware propriamente dito foi feito. Tendo todos os testes sido feitos para os estudos nas etapas anteriores, essa etapa não apresenta grandes mudanças de fluxo de operações aritméticas.

Esse código foi feito com a parte sintetizável da linguagem *System Verilog*, a camada mais próxima do *hardware* dessa linguagem. Isso exigiu ajustes na codificação feita na etapa de modelagem em alto nível, modificando-se as camadas que não podiam ser implementadas diretamente em *FPGA* para camadas implementáveis.

Durante essa fase, é observada a utilização de termos aritméticos de multiplicação e soma que podem ser inferidos durante o processo de síntese em elementos lógicos de presença limitada na *FPGA*, como os *DSPs*. Isso implica um cuidado durante a programação e a fragmentação de determinados trechos de código para que a ferramenta de síntese espalhe as operações na lógica da *FPGA*, economizando blocos *DSPs*, por exemplo.

O modelo de referência em alto nível serve como modelo de corretude dos resultados para o modelo em linguagem *HDL*. Ao longo do desenvolvimento do modelo de mais baixo nível o resultado do operador de diferenças finitas deve ser o mesmo do modelo de alto nível.

Ao final desta etapa, o módulo em hardware foi desenvolvido e se encontra funcional para seguir para a etapa de testes exaustivos.

5.4.4 *Testbench*

Para garantir a confiabilidade dos módulos desenvolvidos durante o desenvolvimento da arquitetura uma estrutura de testes foi elaborada. Esse modelo de testes foi utilizado dentro do projeto HPCIn (2014) dentro do Centro de Informática da UFPE.

O modelo de testes desenvolvido foi aplicado tanto à arquitetura como um todo, quanto a seus módulos menores.

A arquitetura de testes também foi desenvolvida em linguagem *System Verilo*g, que permite certa abstração de implementação de elementos do hardware, mas, ao mesmo tempo, está próxima o bastante do modelo a ser testado.

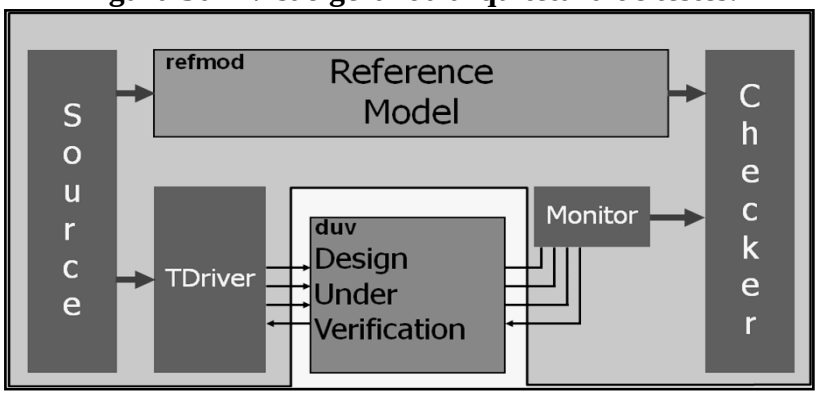
A estrutura do *Testbench* é composta dos seguintes módulos:

- ***Source***: Módulo de entrada do *Testbench*, responsável pela geração dos estímulos de teste. Nele são definidos o alcance numérico dos testes, a frequência de amostras de casos específicos a serem gerados e a duração dos testes.

- ***TDriver***: Esse módulo é responsável pela interface dos números gerados no *Source* com o modelo a ser testado (*DUV*). Ele é capaz de receber os estímulos de entrada e gerar os sinais necessários ao controle do *DUV* no formato adequado para seu funcionamento.
- ***DUV***: Corresponde ao *Device Under Verification* (*DUV*) ou Dispositivo a ser verificado. Nesse módulo estão encapsulados os PE´s desenvolvidos e que precisam ser verificados quanto a sua corretude. Ele recebe os sinais advindos do *TDriver*, já com pulso de relógio (*clock*) adequado, e gera uma saída numérica da mesma maneira que seu funcionamento normal.
- ***Monitor***: Responsável por receber os sinais de saída e fazer a verificação dos resultados gerados pelo *DUV*. Esse módulo também monitora os demais sinais gerados pelo *DUV*, que correspondem a funções de controle, como exceções, por exemplo. Ele também é responsável por realizar a interface com o módulo de checagem de resultados e preparar a saída do *DUV* para ser compatível com a entrada desse módulo de checagem.
- ***Checker***: Esse componente é responsável por verificar a corretude dos resultados gerados pelo *DUV*. Essa corretude engloba aspectos como precisão, variação do alcance numérico, correto tratamento de exceções. Em suma, esse módulo certifica-se de que a saída numérica do *DUV* corresponde a gerada pelo modelo de referência (*RefMod*).
- ***RefMod***: Modelo de referência em linguagem de mais alto nível, do que o dispositivo sob verificação. Esse módulo recebe os estímulos numéricos gerados pelo *Source* e executa o algoritmo a ser desenvolvido pelo *DUV*. Ao fim de sua execução, tem-se um conjunto de dados numéricos de referência. Esses dados servem como entrada para o módulo de checagem que também recebe a saída do *DUV* e realiza a comparação entre eles.

A figura 50 a seguir expõe a visão geral da arquitetura de testes desenvolvida para se verificar a corretude dos resultados dos *PE's*. Nela, pode-se observar todos os módulos que compõem a arquitetura de testes, bem como suas interconexões.

Figura 50 – Visão geral da arquitetura de testes.



5.5 Conclusões

Este capítulo apresentou as arquiteturas essenciais desenvolvidas para este trabalho.

Foi exposta inicialmente uma visão geral da arquitetura com seus módulos mais utilizlados e a estrutura em três grandes blocos. Em seguida, foram expostos os componentes do PE Float. Por fim, o PE Híbrido foi exposto bem como todos os seus módulos menores.

Além de expor os detalhes das arquiteturas dos PE´s, também foi exposta toda a metodologia adotada para seu desenvolvimento, a qual foi a mesma utilizada para todas as suas versões, das mais simples às mais complexas.

Capítulo

6

6. Resultados

Neste capítulo serão discutidos os resultados obtidos com o núcleo de processamento aritmético desenvolvido em *FPGA* bem como os resultados obtidos na plataforma de RTM 3D.

6.1 Resultados de Síntese para uma FPGA Stratix IV da Altera

Nesta seção, estão expostos os resultados obtidos pelos respectivos PE´s implementados para *FPGA*. Esses resultados foram obtidos em nível de síntese pela ferramenta *Quartus II* da *Altera* e fazem parte de seu relatório de síntese.

A tabela 7 a seguir contém os resultados de síntese do PE3D Float utilizados como base para comparação, bem como, os resultados do PE Híbrido. A síntese foi feita para um *Stratix IV EP4SE530H35C2*, na ferramenta *Quartus II* (ALTERA, 2014).

Tabela 7 - Resultado de Síntese do PE Float e PE Híbrido para Stratix IV.

Dispositivo	PE FLOAT	PE Híbrido	Total disponível
ALUTs	12.370	8.611	424.960
Blocos DSP´s	36	40	1.024
Registradores	10.341	8.105	424.960

Analisando-se a tabela acima, pode-se observar que a área ocupada por cada PE em temos de lógica (*ALUTs*) foi bem reduzida em relação à área total disponível em *FPGA*.

Do total de DSP´s disponíveis na Stratix IV apenas uma fração foi utilizada por cada PE. No entanto, os PE´s desenvolvidos consomem relativamente mais DSP´s, cerca de 4% do que ALUT´s, de 2% até 3% aproximadamente. Isso implica que o limitante para uma maior presença de PE´s dentro da *FPGA* são os blocos DSP´s.

Os DSP´s são de suma importância, pois são otimizados para alguns processos aritméticos como multiplicações, melhorando a frequência geral dos módulos aritméticos. No entanto, esse é um recurso mais limitado para alguns *FPGA* do que a lógica, como se observa na tabela 7. Parte da computação que deveria ser realizada nos *DSP´s*, se distribui ao longo da área lógica do *FPGA*, como os *ALUTs*, deixando apenas os segmentos prioritários de computação para os *DSP's*.

Em um caso teórico assumindo-se o pior dos limitantes de recursos, o número de DSP´s, poderia-se instanciar 28 PE´s Float e 25 PE´s Híbridos em uma Stratix IV.

Caso outro modelo de *FPGA*, com número de DSP´s abundante fosse utilizado, o limitante seria a quantidade de ALUT´s consumidas por cada PE. Sob esse ponto de vista seria possível se instanciar 34 PE´s Float e 49 PE´s Híbridos.

Analisando-se esses números pode-se perceber que o PE Híbrido é mais econômico em termos de lógica e mais oneroso em termos de DSP´s. Isso ocorre pois parte de trechos

dos módulos que utilizam recursos de lógica do PE Float passaram a serem feitos em blocos DSP´s.

O PE híbrido foi pensando para se economizar lógica da *FPGA* e portanto atingiu seu objetivo economizando cerca de 30% das ALUT´s.

Implementações do operador de diferenças finitas do *RTM* em *FPGA* cuja a presença de DSP´s é escassa podem fazer uso do PE Float. Sua economia em termos de DSP´s comparados ao PE Híbrido foi de cerca de 10%.

A etapa de redução de área com o PE Hìbrido gerou, portanto, resultados satisfatórios sem afetar significativamente na precisão esperada.

Fazendo a análise de *timing* com a ferramenta *TimeQuest Timing Analyser* do *QuartusII* (Altera, 2015), as frequências de operação obtidas, a nível de síntese, foram 208.78 MHz para o PE Float e 198.33 MHz para o PE Híbrido, respectivamente.

6.2 Resultados obtidos na plataforma para modelagem 3D do algoritmo de RTM em FPGA

Nessa seção, estão presentes os resultados obtidos com o caso de uso desenvolvido. O *FPGA* utilizado foi o *Stratix IV*, referência *EP4SE530H35C2*. A plataforma de síntese utilizada foi o *QuartusII* versão 14.1 de 64 bits da *Altera*.

O caso de uso deste trabalho foi operado e testado também em *hardware*, ou seja, com toda a arquitetura presente em *FPGA*. Os resultados de síntese obtidos em nível de *Place and Route* na plataforma citada estão presentes na tabela 8, a seguir:

Tabela 8 - Recursos utilizados pela plataforma RTM para uma Stratix IV.

Total	PE Float	PE Híbrido
ALUTs	238,607	196,879
Logic Registers	303,098	283,646
DSP´s	684	748
Frequência	154.44 MHz	154.92 MHz

Pode-se observar na tabela 8 que a arquitetura com o PE Float ocupou 238.607 *ALUTs*. A arquitetura com o PE Híbrido ocupou 196.879 ALUTs, ou seja, uma redução de cerca de 17,49% da lógica ocupada em relação ao PE Float.

Em termos de número de *DSP´s* utilizados, a arquitetura com o PE Float ocupou 684, e a com o PE Híbrido, o que representa um aumento de cerca de 8,56% do uso de DSPs em relação ao PE Float.

Na plataforma de modelagem 3D a *FPGA* utilizada possui um bom número de blocos DSP´s, 1024 ao total, que não são utilizados pelos demais componentes da arquitetura do *RTM*. O mesmo não ocorre com a lógica disponível em *FPGA*, que é utilizado pelos componentes da plataforma e, portanto, tinha que ser economizada pelo núcleo de processamento. Com isso o número de DSP´s utilizados pelo núcleo aritmético não possuí impacto significativo na arquitetura geral e o PE Híbrido se torna mais eficiente em termos de recursos utilizados.

As frequências obtidas nas duas arquiteturas foram as mesmas e próximas de 154MHz. Isso demonstra que o PE Híbrido, além de ocupar menor área em *FPGA*, não teve sua frequência de operação degradada.

A tabela 9 a seguir expõe os resultados em termos percentuais, o que torna mais fácil a análise dos ganhos entre os PE Híbrido e PE Float.

Tabela 9 - Recursos utilizados pela plataforma RTM em termos percentuais para uma Stratix IV.

	PE Float	**PE Híbrido**	**Total**
ALUTs	56,15%	46,79%	424,960
Logic Registers	71,32%	66,75%	424,960
DSP´s	66,79%	73,14%	1024
Total Pins	98%	98%	744
Logic Utilization	100%	79%	100%

Analisando-se a tabela 9, têm-se os resultados obtidos com o PE Híbrido. Nota-se a redução no número de *ALUT´s* de 56,15% para 46,79%. Esse é resultado é importante, pois quanto menos área o PE ocupa, mais PE´s poderão estar presentes na arquitetura.

Outro fator relevante é que o peso da área ocupada pelo *PE* em relação aos demais componentes da arquitetura tornou-se menor, liberando, assim, espaço no *FPGA* para a inserção de novos componentes da plataforma 3D.

Pode-se notar que o número de *DSP´s* utilizados sofreu um leve aumento, pois parte da lógica utilizada, que antes era espalhada em *ALUT´s*, migrou para o uso de *DSP´s*. Devido ao fato de os *DSP´s* presentes na placa não serem essenciais para os demais componentes da arquitetura, o *PE* ficou livre para usar esse recurso sem grandes restrições.

Outro recurso utilizado que merece destaque é o número de pinos da *FPGA*. Pode-se observar que ambas as arquiteturas utilizaram quase que a totalidade dos pinos disponíveis para comunicação do *FPGA*. Isso demonstra que a plataforma 3D desenvolvida estava operando no limite da comunicação disponível para o *FPGA* utilizada. Além disso, mostra o quanto o projeto foi otimizado para utilizar o máximo de recursos disponíveis no *FPGA*. Com isso, mesmo que os componentes não ocupassem a totalidade da área disponível, não seria possível se instanciar mais componentes sem uma mudança na arquitetura. Um *FPGA* com uma maior quantidade de pinos permitiria um maior número de componentes instanciados, ao custo de mudanças na arquitetura final.

Por fim, na tabela 9, pode-se notar que o total de lógica utilizada foi de 100% com o PE Float para 79% com o PE Híbrido. A *Altera* descreve essa métrica como uma medida para o quão cheio um dispositivo está baseado em componentes utilizados. Assim, observa-se que o *FPGA* encontrava-se saturado na plataforma 3D, sem nenhuma folga disponível quando da utilização do PE Float.

Por outro lado, na versão com o PE Híbrido, o uso de lógica caiu consideravelmente, liberando espaço para eventuais ajustes na arquitetura, na adição de outros módulos, seus posicionamentos e roteamento de novas redes.

Capítulo

7

7. Conclusões e Trabalhos Futuros

Neste capítulo, estão expostas as conclusões obtidas com o desenvolvimento deste trabalho.

Esse trabalho abordou o desenvolvimento de um módulo aritmético, padrão IEEE-754 para dispositivos reconfiguráveis, *FPGAs*. O módulo realiza a parte aritmética do algoritmo de Sísmica *RTM* (2010), que faz uso de um operador de diferenças finitas para aproximação da equação de onda utilizada no mapeamento sísmico do solo (Santos, 2012).

Inicialmente foi feita uma revisão sobre diversos temas relevantes ao trabalho. Dentre eles, pode-se citar: Computação de alto desempenho, *FPGA*, padrão IEEE-754, Aritmética de Ponto Fixo, o algoritmo de mapeamento sísmico, *RTM*, entre outros.

Na seção seguinte, foram avaliados trabalhos relacionados ao tema dessa dissertação, que envolvem núcleos aritméticos em ponto flutuante para *FPGAs*. Dentre esses trabalhos, estão presentes os de Yong Dou (2005), Abner Barros (2008), Rodrigo Rocha (2010), que são mais relevantes para essa dissertação.

Além desses trabalhos pode-se citar também os de Amaricai (2013), Wilson (2014) e Ramesh (2013) que também apresentarem contribuições ainda que em menor grau.

No capítulo 4 foi abordada a plataforma para resolução do algoritmo de análises sísmicas *RTM* 3D para uma *FPGA* Stratix IV. O núcleo aritmético desenvolvido foi incorporado nessa arquitetura para cálculo da etapa de modelagem do *RTM*. Utilizou-se a etapa de modelagem do *RTM* por critérios de simplicidade de implementação.

A plataforma de sísmica utilizada foi a desenvolvida em parceria com a Petrobras (CENPES) no projeto *HPCIn* do CIn-UFPE (2014). Essa arquitetura completa foi implementada fisicamente em *FPGA* e fez uso do PE Híbrido e do PE Float, bem como, dos demais blocos necessários ao funcionamento de seu algoritmo. Os PE´s desenvolvidos comportaram-se de maneira esperada na plataforma do *RTM* e atingiram os objetivos previstos nos testes.

As duas arquiteturas desenvolvidas para a resolução do operador de diferenças finitas foram abordadas no capítulo 5. Uma delas operando com os algoritmos aritméticos para números em ponto flutuante. A outra, com entradas e saídas em ponto flutuante e internamente com trechos com aritmética de ponto fixo. A primeira arquitetura foi denominada de PE Float e a segunda de PE Híbrido.

O PE Float atingiu os objetivos de precisão e frequência esperados. Conseguiu-se, portanto, realizar o papel de núcleo aritmético do *RTM*. O PE Híbrido reduziu parte da lógica em *FPGA* para a implementação do núcleo aritmético sem afetar no resultado do cálculo do operador de diferenças finitas. Dessa forma, um nova arquitetura capaz de realizar a mesma operação que o PE Float e com precisão semelhante foi feita.

O PE Híbrido opera em frequência semelhante ao PE Float, com uma economia de unidades lógicas de 17,49%. Os resultados de síntese obtidos para o *FPGAs Stratix IV* foram apresentados no capítulo 6.

Na plataforma do *RTM* 3D, observou-se uma redução de 56,15% para 46,79% do número total de *ALUT´s*, utilizadas em uma *FPGA* Stratix IV, pela plataforma de sísmica da versão PE Float para a versão PE Híbrido. Portanto, o PE Híbrido liberou espaço presente em *FPGA* para o uso de mais componentes de processamento.

Apesar dessa economia de lógica na *FPGA* com o PE Híbrido, não foi possível se instanciar mais PE´s além dos 16 presentes na plataforma de *RTM* 3D. Isso ocorre visto que o número de pinos disponíveis e utilizados na *Stratix IV* é limitado.

Um fator de destaque do PE Híbrido, é que mesmo com o ganho de área obtido, não houve impacto na frequência de operação da plataforma de sísmica, que se manteve em cerca de 150 MHz.

Diante disso, o presente trabalho atingiu os objetivos inicialmente previstos. Tanto em termos de confiabilidade de resultados, quanto em termos de eficiência. A versão final do PE Híbrido atendeu os requisitos de uso e se tornou funcional na plataforma de Sísmica do projeto *HPCIn*.

Como possíveis trabalhos futuros, pode-se citar o estudo do impacto da redução de bits da representação em ponto flutuante, com o objetivo de ganho de área em *FPGA*. Seria necessário verificar até onde a redução do número de bits afetaria a precisão final do resultado. Além disso, verificar também, se este resultado final estaria dentro dos limites de precisão e operação esperados para a plataforma sísmica 3D, para o algoritmo *RTM*.

Capítulo

8

8. Referências.

Neste capítulo, estão expostas as referências relevantes para a elaboração desta dissertação.

ALTERA. **ALM**. Acesso em: 29 jul. 2015. Disponível em: <https://www.altera.com/products/fpga/features/stx-architecture.html>.

ALTERA. **DSP**. Acesso em: 29 jul. 2015. Disponível em: <https://www.altera.com/products/fpga/features/stx-dsp-block.html>.

ALTERA. **LUT.** Acesso em: 29 jul. 2015. Disponível em: <https://www.altera.com/products/general/fpga/stratix-fpgas/stratix-ii/stratix-ii/features/architecture/st2-lut.html>.

ALTERA. **Overview Stratix III**. Acesso em: 29 jul. 2015. Disponível em: <https://www.altera.com/content/dam/altera_www/global/en_US/pdfs/literature/hb/stx3/stx3_siii51001.pdf>.

ALTERA. **Overview StratixIV**. Acesso em: 16 jul. 2015. Disponível em: <https://www.altera.com/products/fpga/stratix-series/stratix-iv/overview.html>.

ALTERA. **QuartusII**. Acesso em: 16 jul. 2015. Disponível em: <http://www.altera. com/literature/manual/intro_to_quartus2.pdf> .

ALTERA. **Stratix III Handbook.** Acesso em: 29 jul. 2015. Disponível em: <http://www.altera.com/literature/hb/stx3/stx3_siii5v1.pdf>.

ALTERA. **TimeQuest Timing Analyser**. Acesso em: 16 jul. 2015. Disponível em: <https://www.altera.com/support/support-resources/design-examples/design-software/timequest/sof-qts-timequest.html>.

AMARICAI, A. et al.. **FPGA Implementation of Hybrid Fixed Point Floating Point Multiplication**. Mixed Design of Integrated Circuits and Systems (MIXDES), 2013 Proceedings of the 20th International Conference, 2013, pp. 243-246.

BARROS, A. C., Barbosa J. e Lima M. E. **Implementation of a double-precision multiplier accumulator with exception treatment to a dense matrix multiplier module in FPGA**, 21nd Symposium on Integrated Circuits and Systems Design (SBCCI), pp. 40-45, Gramado, Rio Grande do Sul,2008.

BARROS, A. B. **Uma Metodologia Para a Determinação Da Precisão Numérica Necessária à Implementação do Algoritmo RTM.** 2014. Tese de Doutorado — CIn UFPE. Pernambuco: Centro de Informática da Universidade Federal de Pernambuco.

BARROS, A. C. et al. **Performance evaluation model based on precision reduction and FPGAs applied to seismic modeling**. In: SIMPÓSIO EM SISTEMAS COMPUTACIONAIS (WSCAD-SSC),, Petrópolis, RJ. New York. Proceedings. . . IEEE, 2011. p. 2–2.

BRAGANÇA, R. et al. **Seismic Modeling and RTM Migration on unconventional Hardware**. In: INTERNATIONAL CONGRESS OF BRAZILIAN GEOPHYSICAL SOCIETY & EXPOGEF, 13. Rio de Janeiro, 2013.

BRITANNICA ENCYCLOPAEDIA. **ENIAC**. Acesso em: 7 jul. 2015, Disponível em: <http://global.britannica.com/technology/ENIAC>.

BROCK, David C. **Understanding Moore´s Law**. 1.ed. Chemical Heritage Foundation, 2006. p.82.

COE, T. Inside the Pentium Fdiv bug. Dr. Dobbs Journal (April 1996), pp. 129–135.

CLCBIO. Acesso em: 29 jun. 2015. Disponível em: <http://www.clcbio.com/index.php>.

DOU, Yong. et al. **64-bit floating-point FPGA matrix multiplication.** 2005.

DURBANO, J. P. et al. **FPGA-Based Acceleration of the 3D Finite-Difference Time-Domain Method**. IEEE Symposium on Field-Programmable Custom Computing Machines, 2004.

DUTRA Bruno. H. **Desenvolvimento de Uma Plataforma com uma Arquitetura Escalável para Multiplicação de Matrizes Densas em Sistemas Reconfiguráveis Matrizes Densas em Sistemas Reconfiguráveis de Alto Desempenho**. 2010. Dissertação de Mestrado — CIn UFPE. Pernambuco: Centro de Informática da Universidade Federal de Pernambuco.

FERNANDES, J. P. B. **Implementação do operador de diferenças finitas em sistemas de Computação Reconfigurável para modelagem sísmica.** 2010. Trabalho de Graduação — CIn UFPE. Pernambuco: Centro de Informática da Universidade Federal de Pernambuco.

FU, H. **Application-Specific Number Representation**. 2009. Tese de Doutorado — Imperial College London Department of Computing Application-Specific.

FU, H. et al. **Accelerating seismic computations using customized number representations on FPGAs**. EURASIP Journal on Embedded Systems, [S.l.], v. 2009, n. 3, p. 382983, Jan. 2009.

GIDEL. **Plataforma PROCe III**.. Acesso em: 16 jul. 2015. Disponível em: <http://www.gidel.com/PROCe%20III.htm>

GIDEL. **Plataforma PROCe IV**. Acesso em: 16 jul. 2015. Disponível em: <http://www.gidel.com/pdf/PROCStarIV%20Product%20Brief.pdf>

GIDEL. **ProcWizard**. Acesso em: 16 jul. 2015. Disponível em: <http://www.altera.com/literature/manual/intro_to_quartus2.pdf>.

GONZALEZ, Rafael; WOODS, Richard. **Digital Image Processing**. 2. ed. Prentice Hall, 2002.

GUO, Z. et al. **A Quantitative Analysis of the Speedup Factors of FPGAs over Processors**. In: ACM/SIGDA 12TH INTERNATIONAL SYMPOSIUM ON FIELD PROGRAMMABLE GATE ARRAYS, 2004., New York, NY, USA. Proceedings... ACM, 2004. p. 162–70. (FPGA ’04).

HENNESSY, John L.; PATTERSON, David A. **Computer architecture: A quantitative approach.** 4 ed. Boston: Morgan Kaufmann, 2007.

HPCIN. **HPCIn**. Acesso em: 15 jul. 2015. Disponível em: <http://www.cin.ufpe.br/~ hpcin/hpcin.php>.

INTEL. **PENTIUM 4**. Acesso em: 7 jul. 2015. Disponível em: <http://www.intel.com. br/content/www/br/pt/intelligent-systems/previous-generation/embedded-pentium-iv.html>.

INSTITUTE, A. N. S. (Ed.). ANSI/IEEE 754-1985, **Standard for Binary Floating-Point Arithmetic**. [S.l.]: pub-IEEE-STD, 1985.

OPENCL. **KHRONOS.** Acesso em 08 de nov. 2015. Disponível em: < https://www. khronos.org/opencl/>

MARMOUSI. Proposto pelo IFP (Institut Français du Pétrole) em 1988. Citado em IRONS, T. 2008. **Marmousi Model**. Disponível em: <http://www.reproducibility.org/ RSF/ book/ data/marmousi/paper_html/>.

MEDEIROS, V. W. C. de. fastRTM: **Um Ambiente Integrado para Desenvolvimento Rápido da Migração Reversa no Tempo (RTM) em Plataformas FPGA de Alto Desempenho**. 2013. Tese de Doutorado — CIn UFPE. Pernambuco: Centro de Informática da Universidade Federal de Pernambuco.

MEDEIROS, V. et al. **High Performance Implementation of RTM Seismic Modeling on FPGAs: architecture, arithmetic and power issues**. In: VANDERBAUWHEDE, W.; BENKRID, K. (Ed.). High-Performance Computing Using FPGAs. New York: Springer, 2013. p. 305–34.

MENEZES, G. et al. **Energy Estimation Tool FPGA-based Approach for Petroleum Industry**. In:INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING WORKSHOPS (ICPPW),2012, 41., Pittsburgh, PA. New York. Proceedings... [S.l.: s.n.], 2012. p. 600–1.

MOORE, G.E. **The microprocessor: Engine of the technology Revolution**. In: Communications of the ACM, 1997.

NASA. **Curiosity**. Acesso em: 31 ago. 2015. Disponível em: <https://www.nasa. gov/ mission_pages/msl/overview/index.html>.

NAVABI, Zainalabedin. **Embedded Core Design with FPGAs**. 1. ed. Boston, 2007.

NVIDIA. **GPU**. Acesso em: 15 jul. 2015. Disponível em: <http://www.nvidia.com/ object/ what-is-gpu-computing.html>.

PALMER, John. **The Intel 8087 Numeric Processor**, ACM. 1980.

PETROBRAS. **TECNOLOGIAS**. Acesso em: 31 ago. 2015. Disponível em: <http://presal. hotsitespetrobras.com.br/tecnologias-pioneiras/#3>.

PETROBRAS. **PRÉ-SAL**. Acesso em: 31 ago. 2015. Disponível em: <http://www.petrobras. com.br/pt/energia-e-tecnologia/fontes-de-energia/petroleo/presal/>.

PGS. **Reverse Time Migration**. In: Tech Link, A Publication of Petroleum Geo-Service Vol. 7, No. 1, 2007.

ROCHA, Rodrigo Camarotti Ferreira. **Desenvolvimento de uma Plataforma Reconfigurável para modelagem 2D, em Sísmica, Utilizando FPGA.** Dissertação de Mestrado. Universidade Federal de Pernambuco, Recife-PE, 2010.

SANTOS, J. L. R. dos. **Modelagem da equação da onda acústica aplicada ao imageamento de estruturas geológicas**. 2012. Dissertação de Mestrado - Coppe UFRJ. Rio de Janeiro: Instituto Alberto Luiz Coimbra de Pós-Graduação e Pesquisa de Engenharia.

SMITH, M. J. S. **Application-Specific Integrated Circuits**, Addison-Wesley, 1997.
TOP 500.Org.. **TOP500**. Acesso em: 27de Ago. 2015. Disponível em: <http://www.top500.org>.

WANG, Z.; BOVIK, A. C. **A universal image quality index**. IEEE Signal Processing Letters, [S.l.], v. 9, n. 3, p. 81–4, Mar. 2002.

WILSON. Jose, A.R. Silva, H.C. Neto, M.P. Véstias. **Efficient implementation of a single-precision floating-point arithmetic unit on FPGA**. IEEE, in Proc. FPL, 2014, pp.1-4.

XILINX. **FPGA**. Acesso em: 3 set. 2015. Disponível em: < http://www.xilinx.com/fpga/>.

XILINX. **Mars Exploration Rovers**. Acesso em: 31 ago. 2015. Disponível em: <http://www.xilinx.com/about/customer-innovation/aerospace-and-defense/mars-exploration-rovers.html>.

XILINX. **Virtex-6 Overview**. Acesso em: 31 ago. 2015. Disponível em: <http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf>.

Capítulo

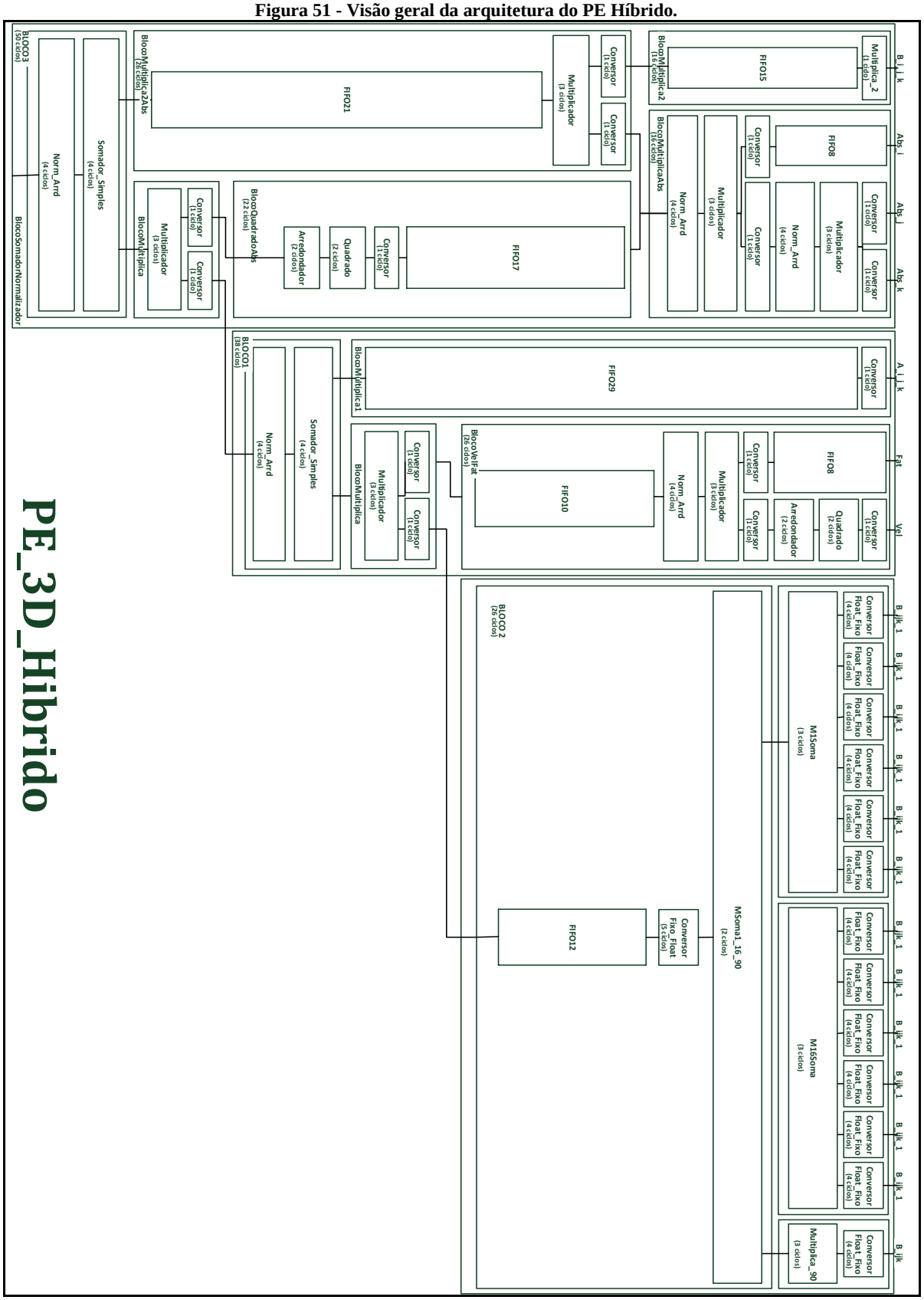
9

9. Anexos.

Neste capítulo, estão expostos os anexos relevantes para a elaboração desta dissertação.

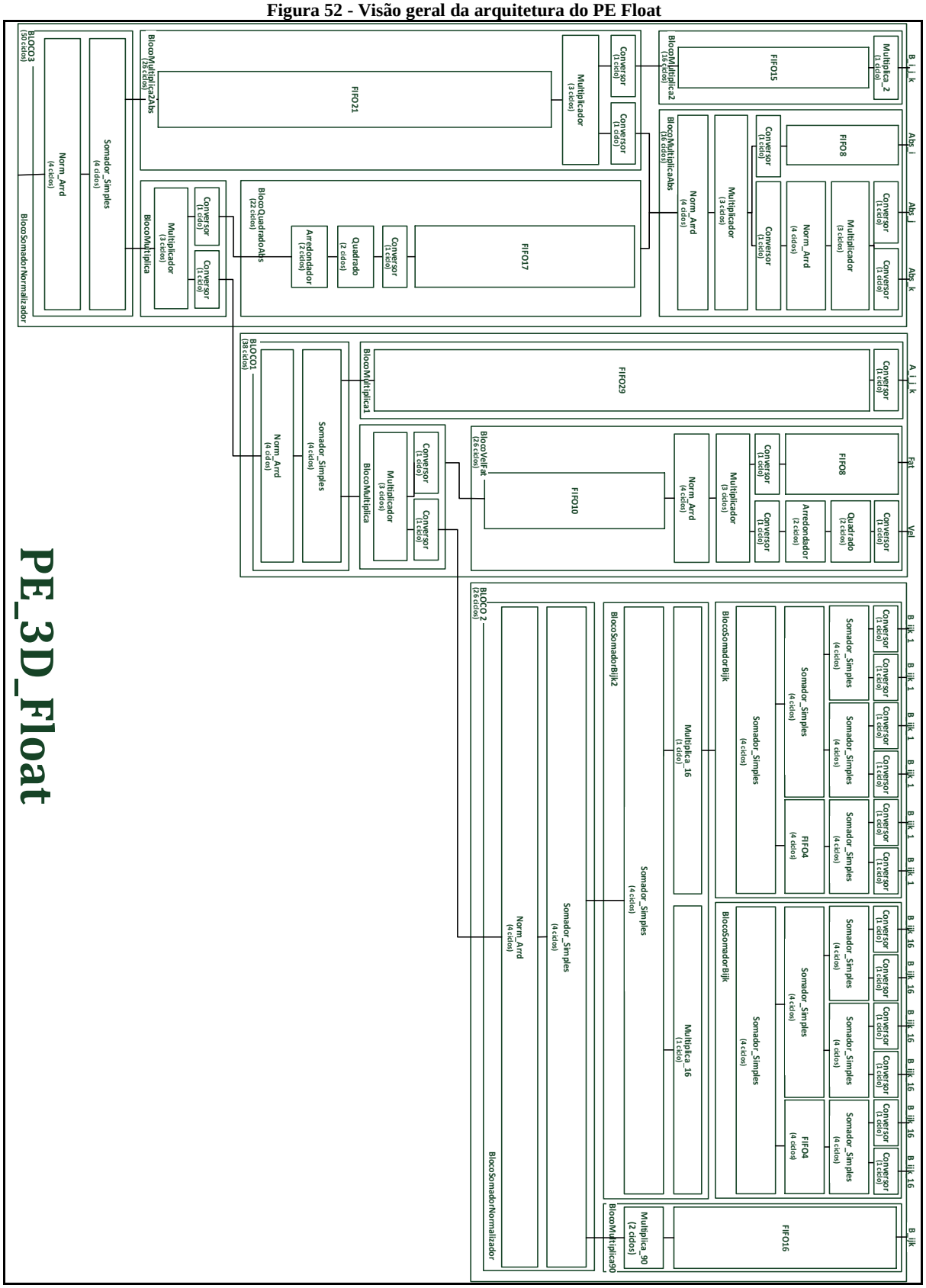
9.1 ANEXO I - Arquitetura do PE Híbrido

Figura 51 - Visão geral da arquitetura do PE Híbrido.



9.2 ANEXO II - Arquitetura do PE Float

Figura 52 - Visão geral da arquitetura do PE Float



9.3 ANEXO III - Arquitetura do RTM

Figura 53 - Visão geral da arquitetura da Plataforma 3D para RTM em FPGA.