UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA MECÂNICA

TESE DE DOUTORADO EM SISTEMAS MECATRÔNICOS

IMPLEMENTAÇÃO EM *HARDWARE* RECONFIGURÁVEL DE OPERADORES MATRICIAIS PARA SOLUÇÃO NUMÉRICA DE SISTEMAS LINEARES

JANIER ARIAS GARCÍA

ORIENTADOR: CARLOS H. LLANOS

COORIENTADOR: MICHAEL HÜBNER
Universidade de Ruhr - Bochum, Alemnha

PUBLICAÇÃO: ENM.TD - 09/14
BRASÍLIA-DF, 14 de Novembro de 2014

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA MECÂNICA

IMPLEMENTAÇÃO EM *HARDWARE* RECONFIGURÁVEL DE OPERADORES MATRICIAIS PARA SOLUÇÃO NUMÉRICA DE SISTEMAS LINEARES

JANIER ARIAS GARCÍA

TESE DE DOUTORADO APRESENTADA AO PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS MECATRÔNICOS DO DEPARTAMENTO DE ENGENHARIA MECÂNICA DA UNIVERSIDADE DE BRASÍLIA, COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE DOUTOR EM SISTEMAS MECATRÔNICOS.

APROVADA POR:

Prof. Carlos Humberto Llanos Quintero, Dr. (ENM-UnB)
(Orientador)

Prof. Frank Sill Torres, PhD. (UFMG)
(Examinador Externo)

Prof. Edward David Moreno, PhD. (UFSe)
(Examinador Externo)

Prof. Alexandre Ricardo Soares Romariz, PhD. (UnB)
(Examinador Externo)

Prof. Guilherme Caribé de Carvalho, PhD. (UnB)
(Examinador Interno)

BRASÍLIA-DF, 14 de Novembro de 2014

FICHA CATALOGRÁFICA

ARIAS-GARCIA, JANIER.
Implementação em *hardware* reconfigurável de operadores matriciais para solução numérica de sistemas lineares [Distrito Federal] 2014.
xviii, 178 p. 210 x 297 mm (ENM/FT/UNB, Doutor, Sistemas Mecatrônicos, 2014).
Tese de Doutorado. Universidade de Brasília. Faculdade de Tecnologia.
Departamento de Engenharia Mecânica.
1. FPGAs 2. Ponto Flutuante
3. Solução de Sistemas Lineares 4. Operadores Matriciais
I. ENM/FT/UNB II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

ARIAS-GARCIA, JANIER. (2014). Implementação em *hardware* reconfigurável de operadores matriciais para solução numérica de sistemas lineares. Tese de Doutorado em Mecatrônica, Publicação ENM.TD - 09/14, Departamento de Engenharia Mecânica, Universidade de Brasília, Brasília, DF, 178 p.

CESSÃO DE DIREITOS

NOME DO AUTOR: Janier Arias García

TÍTULO DA TESE DE DOUTORADO: Implementação em *hardware* reconfigurável de operadores matriciais para solução numérica de sistemas lineares.

GRAU / ANO: Doutor / 2014

Janier Arias García
SCLN 212 Bloco B, apt 204
70864-520 Brasília/DF, Brasil

Dedicatória

A minha amada esposa

Agradecimentos

Depois de quatro anos são muitas as pessoas que foram parte especial e contribuíram na conclusão de esta etapa, é para eles que escrevi estas linhas...

Agradeço a meus pais Luz Marina García e Soriel Arias, pelo ânimo e apoio incondicional durante esta jornada. São sem duvida um exemplo e me sinto grato por poder compartilhar um logro mais com vocês. A minha doce irmã Nadina Arias, quem sempre teve um sorriso para mim.

Ao meu orientador Prof. Carlos Humberto Llanos Quintero pela paciência e dedicação. Em terras estrangeiras encontrar orientação e conhecimento são inestimáveis, mais ainda, de quem é e será sempre mais do que um orientador, um grande amigo.

Ao Prof. Mauricio Ayala Rincón pela coorientação, incentivo, suporte e pelo exemplo de dedicação e disciplina. Ao Prof. Ricardo Jacobi pelas ideias, observações e sugestões durante a fase de desenvolvimento deste trabalho.

Aos meus colegas e amigos, especialmente ao Daniel Muñoz, Jones Yudi, André Braga, Sergio Cruz, Renato Sampaio, Jesus Pinto, Gloria Lopéz, Camilo Sanchez e Magno Correa pela ajuda, conselhos, também pelas alegrias compartidas e pela companhia nos momentos difíceis.

A todos os colegas que de uma forma ou outra contribuíram com o desenvolvimento deste trabalho e me brindaram momentos felizes nestes anos em Brasília.

A CAPES pelo suporte financeiro.

Finalmente, este trabalho é dedicado a minha esposa Diana Paola Gómez Mendoza, quem com tanto amor soube me dar palavras de ânimo e sabedoria para alcançar esta etapa..."gracias amor".

RESUMO

Este trabalho apresenta um estudo da implementação de operadores matriciais para solução numérica de sistemas lineares em FPGAs (*Field Programmable Gate Arrays*). As arquiteturas foram baseadas nos métodos diretos *QR*, *de Schur*, assim como na Eliminação Gaussiana. Os métodos foram desenvolvidos usando topologias orientadas a controle e fluxo de dados com representação aritmética de ponto flutuante, permitindo explorar o paralelismo intrínseco dos diferentes algoritmos para solução de sistemas lineares. Desta forma, mantendo o controle da propagação do erro e ganhos de desempenho em termos do tempo de execução, visando a sua aplicabilidade em problemas inversos. As arquiteturas foram desenvolvidas para obter a inversa de uma matriz assim como a solução de um sistema de equações lineares, baseados no método de eliminação Gaussiana (ou sua variante Gauss-Jordan). Além disso, neste trabalho foi proposta e implementada uma nova arquitetura baseada no método de Schur formada pelos seguintes circuitos: QRD-MGS (*QR Decomposition via Modified Gram-Schmidt*), MMM (Multiplicação Matriz-Matriz) e MDTM (Multiplicação-Diagonal-Transposta-Matriz). Adicionalmente, estudos de consumo de recursos para diferentes tamanhos de matrizes assim como uma análise da propagação do erro foram realizados no intuito de verificar a aplicabilidade dos algoritmos em arquiteturas reconfiguráveis. Neste trabalho, o modulo de Eliminação Gaussiana desenvolvido foi usado para apoiar os cálculos de uma rede neuronal do tipo GMDH na predição da estrutura 3D de uma proteína. Finalmente, foram implementadas duas metodologias, Fusão de *Datapath* para manter o controle da propagação de erro usando apenas uma representação com precisão simples e a Verificação/Validação para realizar uma padronização na validação dessas implementações.

Palavras-chave: FPGAs, Decomposição QR, Método de Schur, Eliminação Gaussiana, Sistema Linear.

ABSTRACT

This work presents a study on the implementation of matrix operators for the numerical solution of linear systems on FPGAs (Field Programmable Gate Arrays). The architectures were based on direct methods such as QR, Schur as well as the Gaussian elimination. The methods were developed using topologies oriented to both control and to data-flow with a floating point arithmetic representation, exploring the intrinsic parallelism of different algorithms for solving linear systems. Thus, the developed architectures have been achieved maintaining both the control of the error propagation and performance gains in terms of runtime, seeking their applicability in inverse problems. The architectures have been developed to deal with the inverse of a matrix as well as for solving a system of linear equations based on the Gaussian elimination method (or its Gauss-Jordan variant). Additionally, this work has proposed and implemented a novel architecture based on the Schur method composed of the following circuits: QRD-MGS (QR Decomposition via Modified Gram-Schmidt), MMM (Matrix-Matrix Multiplication) and MDTM (Matrix-Diagonal-Transpose-Multiplication). Furthermore, this work presents studies of the resource use for different sizes of matrices as well as the error propagation analysis in order to verify the applicability of the algorithms on reconfigurable hardware. Additionally, the Gaussian elimination module developed in this work was used to support the calculations of a GMDH neural network on an application to predict the 3D structure of a protein. Finally, two methodologies were implemented, the Datapath Fusion to maintain the control of the error propagation using only one representation with single precision and the Verification/Validation to create a benchmark to validate the results of the hardware implementations.

Keywords: FPGAs, QR Decomposition, Schur Method, Gaussian Elimination, Linear System.

Sumário

Lista de Tabelas

Lista de Figuras

Lista de Símbolos

Siglas

ASIC	Application Specific Integrated Circuits
CLBs	Blocos Lógicos Configuráveis
DSPs	Digital Signal Processing
EDIF	Electronic Design Interchange Format
EPROM	Erasable Programmable Read-Only Memory
FF	Flip-flops
FFT	Transformada Rápida de Fourier
FPGAs	Field Programmable Gate Array
FSM	Finite State Machine
GMDH	Group Method of Data Handling
GFLOP	Giga FLoating-point Operations Per Second
GPPs	General Purpose Processors
GPU	Graphics Processing Unit
HIL	Hardware in the loop
HPC	High-Performance Computing
HPRC	High Performance Reconfigurable Computing
HRCS	High Radix Carry-Save
HW/SW	Hardware/Software
LUT	Look Up Table
MAE	Mean Absolute Error
MSE	Mean Square Error
NGC	Xilinx-specific netlist files
NRE	Non-Recurring Engineering
PC	Personal Computer
PE	Elemento de Processamento
PLB	Processor Local Bus
PLL	Phase-Locked Loop

RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RN	Arredondado para o mais proximo
RTL	Register Transfer Level
SRL	Registradores de Deslocamento LUT
ULA	Unidade lógica e aritmética
VHDL	Very High Speed Integrated Circuits Hardware Description Language
VLSI	Very Large Scale Integration
XST	Xilinx Synthesis Tool
XSG	Xilinx System Generator

Arquiteturas de *hardware*

CAM	Bloco Comparar e Alinhar a Mantissa
FP-Add	Floating-point Addition/subtraction unit
FP-Mul	Floating-point Multiplication unit
FP-FMAC	União Multiplicação Soma Acumulação em Ponto Flutuante
GEM	Gaussian Elimination Module
GJ	Gauss-Jordan Elimination
MatrixSqRt	Matrix Square Root
MDTM	Multiplicação Diagonal Transposta Matriz
MGS	Modified Gram-Schmidt
MMM	Multiplicação Matriz Matriz
NM	Bloco de Normalização da Mantissa
NP	Processo de Normalização
OP	Processo de Ortogonalização
QRD-GR	QR Decomposition via Givens Rotations Method
QRD-MGS	QR Decomposition via Modified Gram-Schmidt Method
SSL	Linear System Solution

Capítulo 1 INTRODUÇÃO

1.1 Aspectos Gerais

As matrizes saíram da sombra dos determinantes apenas há um pouco mais de 150 anos para ser parte importante de muitas demonstrações e modelos matemáticos, que atualmente se encontram nas diferentes áreas da ciência. *Cauchy*, segundo a história foi o primeiro a lhes dar um nome "*table*" por volta de 1826 [1]. Já o nome de matriz veio só com o *James Joseph Sylvester* em 1850, mas foi o *Cayley*, quem em 1858 divulgou esse nome e veio a demonstrar sua utilidade na sua obra "Memoir on the Theory of Matrices" [2].

A história das matrizes remonta-se aos tempos 200 AC e 300 BC, onde os chineses já mostravam na sua literatura estudos de sistemas com equações lineares simultâneas. O primeiro exemplo do uso das matrizes para resolver equações simultâneas encontra-se na obra "*Nove capítulos sobre o arte matemática*" [3]. Nesse livro são apresentados 246 problemas baseados em uma reunião de textos matemáticos sobre impostos, cálculos, engenharia, soluções de equações, agricultura e propriedades dos triângulos retângulos, entre outros. Esse livro mostra também o conceito do determinante pela primeira vez, 2000 anos antes da sua publicação pelo matemático japonês *Seki Kowa* em 1683 ou pelo matemático alemão *Gottfried Leibniz* em 1693.

Depois do desenvolvimento da teoria dos determinantes por *Seki Kowa* e *Leibniz*, o matemático *Gabriel Cramer* apresentou em 1750 o que agora é denominado *regra de Cramer* na sua obra "*Introduction à l'analyse des*

lignes courbes algébriques", facilitando a solução das equações lineares [4]. Contudo, foi o *Cayley* quem introduziu a notação de matrizes, como forma abreviada de escrever um sistema de m equações lineares com n incógnitas [2]. A partir desse momento os trabalhos sobre matrizes aumentaram consideravelmente.

Assim, ao longo dos anos vem sendo necessário representar, manipular e calcular a solução numérica de sistemas matriciais em uma grande variedade de aplicações tais como o projeto de sistemas de controle, no estudo sobre mecânica de fluidos, processamento de sinais, simulações de ciência dos materiais, entre outras. Algumas dessas aplicações precisam um elevado tempo de processamento dos cálculos, devido ao envolvimento de matrizes de grande porte, cujo tamanho pode ir à ordem de centenas e mesmo milhares de linhas e colunas [5]. Fato que acontece em problemas que envolvem o estudo de campos elétricos, magnéticos, de tensões elásticas, térmicos ou ainda em problemas associados às redes estaduais de distribuição de energia elétrica, grandes redes de comunicação, etc. Em outras aplicações são necessários muitos cálculos por unidade de tempo (especialmente multiplicações matriciais), como no caso da geração dos movimentos e deformações, os quais podem ser observados nos efeitos especiais do cinema, da TV e dos jogos de computadores.

Mediante um processo de discretização esses problemas físicos são reduzidos a sistemas matriciais como por exemplo sistemas de equações lineares, cujas matrizes podem ter dimensões grandes [6, 7]. Desta maneira, pode-se obter uma sequência de passos que permite resolver numericamente um problema físico, tomando o seu modelo matemático definido através de equações diferenciais parciais (ver figura 1.1).

Historicamente, a necessidade de acelerar a solução numérica de sistemas matriciais usando computadores é facilmente demostrada nas aplicações

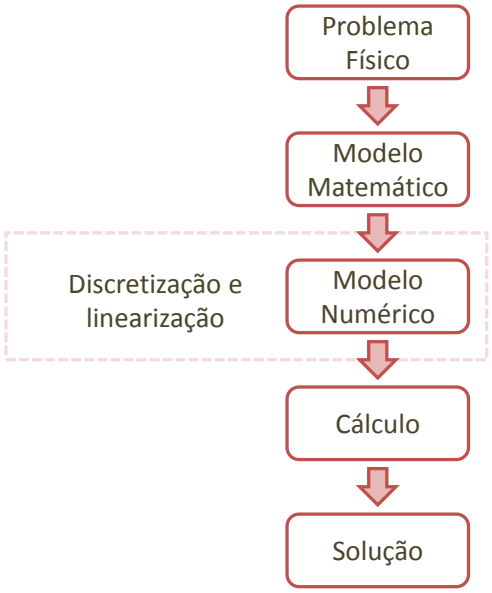


Figura 1.1. Sequência de passos para encontrar a solução numérica de um problema físico.

científicas da física, como o caso da previsão numérica de tempo. Na década dos 50, a primeira previsão numérica de tempo bem sucedida foi no primeiro computador digital eletrônico de grande escala, o ENIAC [8]. Já que 30 anos antes, a primeira previsão de tempo de seis horas levou seis semanas para ser terminada por *Lewis Fry Richardson* quem fez todos os cálculos sem ajuda computacional.

É tão alto o grau de complexidade computacional nos algoritmos para encontrar a solução numérica de sistemas lineares que foram propostos e definidos *benchmarks* para avaliar o desempenho de supercomputadores, tal como o *LINPACK benchmarks*. O *LINPACK* é um *software* com uma biblioteca para desenvolver operações numéricas da álgebra linear sobre computadores digitais. Desenvolvido em *Fortran* por *Jack Dongarra*, *Jim Bunch*, *Cleve Moler*, e *Gilbert Stewart*, foi concebido para utilização nos modernos supercomputadores vectoriais de memória compartilhada entre 1970 e começo dos anos 80. Atualmente, mudou seu nome para LAPACK sendo sua versão atual mais eficiente para as arquiteturas atuais, devido a que o mesmo foi projetado para explorar eficazmente os modernos sistemas

de memória cache. Assim, o LAPACK, alcança índices de desempenho de algumas ordens de magnitudes maiores com respeito a seu antecessor, o LINPACK [9].

Na atualidade, as simulações computacionais fazem uso de diferentes bibliotecas para calcular e obter uma solução numérica de problemas envolvendo sistemas matriciais. Assim, faz-se necessário o desenvolvimento de *hardwares* eficientes que permitam acelerar cada vez mais o desempenho em meios onde *clusters* de computadores (ou supercomputadores) não são uma opção, devido aos altos custos operacionais envolvidos.

1.2 O Papel da Álgebra Linear Numérica

Ao contrário dos métodos analíticos que conduzem a soluções exatas para os problemas, um método numérico produz, em geral, apenas soluções aproximadas. Devido ao fato de os computadores não terem uma memória infinita, as representações numéricas devem estar limitadas a um número de dígitos finitos, onde esse número de dígitos é chamado de *precisão* na representação numérica. Por isso, antes da utilização de qualquer método numérico é necessário decidir qual a precisão dos cálculos com que se pretende obter a solução numérica desejada. A precisão dos cálculos numéricos é um importante critério para a seleção de um algoritmo particular na resolução de um dado problema, sendo que a mesma fica atrelada à precisão numérica, assim como a outros fatores bem estudados no cálculo numérico[10].

Métodos algébricos para encontrar uma solução numérica a problemas da engenharia, onde há uma necessidade de modelar matematicamente mediante o uso de sistemas matriciais, vêm ganhando resultados no aspecto teórico, embora a sua implementação em *hardware* e sua posterior apli-

cação encontra-se limitada pelos avanços tecnológicos atuais, dada a sua complexidade computacional cúbica ($O(n^3)$) [11]. Devido às limitações tecnológicas geralmente presentes em sistemas embarcados, se obter um balanço entre os principais parâmetros (tais como *desempenho*, *precisão*, *memória*, *consumo de potência* e *consumo de recursos lógicos*) se faz necessário, chegando a ser um fator chave a fim de se obter a solução numérica adequada.

Na *Álgebra Linear Numérica*, área que trata os métodos e algoritmos numéricos computacionais onde existem cálculos matriciais, estudam-se e desenvolvem-se modelos que visam minimizar os passos necessários para resolver um problema da álgebra linear, assim como procurar os algoritmos com melhor precisão, cumprindo assim com o objetivo de se manter o balanço entre os parâmetros anteriormente mencionados. No entanto, encontrar esse balanço não é uma tarefa fácil, embora esteja sendo realizada por vários grupos de pesquisadores ao longo dos últimos anos [12, 13, 14, 15, 16].

Entre os objetivos da álgebra linear numérica estão a solução de sistemas lineares, resolver problemas com autovalores, encontrar a solução numérica de funções matriciais, entre outros. Neste contexto, têm-se métodos classificados de: (a) *diretos* e (b) *iterativos*, assim como técnicas desenvolvidas para lidar com matrizes tipo esparsas e/ou densas (ver figura 1.2). Cada um destes métodos pode apresentar vantagens e desvantagens dependendo do tipo de comparação a ser feita, mas nunca será uma solução geral para qualquer tipo de problema existente.

Uma matriz *esparsa* (ou dita *esparsa*) é uma matriz que possui uma grande quantidade de elementos zero ou sem valor/necessidade para o sistema. Essas matrizes estão em vários contextos físicos, por exemplo: (1) engenharia estrutural, (2) simulação de reservatórios, (3) redes elétricas e (4)

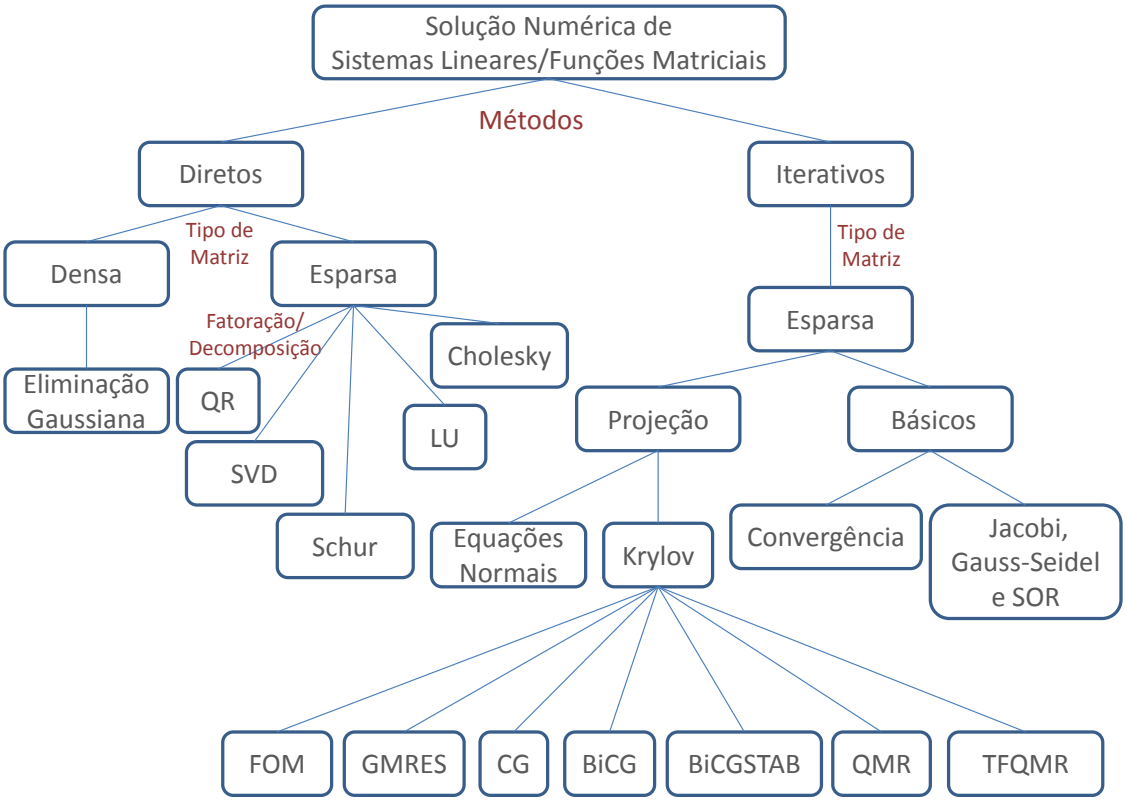


Figura 1.2. Classificação dos diferentes métodos para encontrar uma solução numérica em sistemas lineares e/ou funções matriciais. Esta classificação não é absoluta e apenas considera, no caso de métodos iterativos esparsos, os de projeção

problemas de optimização. O objetivo principal das técnicas com matrizes esparsas é desenvolver cálculos matriciais econômicos; por exemplo, sem envolver o armazenamento das entradas nulas. Já uma matriz dita *densa* tem todos os seus elementos diferentes de zero, matrizes normalmente achadas em sistemas físicos fortemente ligados.

Existem vários métodos diretos que podemos encontrar na literatura para resolver sistemas lineares com matrizes esparsas, sendo os principais a fatoração de *Cholesky*, LU, e QR [17]. Entre os citados um dos mais importantes é a fatoração QR, devido à permissibilidade que oferece o mesmo em relação ao tipo de matriz (definida positiva, singularidade, etc.). O processo de fatoração é simples teoricamente, basta transformar a matriz

A dada em duas matrizes: Q ortogonal e R triangular superior, tal que

$$A = QR.$$

O método encontra-se como parte de outras técnicas para resolver sistemas lineares, como o SVD (*Singular Value Decomposition*), o método de Schur ou métodos iterativos como o BiCGSTAB (*Biconjugate gradient stabilized method*), ver figura 1.2, em diferentes aplicações cientificas tais como problemas inversos. Esses problemas inversos são uns dos problemas matemáticos mais importantes e bem estudados. Os problemas inversos surgem em diferentes ramos da ciência e da matemática, incluindo visão computacional [18], estatística [19], geofísica [20], física [21] e muitas outras áreas.

A popularidade da teoria das matrizes está continuamente crescendo graças a sua grande aplicabilidade na modelagem e soluções de sistemas e problemas na engenharia. Operações matriciais tais como raiz quadrada, inversão entre outras, vêm ganhando uma especial atenção nos últimos anos devido aos avanços computacionais tecnológicos e à possibilidade de se desenvolverem algoritmos mais rápidos e estáveis sobre as plataformas computacionais modernas [22]. O interesse está focado nas vantagens da teoria, levando em conta que tais algoritmos prestam atenção a várias propriedades das matrizes e suas operações, gerando ganhos pela exploração de suas potenciais características e seus benefícios. Assim, algoritmos computacionalmente mais eficientes são alguns dos resultados obtidos, uma vez que os mesmos obtêm vantagens quando executados em sistemas computacionais de grande porte [5].

A função *raiz quadrada* de uma matriz é uma operação computacionalmente custosa e complexa, a qual requer o uso de processadores de alto desempenho para tratar com a decomposição e transformações de matrizes junto com operações em ponto flutuante. Neste sentido, o teorema de

fatoração de Schur é uma das técnicas matemáticas que podem ser usadas para resolver uma parte da raiz quadrada de uma matriz [23], devido a sua alta eficiência provida pela exploração de certos tipos de características matriciais, para as quais vários métodos numéricos conhecidos são aplicados (a grande maioria em *software*). Ainda sabendo que a teoria da raiz quadrada de matrizes é complexa, algumas simplificações podem ser feitas para certos tipos de matrizes. Por exemplo, quando uma matriz é simétrica semi-definida positiva (ver definições no apêndice B), tal matriz tem uma única raiz quadrada simétrica semi-definida positiva [24].

Cabe mencionar que os algoritmos QR e o *Método de Schur* possuem uma capacidade de paralelização inerente que pode ser utilizada em implementações *hardware*, incrementando assim o seu desempenho. Uma revisão bibliográfica do estado da arte mostra que são poucos os estudos que têm sido realizados visando paralelizar estas técnicas em conjunto (especialmente o método de Schur em uma única arquitetura), principalmente, quando aplicadas ao projeto de soluções embarcadas envolvendo sistemas matriciais. Fato devido à complexidade algorítmica, assim como a limitações tecnológicas impossibilitando implementações eficientes através de modelos computacionais tradicionais.

1.3 A Propagação do Erro nos Cálculos Numéricos

No trabalho com matrizes deve-se considerar que as mesmas são objetos matematicamente "mal condicionados", particularidade que as torna muito susceptíveis a pequenas variações nos elementos de entrada da matriz, podendo modificar por completo os resultados finais [25]. Como no exemplo 1.1, onde uma variação na centésima casa em apenas um dos elementos da matriz, transforma completamente os valores obtidos dos autovalores

(ver definições no apêndice B), produzindo dois sistemas lineares completamente diferentes.

***Exemplo* 1.1**

$$A_{ij} = \begin{pmatrix} -149 & -50 & -154 \\ 537 & 180 & 546 \\ -27 & -9 & -25 \end{pmatrix}$$

$$com\ autovalores\ \{1,\, 2,\, 3\}$$

e

$$A_{ij} = \begin{pmatrix} -149 & -50 & -154 \\ 537 & 180.01 & 546 \\ -27 & -9 & -25 \end{pmatrix}$$

$$com\ autovalores\ \{0.2073,\, 2.3008,\, 3.5019\}\,.$$

□

Assim, a manipulação correta dos dados é fundamental no trabalho com matrizes, uma vez que a escolha certa da representação dos dados, bem como o comprimento da palavra, pode ajudar a reduzir a diferença entre o valor obtido (aproximado) e o valor exato durante o cálculo que envolve o uso de matrizes. Essa diferença é chamada de *erro*.

Essa manipulação dos dados deve começar desde as mínimas unidades aritméticas que formam as operações matriciais, como no caso da fatoração QR, onde a mesma pode ser dividida em várias operações aritméticas fundamentais tais como: *soma*, *multiplicação*, *raiz quadrada*, e uma quinta operação chamada de *soma-acumulador*, segundo a referência [26]. Essa operação de soma-acumulador é importante nas operações de redução vetorial, como o módulo e/ou norma vetorial, onde de forma consecutiva uma simples soma S dos n elementos individuais x é feita [27], ver equação 1.1.

$$S = \sum_{i=1}^{n} x_i \tag{1.1}$$

Em áreas da computação cientifica tais como física, química e biologia, entre outros, a representação numérica de dados é um aspecto importante e um problema bem conhecido e nada fácil de ser tratado. O maior problema está em derivar o balanço entre a *precisão*, *desempenho*, *custo em área lógica*, *latência* e *consumo de potência* [28], especialmente quando usados diferentes formatos de representação numérica.

Duas representações podem ser utilizadas para efetuar as operações matriciais: (a) *representação em ponto fixo* ou (b) *representação em ponto flutuante*. A representação em ponto fixo não é a escolha mais adequada, pois os valores (faixa de valores) são desconhecidos na maioria das aplicações científicas e práticas, e não há forma de se dar um valor fixo para uma representação geral dos mesmos. Neste caso, a representação em ponto flutuante pode ser a escolha certa, pois apresenta uma ampla faixa dinâmica para representar qualquer valor real restrito apenas à precisão escolhida [29, 30]. Desta maneira, uma representação em ponto flutuante se apresenta mais apropriada para o tratamento e controle de possíveis erros de propagação associados aos cálculos matriciais; permitindo assim obter resultados com melhores precisões sem afetar a estabilidade das matrizes, parâmetro importante dada a condição de um sistema"mal condicionado".

Assim, usando um formato de alta precisão (por exemplo, em ponto flutuante) geram-se menores erros de quantização na implementação final, enquanto que um formato com baixa precisão pode gerar implementações de com melhor desempenho e com reduções em área e consumo de potência [31]. No final, um balanço entre área, desempenho e, em especial, com

respeito à propagação do erro associado aos cálculos matriciais deve ser realizado para decidir a melhor precisão a ser usada.

Em computação cientifica, várias aplicações usam os formatos padrões para representação numérica IEEE Std. $754-1985$ ou $754-2008$ (*IEEE Standard for Binary Floating-Point Arithmetic*), com precisão simples, dupla ou customizada [32]. Mas quando muitos formatos são suportados no mesmo ambiente computacional, se torna difícil saber em qual formato algumas operações serão desenvolvidas [33]. No entanto, alguns usuários devem enfrentar esse problema, onde em alguns casos uma representação com menor precisão pode ser suficiente e, ocasionalmente, uma representação com maior precisão seria necessária.

Desde o começo do uso dos computadores os problemas de precisão têm existido na representação computacional, uma vez que não todo número pode ser representado em um formato com representação fixa (ou capacidade de representação finita). Quando isto acontece, o uso de uma técnica de arredondamento adequada pode ser introduzida para representar os números de uma melhor forma, à custa de uma perda de precisão [34]. Esse tipo de perda é comumente chamado de erro de arredondamento, ver figura 1.3. O erro de arredondamento assim como os de truncamento e inerentes ao modelo são introduzidos quando um problema físico é modelado matemáticamente, seguido de uma modelagem numérica para finalmente, realizar cálculos e obter a solução (ver figura 1.3).

Um desses erros é o duplo arredondamento. Acontece quando o resultado de uma operação, primeiramente arredondado para caber em algum destino intermediário q e depois enviado de volta para o seu destino final p, não tem o mesmo resultado ao realizar o arredondamento diretamente para a posição p, tal como mostrado no exemplo 1.2. Em computação, o resultado da operação é tipicamente arredondado para caber em um registrador, o

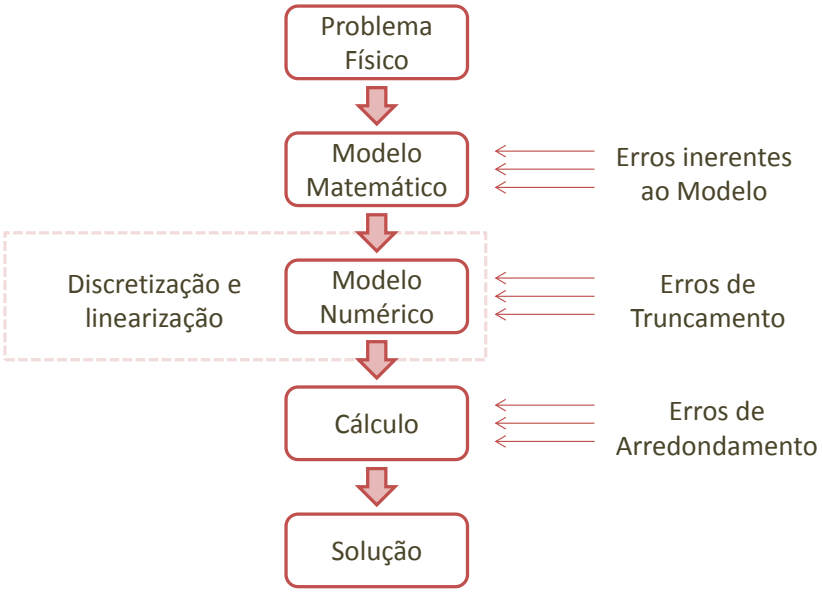


Figura 1.3. Erros nos Cálculos Numéricos.

qual pode ser de precisão dupla ou maior, antes de ser armazenado em algum lugar da memória em um formato menor do que o tamanho dos registradores [35, 36].

No exemplo 1.2 o número foi primeiro arredondado para o mais proximo (RN) 16 dígitos e depois para 9 dígitos, assim o número resultante não igual ao número arredondado diretamente para 9 dígitos.

***Exemplo* 1.2**

$$x_{real} = 7.619402384999999960000 \times 10^{-180},$$

$$x_{errado} = RN_9\left(RN_{16}\left(x_{real}\right)\right) \neq RN_9\left(x_{real}\right) = x_{correto}$$

$$\underbrace{7.61940239 \times 10^{-180}}_{x_{errado}} \neq \underbrace{7.61940238 \times 10^{-180}}_{x_{correto}}$$

□

Se bem o IEEE-754 é o formato comum para o desenvolvimento de sistemas computacionais, temos de enfrentar o erro de duplo arredondamento, quando o mesmo for usado. Isto é um problema sério devido ao fato de as matrizes serem "elementos matemáticos mal condicionados", sendo que o

uso tradicional (na prática) está restrito a sistemas onde erros relacionados ao duplo arredondamento não afetam muito a qualidade dos resultados; um problema inerente às operações matriciais de grande porte. Por este motivo, se faz necessário propor diferentes arquiteturas que consigam lidar com estes erros, mantendo baixos os parâmetros de propagação de erro, obtendo assim uma melhor precisão nos resultados.

1.4 Motivação do Trabalho

Atualmente, a maioria das soluções propostas para lidar com a complexidade computacional para se obter a solução numérica de sistemas matriciais encontra-se apenas em implementações em *software*, no entanto, há uma carência de processadores com *hardwares* dedicados capazes de aproveitarem eficientemente o paralelismo intrínseco das soluções algorítmicas, e assim acabam executando-se em processadores de propósito geral (GPPs). Uma alternativa adotada na implementação em sistemas embarcados é o uso cada vez mais frequente de soluções em *hardware* reconfigurável. Essa busca pelo *hardware* reconfigurável (como aquele suportado pelas *FPGAs*) é preferida nos casos em que a quantidade de usuários não justifica uma produção em larga escala devido aos custos de desenvolvimento, especificados por NREs (*Non-Recurring Engineering*) altos, como no caso dos ASICs (*Application-Specific Integrated Circuit*).

Por um lado, uma revisão bibliográfica mostra que são poucos os trabalhos desenvolvidos utilizando os métodos diretos onde são abordadas arquiteturas paralelas usando *hardware* reconfigurável. Se bem, o desenvolvimento de arquiteturas paralelas aproveitando técnicas de *acessos a memória*, *pipeline* e *arranjos sistólicos*, representam um desafio na implementação em *hardware*, as mesmas podem vir a permitir obter o balanço entre os principais parâmetros de desenho, citados anteriormente: *precisão*, *desempenho*,

consumo de recursos lógicos e *potência.*

Por outro lado, existe a possibilidade tecnológica atual de se propor arquiteturas baseados em fluxo de dados, onde pode ser explorado o paralelismo intrínseco das técnicas de soluções numéricas de sistemas lineares, podendo ser implementadas diretamente em *hardware* reconfigurável, e direcionadas ao projeto de soluções embarcadas.

O alto grau de paralelismo que permitem os *FPGAs* é uma caraterística que torna interessante o seu uso em diferentes campos, onde há uma necessidade pela computação de alto desempenho. Esse paralelismo intrínseco das *FPGAs* vem ganhando interesse na comunidade de computação paralela, a qual trata de sistemas com alta demanda de tarefas computacionais, como por exemplo a de obter a solução numérica de sistemas matriciais. Assim, os métodos numéricos (ver figura 1.2) vêm sendo atraentes na solução de problemas envolvendo sistemas matriciais de grande porte, e aparecem como candidatos ideais para se desenvolverem arquiteturas paralelas; como por exemplo aquelas baseados em fluxo de dados [37, 38].

Essa falta de ferramentas computacionais de baixo custo, alto desempenho e precisão, que permitam lidar e, sobretudo, acelerar a solução numérica de sistemas matriciais baseados na paralelização dos algoritmos, são os fatores que motivam a realização do presente trabalho. Neste trabalho foram estudados os algoritmos *QR* e *Método de Schur* para adaptá-los e implementá-los em um *hardware* dedicado, a fim de enfrentar essa complexidade computacional. Neste caso, três parâmetros de *design* serão considerados:

1. *Propagação do erro*: fatores que incrementem a acurácia dos resultados,

2. *Limitações em hardware*: quantidade de recursos lógicos necessários para implementar a solução de sistemas lineares, considerando que, grande porte em *hardware* (cujo tamanho pode ir à ordem de centenas e mesmo milhares de linhas e colunas) é diferente em termos de grande porte em *software* (cujo tamanho pode ir à ordem de milhões de linhas e colunas).

3. *Desempenho*: medido em termos de latência, frequência assim como a taxa de transferência entre os diferentes blocos do sistema implementado em *hardware.*

Finalmente, este trabalho pode resumir a sua hipótese fundamental na viabilidade de se propor soluções em *hardware* eficientes para acelerar o desempenho e, por sua vez, obter a solução numérica de sistemas lineares baseados nos métodos de eliminação Gaussiana, decomposição QR e de Schur. Para este fim, foram propostas arquiteturas baseadas em fluxo de dados, no formato de arranjos sistólicos [37, 38, 39, 40].

1.5 Objetivos

1.5.1 Objetivo Geral

Este trabalho propõe desenvolver arquiteturas paralelas em *hardware* reconfigurável para se obter e acelerar a solução numérica em problemas que envolvem sistemas matriciais. As arquiteturas foram baseadas no método direto *QR* e no *Método de Schur*, utilizando topologias baseadas em fluxo de dados e arranjos sistólicos. Também, foram utilizadas arquiteturas baseadas no algoritmo de Eliminação Gaussiana para encontrar a solução numérica de sistemas lineares. Para representar numericamente os dados,

foi usada uma representação em ponto flutuante. As implementações foram em arquiteturas reconfiguráveis baseadas em FPGAs, visando a sua aplicabilidade em problemas inversos de pequeno ou grande porte.

1.5.2 Objetivos Específicos

1. Projetar arquiteturas paralelas em *hardware* reconfigurável que permitam acelerar a solução numérica implementado os métodos *QR*, *Método de Schur* e *Eliminação Gaussiana*, visando o mínimo consumo de recursos lógicos, maximizando as características de acurácia.
2. Implementar uma verificação do bloco de solução de sistemas de equações lineares em FPGA mediante o uso do *hardware in the loop*.
3. Avaliar *in situ* a arquitetura projetada para solução de sistemas de equações lineares em uma FPGA.
4. Implementar uma metodologia de fusão de *datapath* que permita a redução da propagação do erro de duplo arredondamento.
5. Definir uma metodologia que permita a verificação tanto funcional (via simulação) quanto em *hardware* (na FPGA) dos resultados obtidos.
6. Desenvolver uma ferramenta de geração automática de código VHDL em MatLab como apoio à metodologia de verificação/validação.

1.6 Metodologia do Trabalho

No desenvolvimento do presente trabalho foi seguida uma metodologia descrita à continuação:

- *Revisão Bibliográfica*: Foi realizada uma revisão bibliográfica do estado da arte dos métodos de solução numérica de sistemas matriciais, considerando as implicações da sua implementação em *hardware*.

- *Análise da Propagação do Erro*: Usando MatLab como estimador estatístico foi feita uma implementação em *software* dos métodos para encontrar uma solução numérica de sistemas matriciais passando por uma análise da propagação do erro. Cada um dos métodos foi avaliado utilizando valores aleatórios como entradas.

- *Implementações em Hardware* : Implementação das arquiteturas para solução numérica de sistemas matriciais usando os métodos estudados para criar as diferentes bibliotecas para tal fim, todas usando uma representação em ponto flutuante.

- *Verificação Comportamental e Verificação em Hardware as Arquiteturas*: Para cada uma das arquiteturas foi feita uma verificação comportamental assim como uma verificação em *hardware*. Para tal fim foram utilizados o ModelSim e o SystemGenerator executando uma série de valores aleatórios como entrada do bloco solução numérica de sistemas matriciais.

Adicionalmente, neste trabalho definiu-se como *pequeno porte* sistemas lineares a partir de 4×4 até 25×25, como médio porte sistemas lineares a partir de 26×26 até 75×75 e como grande porte sistemas lineares a partir de 76×76. Embora estas definições possam ser arbitrárias, as mesmas foram dadas tendo em conta tamanhos de FPGAs disponíveis no mercado, assim como dimensões de matrizes reportadas na literatura referente à implementação em *hardware* de soluções de sistemas lineares.

1.7 Resultados Alcançados

Durante o desenvolvimento deste trabalho vem se gerando um conjunto de publicações listadas no Apêndice A, as quais são descritas neste documento, além de alguns resultados por serem publicados. A seguir, são descritos os principais resultados alcançados:

1. *Implementação em hardware da arquitetura para solução da função inversão de matriz*: Neste documento são apresentados os resultados obtidos na implementação em *hardware* reconfigurável FPGA da arquitetura para calcular a inversa de uma matriz usando representação em ponto flutuante com três diferentes precisões: 32, 64 e 40 *bits*. A descrição em *hardware* da arquitetura é baseada no algoritmo de Eliminação Gaussiana (e sua variação Gauss-Jordan), criando como sua principal unidade o circuito *Gauss-Jordan elimination*. Esse componente contem outros processadores especializados para lidar com as diferentes tarefas ou partes do algoritmo, sendo estes cinco, quatro módulos e uma unidade: *Change Row Module*, *Pivo Module*, *Matrix Elimination Module*, *Normalization Module* e por fim *the Gauss-Jordan Control-Circuit Unit*. Essa divisão permitiu trabalhar com outras pequenas unidades aritméticas, organizadas para manter a precisão dos resultados sem a necessidade de internamente normalizar ou desnormalizar os dados em ponto flutuante. A implementação de toda a arquitetura toma vantagem dos recursos *hardware* disponíveis na FPGA Virtex-5. Os resultados de desempenho e consumo de recursos da implementação são melhorados quando comparados com outras arquiteturas onde a sua implementação torna-se mais complexa especialmente para implementação de baixo custo. Alguns pontos de referência foram feitos com as soluções implementadas previamente no FPGA e no *software*, tais como MatLab. A propagação do erro

assim como o consumo de recursos na implementação, especialmente os blocos de memórias RAM internas que são usados, constituem melhoras quando comparadas com trabalhos prévios.

2. *Implementação em hardware de uma arquitetura para encontrar a solução numérica de um sistema de equações lineares*: Também, é mostrado o resultado da implementação de uma arquitetura de baixo custo para resolver sistemas lineares de equações baseado no método de eliminação Gaussiana usando um sistema reconfigurável baseado em FPGA. A arquitetura pode lidar com uma precisão simples usando o padrão em ponto flutuante IEEE 754. A implementação toma vantagem das memórias internas assim como dos DSPs disponíveis na Virtex−5. A arquitetura proposta é composta de quatro módulos e uma unidade : *Change Row Module*, *Pivo Module*, *Matrix Elimination Module*, *Normalization Module* e por fim *the Gaussian-Control-Circuit Unit*. Igualmente, uma unidade especial de acesso a memória foi desenvolvida para lidar como as operações de escrita/leitura desde/para a memória interna RAM.

3. *Implementação de uma verificação do bloco de solução de sistemas de equações lineares em FPGA chamada de HIL (hardware in the loop)*: Este trabalho introduz o fluxo de simulação baseado no uso da ferramenta XSG (*Xilinx System Generator*), de uma arquitetura para resolver sistemas lineares com matrizes densas presentes em sistemas fortemente ligados. Um processo de verificação funcional foi alcançado tomando vantagem do XSG, permitindo ambas as simulações *software* e *hardware in the loop* (HIL) usando os resultados respectivos do modelo de referencia em MatLab. O bloco SSL no XSG pode lidar com precisões simples, duplas e customizadas seguindo o padrão em ponto flutuante IEEE 754.

4. *Aplicação da arquitetura solução de sistemas de equações lineares*: Os

resultados obtidos de uma aplicação do bloco de solução de sistema de equações lineares (SSL) apoiando os cálculos de uma rede neural tipo GMDH e seu uso para predizer de forma aproximada a estrutura tridimensional das proteínas são apresentados neste documento. O bloco SSL foi usado especificamente para acelerar o método dos mínimos quadrados. Ambos o *hardware* GMDH e o *hardware* SSL foram escritos em VHDL e provados no dispositivo FPGA.

5. *Implementação de operadores aritméticos em hardware*: entre outros resultados encontra-se o desenvolvimento de arquiteturas de *hardware* parametrizáveis para as operadores de soma/subtração e multiplicação usando uma representação aritmética de ponto flutuante com 32 e 64 bits de precisão. Estas arquiteturas foram criadas como arquiteturas de fluxo de dados. Além disso, foi desenvolvida uma arquitetura capaz de realizar em um único bloco a operação conjunta de multiplicação soma acumulação em ponto flutuante.

   Todas as arquiteturas aritméticas foram simuladas, implementadas e validadas usando o MatLab como estimador, seguindo uma metodologia de verificação funcional, de verificação em *hardware* e de validação da arquitetura. Uma metodologia de fusão de *datapath* foi seguida e explorada para desenvolver e implementar a arquitetura de fusão de multiplicação soma acumulação. Os resultados mostraram reduções em uso de recursos DSPs e do erro associado aos cálculos.

6. *Implementação em hardware reconfigurável FPGA da função raiz quadrada de uma matriz*: Como um dos resultados principais deste trabalho se encontra uma nova implementação em *hardware* reconfigurável FPGA do método de Schur para calcular a função raiz quadrada de uma matriz simétrica e definida positiva. Esta arquitetura usa uma representação em ponto flutuante de 32 bits de precisão baseada no padrão IEEE-754. Para lidar com todos os processos

de cálculo internos foram usadas estruturas em forma de arranjos sistólicos baseadas em três blocos matriciais principais: QRD-MGS, MMM e MDTM. A implementação desta arquitetura alcançou resultados de desempenho de 10.41 GFLOPs e de propagação do erro de 23.0494/7.5706 para cada caso numérico de estudo, quando embarcada na plataforma Kintex7-XC7K325T, utilizando paralelização e *pipeline* das tarefas.

Adicionalmente, cabe salientar três aspectos metodológicos usados no desenvolvimento deste trabalho:

1. O primeiro foi a implementação da metodologia de fusão de *datapath* para conseguir descrever uma operação de multiplicação soma e acumulação em um único bloco *hardware* , evitando erros de duplo arredondamento. Esta metodologia pode ser usada no desenvolvimento de processos com grandes cadeias aninhadas de operações aritméticas em ponto flutuante, o que significa uma redução significativa não apenas do erro associado nos cálculos mas também, uma redução dos recursos lógicos utilizados uma vez que são em menor número as etapas de denormalização/normalização de dados realizada.

2. No segundo foi implementada uma metodologia de verificação validação de arquiteturas. Esta metodologia ajudou no processo de verificação padronizando cada um dos experimentos e permitindo uma comparação dos resultados de forma normalizada. A metodologia permitiu realizar uma verificação funcional e verificação em *hardware* assim como também realizar a validação das arquiteturas referentes a parâmetros de desenho previamente estabelecidos.

3. Por fim, pode-se citar o seguinte resultado paralelo ao desenvolvimento das arquiteturas. Onde, como parte e apoio à metodologia de

verificação validação foi desenvolvida uma ferramenta de geração automática de código VHDL no Matlab, facilitando a implementação em *hardware* das diferentes arquiteturas propostas para operações matriciais assim como para a geração das operações aritméticas. Alguns parâmetros de desenho podem ser configurados usando este gerador de código, tais como o tamanho do sistema (ou da matriz) e a precisão numérica do formato em ponto flutuante utilizado. Isto facilita a sua implementação em FPGAs e acelera o tempo de desenvolvimento em *hardware*.

1.8 Organização do Trabalho

O presente trabalho está organizado da seguinte forma:

No capítulo 2 é feita uma breve introdução ao *hardware* reconfigurável, explicando a arquitetura von Neumann, os FPGAs e alguns aspectos relevantes da tecnologia usada. Também é apresentada a representação numérica utilizada neste trabalho assim como uma seção é usada para descrever alguns métodos para solução numérica de sistemas matriciais. Onde alguns fundamentos e conceitos básicos da álgebra linear, definindo os conceitos de métodos de decomposição e/ou fatoração e de eliminação assim como a função raiz quadrada de uma matriz.

O capítulo 3 descreve as implementações realizadas para calcular a solução de um sistema de equações lineares assim como para calcular a inversa de uma matriz. São apresentados neste capítulo, os resultados obtidos assim como sua análise. O capítulo 4 apresenta os resultados obtidos da implementação das arquiteturas em fluxo de dados, mostrando as estruturas propostas, as metodologias implementadas assim com os diferentes circuitos desenvolvidos para, no fim, obter uma arquitetura capaz de encontrar

a solução numérica da função raiz quadrada de uma matriz. O caítulo 5 apresenta a discussão dos resultados e as contribuições deste trabalho. Finalmente, o capítulo 6 apresenta as conclusões e perspectivas de trabalhos futuros na área de desenvolvimento deste trabalho.

Capítulo 2 ASPECTOS TEÓRICOS

Neste capítulo serão descritos alguns temas vinculados ao tema de *hardware* reconfigurável, o qual foi utilizado como plataforma de desenvolvimento e prototipação das arquiteturas propostas, assim como alguns tópicos sobre a representação e/ou formato numérico utilizado. A seguir, uma seção é dedicada aos métodos utilizados para obter soluções numéricas de alguns sistemas tomados como estudo de caso, envolvendo métodos matriciais, tais como decomposições e processos de eliminação. Nesta direção é mostrado o uso da decomposição visando a obtenção da raiz quadrada de uma matriz simétrica (um dos temas a serem abordados neste trabalho). Adicionalmente, alguns conceitos e definições básicas da álgebra linear serão citados, a fim de fornecer os principais fundamentos necessários para a compreensão das propostas deste trabalho. No apêndice B são fornecidos maiores detalhes relacionados com técnicas e conceitos de álgebra linear, importantes para a compreensão dos algoritmos alvo deste trabalho.

2.1 *Hardware* Reconfigurável

2.1.1 Arquitetura von Neumann

John von Neumann foi um matemático húngaro naturalizado estadunidense quem formalizou uma arquitetura para cálculo computacional de propósito geral junto com o *Arthur Walter Burks* e o *Herman Heine Goldstine*, ambos também matemáticos. Os mesmos definiram que a máquina devia conter certos dispositivos principais, tais como: (a) ULA (Unidade

Lógica e Aritmética), (b) memória de armazenamento, (c) unidade de controle e (d) conexões com o exterior (sistemas de entrada/saída) assim como também a forma de como os programas e dados compartilhariam o espaço da memória de armazenamento. Com isso, *von Neumann* sedimentou os conceitos de um modelo computacional [41, 42].

Esta arquitetura, conhecida atualmente como Modelo de *von Neumann*, representa um paradigma de computação, baseado no princípio de "programa armazenado em memória", o qual permite armazenar no mesmo espaço de memória tanto as instruções quanto dados. As instruções são numericamente codificadas de tal forma que a máquina pode distinguir entre os dados e as instruções. O modelo de *von Neumann* propõe um circuito capaz de executar as instruções armazenadas na memória, denominado *unidade de controle*. Desta maneira, o modelo de *von Neumann* é orientado à execução de instruções (fluxo de instruções), contendo para isto um registro específico objetivando o armazenamento da próxima instrução a ser executada (o *contador de programa* - PC).

Adicionalmente, o modelo define um dispositivo capaz de realizar operações aritméticas, tais como: soma, subtração, multiplicação e divisão, o qual pode ser estendido para operações mais complexas (ver figura 2.1), a ULA (*Unidade Lógica-Aritmética*). De maneira geral, o modelo apresenta uma via de dados (*data-path*) e uma via de controle, especificadas na figura (2.1), conectadas através de barramentos de comunicação. O mesmo sistema de comunicação baseado em barramentos é usado para interconectar toda a CPU e a memória do sistema [41]. Desta forma, o barramento de dados se comporta como um canal de comunicação com vazão de dados (*througput*) limitada pelas próprias características físicas do canal.

Na maioria dos computadores modernos as CPUs conseguem atingir tempos de processamento altos quando comparadas com o fluxo de informação

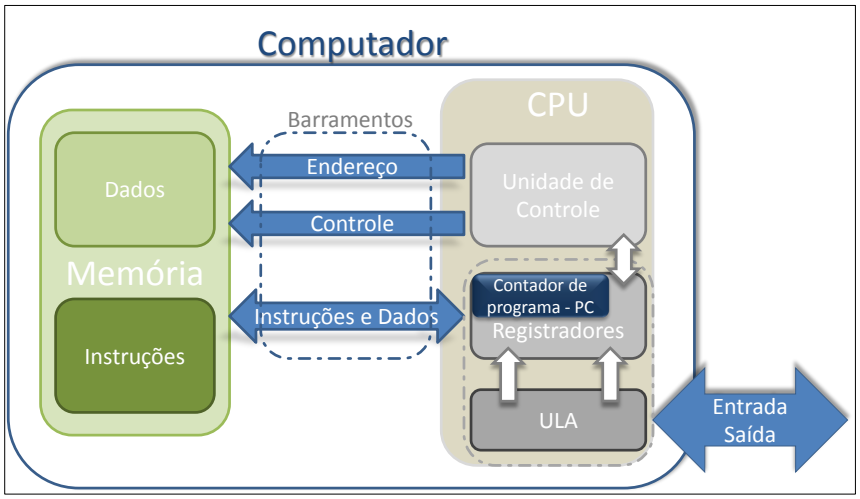


Figura 2.1. Arquitetura *von Neumann*

desde/para a memória RAM, obrigando ao mesmo fazer paradas para tarefas de sincronização de transferência de dados. Esta diferença em desempenho entre os processadores e as memórias, conhecida como "*memory wall*" [43], é um dos maiores problemas na computação de alto desempenho. As limitações supracitadas referentes às limitações físicas da comunicação entre memória/processador e "*memory wall*" sumarizam o denominado "*gargalo de von Neumann*", sendo este um dos maiores inconvenientes das arquiteturas convencionais [44].

Neste contexto, a indústria de processadores, vem sendo sustentada durante muitos anos pelas previsões feitas pela lei de Moore. No transcurso dos anos a lei de Moore permitiu melhorar o desempenho das arquiteturas "*von Neumann*", pelo incremento da frequência do *clock* do sistema, a cada certo período de tempo, gerando como efeito colateral um aumento no consumo de potência do chip (o qual é proporcional à $carga_{capacitiva} \times Voltagem^2 \times freq_{clock}$, ver [42]), o qual chegou a ser no começo do milênio um problema crítico no projeto de processadores (*the power wall*, vide capítulo 1 de [42]). Desta forma, ao em vez de se continuar aumentando o desempenho do *software* executado em um único processador pelo incremento da frequência do relógio, a solução adotada

pela indústria tem sido a implementação de plataformas multinúcleo, como o *Intel Xeon 4 cores* ou o *Intel Core i7 4 cores*.

2.1.2 *Field Programmable Gate Arrays*

Os *FPGAs* foram desenvolvidas pela Xilinx em 1984 representando estruturas regulares em formas de arranjos de portas/memórias (ou elementos lógicos). Uma arquitetura de um FPGA consiste em um conjunto de arranjos de *CLBs*(blocos lógicos configuráveis), blocos I/O configuráveis, assim como interconexões programáveis. Entre os recursos lógicos podem incluir *ULAs*, elementos/blocos de memória e decodificadores, dispositivos de processamento digital *DSP*, entre outros. Existem três diferentes tipos de elementos de programação para uma *FPGA*: a *RAM estática*, o *anti-fusível*, e a *flash EPROM*, onde podem ser interconectados de forma arbitrária por chaves programáveis para formar as redes de conexões entre as células.

Uma arquitetura genérica *FPGA* é mostrada na figura 2.2, onde, pode-se observar como é constituída internamente. Na figura 2.2 são mostrados os *slices* junto com a matriz de chaveamento, ambas sendo parte das colunas internas de recursos digitais disponíveis no FPGA.

FPGAs permitem implementar, em princípio, qualquer tipo de circuito digital, o qual permite que empresas especializadas no projeto de circuitos integrados utilizem FPGAs no seu fluxo de desenvolvimento, incluindo a prototipação e verificação [46]. Embora estes dispositivos apresentam uma versatilidade, os mesmos apresentam desvantagens em termos de desempenho, área, consumo de potência e tempo de configuração [47]. *FPGAs* podem ser usados em qualquer fluxo de projeto digital, proporcionando benefícios em termos de possibilidade de paralelizar algoritmos com tempos

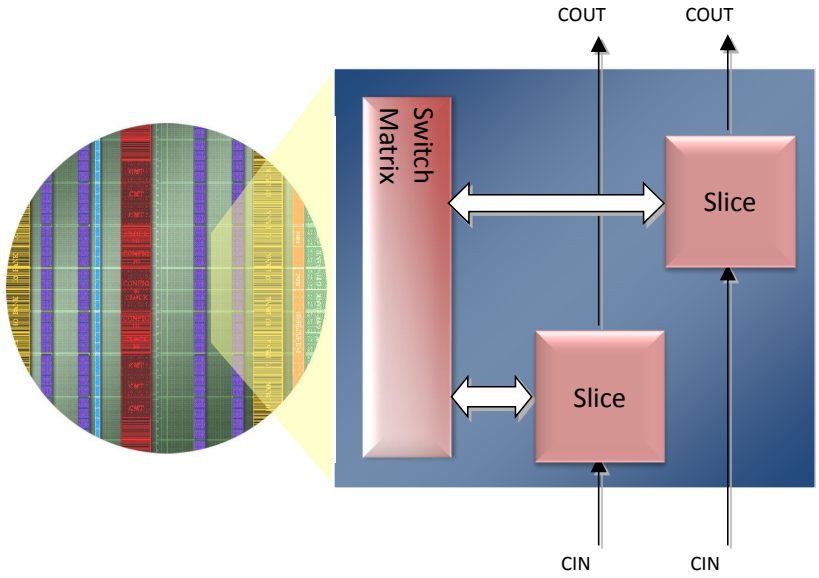


Figura 2.2. Arquitetura do FPGA (modificado de [45]).

baixos de projeto NRE (*Non-recurring engineering*), sem os altos custos do desenvolvimento personalizado dos ASICs (*Application Specific Integrated Circuit*). No caso dos FPGAs, muitos dos recursos são usados para tarefas de configuração, fazendo os mesmos deficientes em termos de velocidade e consumo de potência, se comparados com ASICs (ver Tabela 2.1).

Com a redução continua das características em tamanho dos processos de tecnologia em semicondutores, o custo de um conjunto completo de máscaras de litografia passou de mais de US1,5 milhões (para tecnologia de 90nm) para US2 milhões (para tecnologia de 65nm) [48]. Essa complexidade (em termos de número de transistores por chip) dos grandes projetos também aumenta o número *re-spins* necessários antes da produção, devido aos erros produzidos em etapas de projeto, por exemplo, problemas na verificação. Esses quesitos elevam consideravelmente os custos de NRE nos projetos baseados em ASICs, para aplicações com baixos (ou médios) volumes de produção [48]. Como alternativa FPGAs podem ser utilizados, mas os mesmos consumem mais potência e apresentam menor desempenho do que os ASICs (ver Tabela 2.1).

Neste sentido, em um ponto intermediário entre os FPGAs e os ASICs situam-se os chamados *Structured ASIC*. Esses dispositivos são geralmente usados para volumes de aplicações médias e apresentam uma combinação do melhor das duas tecnologias. Um *Structured ASIC* consiste de um padrão regular e repetitivo de estruturas combinacionais e sequenciais, com uma variedade de elementos bem estruturados e otimizados através do chip inteiro. Desta forma, com algumas alterações em umas poucas máscaras, é possível modificar as conexões entre os elementos estruturados, projetando assim as funções desejadas para gerar o circuito. Adicionalmente, um projeto prototipado em um FPGA pode ser facilmente direcionado para structure-ASICs, tendo ganhos automáticos em desempenho. Esta é uma vantagem devido a que permite que vários projetos compartilhem a maioria das máscaras, reduzindo assim os custos de NRE e *time-to-market* [49]. Contudo, os FPGAs continuam sendo a tecnologia usada para aplicações com baixa demanda devido a sua reusabilidade e flexibilidade sem gerar grandes custos de NRE, permitindo um rápido *time-to-market*. A Tabela 2.1 apresenta a comparação entre FPGAs, ASICs e os *Structured ASIC*.

Tabela 2.1. Comparação entre as tecnologias: FPGA vs Structured ASIC vs ASIC

	FPGA	Structured ASIC	ASIC
Processo de desenho	Simples	Simples	Longo e complexo
NRE (*Non-recurring engineering*)	Não apresenta	Baixos custos	Altos custos
time-to-market	Rápido	Curto	Longo
Reusável e flexível	Sim	Não	Não
Desperdício de *hardware*	Sim	Não	Não
Ideal para aplicações com demanda	Pouca	Média	Alta
Tamanho	Grande	Pequeno	Pequeno
Consumo de Potência	Alto	Baixo	Baixo
Desempenho	MHz	Entre MHz e GHz	GHz

A arquitetura do FPGA é organizada por CLBs, os quais possuem uma estrutura especifica dada por cada fabricante e para cada família de dispositivos. Neste caso a família 7 da Xilinx apresenta uma arquitetura típica

tal como mostrado na figura 2.3. Cada CLB contem dois *slices* e sua matriz de chaveamento o que permite, mediante interconexões, construir arranjos maiores (ver figura 2.3).

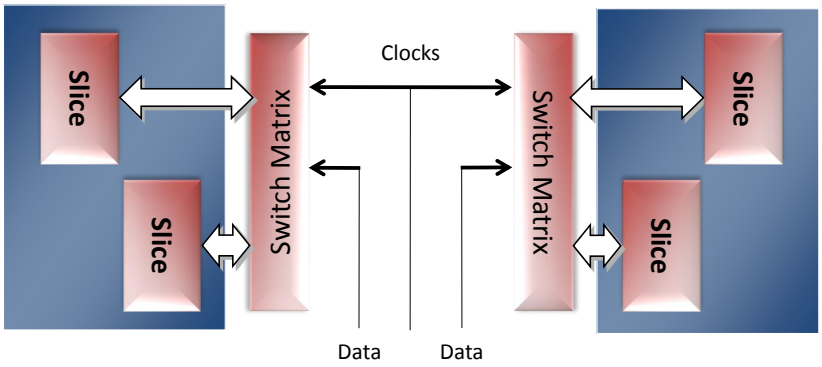


Figura 2.3. Arquitetura do CLB e seus recursos: *slices*, matriz de chaveamento e interconexões (modificado de [45]).

Os *slices* são elementos presentes no CLB, contendo dentro da sua arquitetura LUTs (*Look up Table*). Estas LUTs permitem implementar funções Booleanas, dependendo do número de entradas necessárias onde, a implementação é feita mediante o uso de multiplexadores e flip-flops presentes na própria arquitetura (ver figura 2.4). Quando o número de entradas disponíveis da LUT é menor do que o número necessário, uniões de LUT são feitas incrementando assim a complexidade do arranjo tal como mostrado na figura 2.5. Adicionalmente, algumas LUT podem ser configuradas também como registradores de deslocamento (SRL).

A metodologia de projeto para implementar uma arquitetura em um *FPGA* é mostrada na figura 2.6. A mesma figura inclui as etapas necessárias para esta tarefa: (a) especificações do desenho, (b) a síntese do desenho/projeto, (c) implementação e (d) programação do dispositivo.

O processo de síntese do projeto verifica o código e analisa a hierarquia do circuito na etapa inicial, fornecendo o esquemático RTL (do inglês *Register Transfer Level*) e o mapeamento tecnológico. Esta etapa garante que o projeto citado será otimizado para a arquitetura projetada. As conexões são

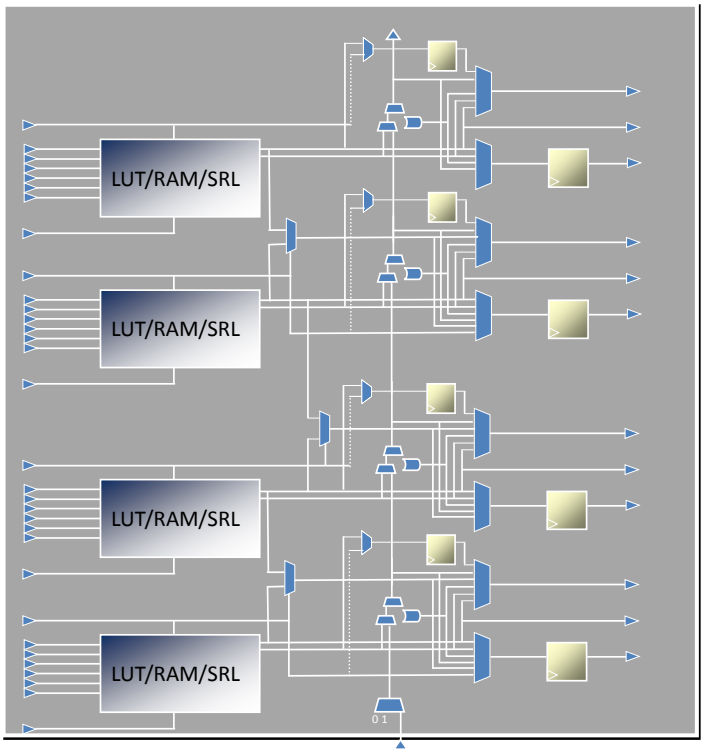


Figura 2.4. Arquitetura do *slice* e seus recursos: LUT/RAM/SRL, multiplexadores, Flip-Flops e interconexões (modificado de [45])

criadas como uma *netlist*, sendo armazenadas como um arquivo *NGC* (no caso da Xilinx Synthesis Technology (XST)) ou como um arquivo *EDIF* (no caso de LeonardoSpectrum, Precision, ou Symplify/Symplify Pro).

O fluxo do projeto em FPGA segue os seguintes passos após a síntese lógica: (a) mapeamento tecnológico, (b) posicionamento e roteamento e (c) geração do arquivo de programação. A fase de *mapeamento tecnológico* associa elementos do *netlist* aos recursos disponíveis no dispositivo definido pelo usuário. A fase *posicionamento e roteamento* consiste na definição da localização física dos blocos lógicos no FPGA e na realização das interconexões entre eles e a geração do arquivo de programação *bitstream* para configuração do FPGA.

Como parte importante e relevante no desenvolvimento do projeto, este deve passar por várias etapas de verificação. A verificação começa com as simulações comportamentais do projeto onde é analisada a lógica do

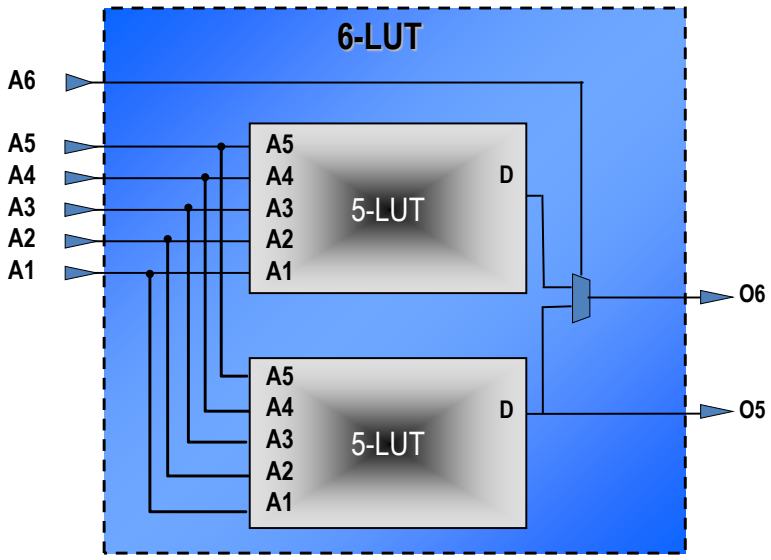


Figura 2.5. Arquitetura de uma LUT de 6 entradas formada apartir de duas LUT de 5 entradas (modificado de [45])

mesmo baseado na proposta inicial do projetista. Já a verificação funcional certifica a funcionalidade do projeto em diferentes pontos no fluxo do desenho com simulações comportamentais (antes da síntese), simulações funcionais (depois da síntese) e/ou verificações no circuito (depois da programação do dispositivo). Essas verificações no circuito, no momento da implementação, também contemplam a análise temporal estática do circuito, assim como as simulações temporais onde são inseridos os valores ou bibliotecas com as informações dos atrasos dos circuitos projetados.

A verificação no circuito faz uma análise da sincronização do projeto em diferentes pontos no fluxo do mesmo com sincronização estática (depois das etapas de posicionamento e roteamento) e simulação de sincronização (depois das etapas de posicionamento e roteamento). Essa verificação permite encontrar erros não contemplados durante o projeto, tais como os gerados pelo próprio ambiente de projeto.

No contexto do projeto de sistemas digitais, FPGAs permitem implementar soluções baseadas em fluxo de dados, desenvolvendo *hardware* específicos para cada algoritmo, pelo mapeamento diretamente em *hardware* do

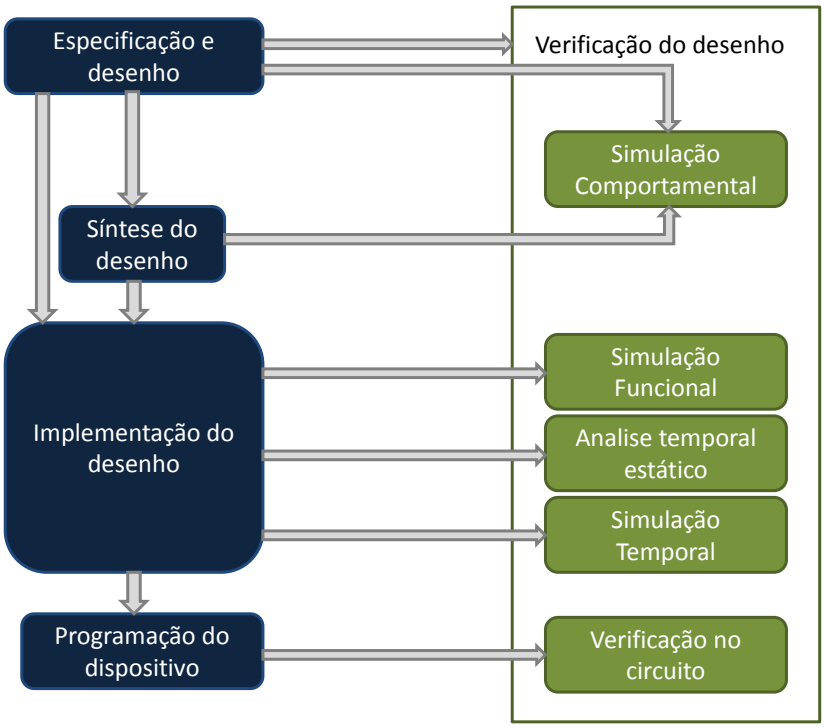


Figura 2.6. Etapas de um projeto com FPGAs da Xilinx

algoritmo. Neste caso, o fluxo de instruções típico do modelo de *von Neumann* pode vir a se converter em fluxo de dados, evitando o uso de etapas de leitura/escrita de dados desde/para a memória. Desta maneira, propostas de arquiteturas baseadas em fluxo de dados podem ser facilmente prototipadas neste tipo de dispositivo *floware* e *configware*, tal como propostas em [50]. FPGAs de granularidade fina fornecem uma grade de grão fino de unidades funcionais de *bit-wise*, que podem ser compostas para criar qualquer circuito desejado ou mesmo um processador. A maioria da área nas *FPGAs* é realmente dedicada à infra-estrutura de roteamento o qual permite que as unidades funcionais sejam interligadas em tempo de configuração.

FPGAs de granularidade grossa fornecem um número de unidades funcionais dedicadas, como blocos de DSP contendo multiplicadores, e blocos de RAM, diminuindo a configuração na memória e os tempos de configuração assim como a complexidade no problema de *posicionamento e roteamento* [51]. Atualmente, as FPGAs contêm centenas de DSPs e blocos RAM

compartilhando a infra-estrutura de roteamento permitindo aumentar a complexidade dos projetos.

O mercado dos DSP inclui aplicações que abrangem um amplo espectro de desempenho e de requisitos de custos, tais como: 3G, voz sobre protocolo de internet (VoIP), sistemas multimídia, radar e sistemas satelitais, sistemas médicos, aplicações em processamento de imagens e eletrônica de consumo.

Processadores especializados do tipo DSP podem implementar muitas dessas aplicações. Embora, esses processadores sejam programáveis mediante *software*, a sua arquitetura em *hardware* não é flexível. Neste sentido, os DSPs são arquiteturas de *von Neumann* com via de dados (datapaths) sofisticados, envolvendo conjunto de ULAs, acumuladores, etc,. Portanto, uma arquitetura de *hardware* fixo, tal como um DSP, apresenta gargalos relacionados ao número de blocos multiplicadores-acumuladores (MAC), ao tipo e tamanho de memória fixa e ao tamanho da largura de dados, a qual limita o desempenho deste tipo de arquiteturas. Desta maneira, os processadores do tipo DSP não são adequados para algumas implementações que requerem implementações de funções DSP customizadas (ver Tabela 2.2)[52].

Por outro lado, os FPGAs fornecem soluções para implementar aplicações para processamento digital de sinais, devido à possibilidade de mapear diretamente as equações dos filtros em estruturas customizadas de *hardware*. Neste contexto, os fabricantes fornecem ferramentas de projeto a fim de acelerar o projeto do tipo DSPs em FPGAs por exemplo: *System Generator* da Xilinx o qual usa MatLab *SimuLink* como interface. A cada nova geração tecnológica os FPGAs proporcionam mais recursos lógicos e de memória, sendo este aumento da capacidade o quem vem permitindo, recentemente, o mapeamento de aplicações HPC (*high-performance com-*

puting) para FPGAs.

No contexto de possibilidades de acelerar o desempenho de aplicações HPC, os GPUs (*Graphics Processing Unit*) têm crescido consideravelmente em programação, assim como em desempenho (de operações em ponto flutuante) e vem sendo usados na aceleração de *software*. As GPUs oferecem um grande poder de processamento baseados em milhares de núcleos para processar cargas de trabalho paralelas de forma eficiente, acelerando assim as aplicações de HPC enquanto são utilizadas junto com uma CPU. Isto é feito mediante a transferência de partes do processamento intensivo dos aplicativos executados na CPU para a GPU, a fim de acelerar cálculos computacionais intensivos, enquanto o resto do código continua sendo executado pela CPU [53]. Além disso, as GPUs permitem um desenvolvimento de processo simples devido a seu *hardware* de proposito geral e de fácil programação quando comparado ao dos FPGAs 2.2.

Embora, as GPUs modernas oferecem picos muito altos de desempenho em ponto flutuante, a verdadeira força dos FPGAs está no mapeamento em *hardware* de algoritmos específicos onde podem ser extraídos picos de desempenho altos (ver Tabela 2.2).

Tabela 2.2. Comparação entre as tecnologias: FPGA vs GPU vs DSP

	FPGA	GPU	DSP
Processo de desenho	Simples	Simples	Simples
NRE (*Non-recurring engineering*)	Não apresenta	Não apresenta	Não apresenta
time-to-market	O maior entre os três	Rápido	Rápido
Reusável	Sim	Sim	Sim
Hardware flexível	Sim	Não	Não
Desperdício de *hardware*	Sim	Sim	Sim
Consumo de potência	Médio	Alto	Baixo
Desempenho	MHz	GHz	MHz

2.1.3 Arquiteturas Baseadas em Fluxo de Dados Inspiradas em Arranjos Sistólicos

O termo, *arranjo sistólico*, designa uma classe especial de arquiteturas computacionais paralelas, capazes de usar um número muito grande de processadores (de forma simultânea) para cálculos em aplicações como computação científica e processamento de sinais. Este termo foi introduzido em 1978 pelo *Kung* e *Leirserson*, formalizando o conhecimento em torno dessa classe de computadores onde algumas técnicas de arquiteturas sistólicas já eram usadas pelos projetistas de arquiteturas [54].

Um sistema sistólico é uma rede de processadores que ritmicamente calculam e passam dados através do sistema. Desta forma, como analogia com o sistema vascular humano, em um sistema computacional sistólico a função do processador é análoga à do coração, bombeando regularmente dados enquanto desenvolve pequenos cálculos computacionais. Assim, um fluxo de dados é mantido constante dentro do sistema onde todos os PE (elemento de processamento) têm uma tarefa especifica, recebendo e entregando dados de forma síncrona (ver figura 2.7) [55].

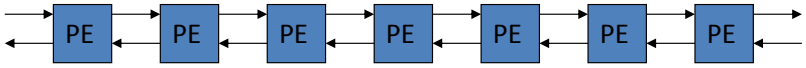


Figura 2.7. Arquitetura de fluxo (arranjos sistólicos)

Muitos cálculos matriciais básicos podem ser acelerados em um *pipeline*, de forma eficiente e elegante, utilizando redes com estruturas de arranjos sistólicos [56, 57, 58, 59]. O poder por trás dos arranjos sistólicos vem do modo como os dados fluem entre os elementos de processamento e, dado o aumento recente da capacidade dos FPGAs, é possível o mapeamento de aplicações HPC mediante o uso desta classe de arquiteturas, capazes de realizar operações matriciais tais como multiplicação de matrizes ou inversão. Desta maneira, são desenvolvidos circuitos para uso principal-

mente em máquinas com finalidade dedicada e não em computadores do tipo GPP (ver figura 2.8).

Desta forma, a figura 2.8 mostra como os elementos de duas matrizes atravessam os PE, de forma ordenada e síncrona, obtendo como resultado final a matriz C produto da multiplicação de duas matrizes A e B tal como descrito pela equação 2.1.

$$C_{44} = A_{44} \times B_{44} \tag{2.1}$$

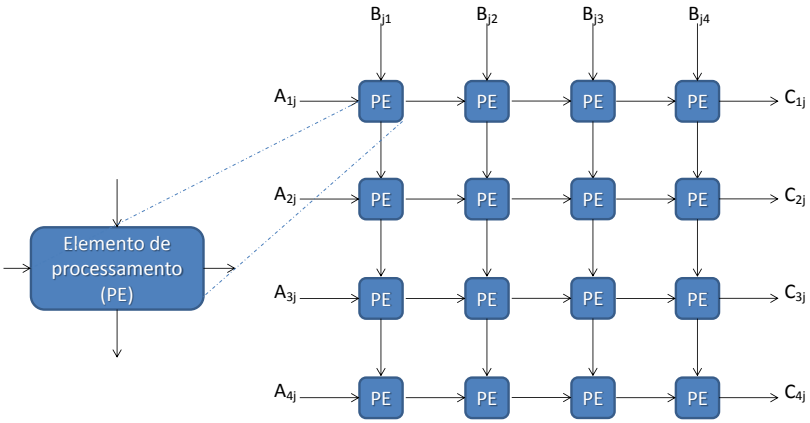


Figura 2.8. Exemplo de uma arquitetura de fluxo (arranjos sistólicos) para processar uma multiplicação matricial

Em 1995, Rainer Kress generalizou o modelo de arranjos sistólicos, permitindo o desenvolvimento de uma metodologia que suporta toda classe de esquemas irregulares destes modelos [60]. Desta forma, um grande número de modelos de arranjos sistólicos foram desenvolvidos e vem sendo utilizados para executar cálculos em diferentes aplicações. Entre essas aplicações podem se citar algumas em *processamento de sinais e imagens* como: filtros digitais, convolução e correlação, transformadas discretas de *Fourier* ou transformada rápida de *Fourier* (FFT), decodificação ou codificação para compressão, etc, [61, 62, 63, 64]. Também, algumas aplicações não numéricas como *pattern matching* [65] ou reconhecimento regular

de linguagem [66], etc. Isto demostra a disseminação destes modelos de computação (especialmente em *hardware reconfigurável*) como soluções a problemas computacionalmente exigentes.

Por um lado, o maior problema relacionado ao desenvolvimento de cálculos matriciais em sistemas computacionais, além do erro associado ao método numérico, está na natureza da tarefa computacional ser classificada de *compute-bound* (ou simplesmente *CPU bound*). As operações matriciais, em sua maioria, exigem um processamento maior do processador do que o processamento do número de entradas e saídas, fazendo assim com que o tempo total de processamento seja muito maior [67, 68].

Por outro lado, a HPC é conhecida como a área de computação onde o desempenho da máquina nunca é suficiente. Devido a isto, com o uso deste *hardware reconfigurável* e o crescente interesse pelo mapeamento da computação de alto desempenho em essa tecnologia, foi acunhado outro novo termo HPRC (*High Performance Reconfigurable Computing*), sendo esta uma área relativamente nova. Mas, um problema ainda a ser resolvido aqui é a produtividade desses programadores e o desafio tecnológico assim como educacional que envolve o seu uso em HPC [69]. Contudo, faz-se necessário uma revolução no pensamento e na educação para obter estas ferramentas assim como programadores qualificados que desenvolvam eficazmente as arquiteturas propostas pelos algoritmos [60].

Neste sentido, as arquiteturas em arranjos sistólicos visam solucionar problemas computacionais tipo *CPU bound*, visto que os dados são aproveitados ao máximo pelo arranjo sistólico antes de voltar para a memória principal do sistema. Assim, utilizando *hardware reconfigurável* para implementar arquiteturas baseados em modelos em arranjos sistólicos pode se reduzir o consumo de energia elétrica e acelerar o processamento em várias ordens de grandeza (quando comparado com o modelo de *von Neumann*)

[69]. Adicionalmente, com o avanço das ferramentas computacionais (*software* e *hardware*), projetos ou programas com arranjos sistólicos eficientes podem ser automaticamente gerados, permitindo implementações de grandes sistemas usando estas arquiteturas de forma barata, e disseminando estes modelos em *hardware reconfigurável* como soluções a problemas computacionalmente exigentes, como no caso de HPC.

2.2 Representação Numérica em Ponto Flutuante

Nesta representação, o ponto decimal é posicionado dinamicamente até ser colocado em uma posição desejada, sendo usado para tal fim o expoente o qual indica a posição original do ponto decimal. Como exemplo, tem-se o número 893.000.000.000.000.000 o qual pode ser transformado usando a notação cientifica em $8,93 \times 10^{17}$. Essa notação permite abranger um intervalo ou conjunto maior de valores usando poucos dígitos para representar diferentes ordens de grandeza/magnitude, chamado comumente de faixa dinâmica. A notação faz uso da representação representada pela equação 2.2

$$N = M \times B^E, \tag{2.2}$$

onde M representa a mantissa, B a base e E o expoente.

Essa notação é também estendida aos números binários, sendo conhecida como *representação binária em ponto flutuante.*

No intuito de padronizar a representação numérica em ponto flutuante, a IEEE propôs o padrão IEEE-754 em 1958, estabelecendo a representação

e a forma de operação dos números baseados nessa representação. Desta forma, acabando com as incompatibilidades entre sistemas, causadas pelos diferentes formatos utilizados por cada fabricante. Este fato normalmente gerava resultados diferentes quando os mesmos algoritmos eram executados em máquinas diferentes [32].

O padrão supracitado permite a portabilidade de aplicações dos programas existentes para os novos computadores que adotá-lo. Este padrão permite assim desenvolver programas numéricos sofisticados sem necessidade de se ter experiência em métodos numéricos, sendo mais eficientes, robustos e portáveis. Desta maneira, o mesmo padrão permite realizar a análise e diagnóstico de anomalias, facilitando assim a manipulação das exceções de forma apropriada a fim de desenvolver funções elementares tais como expoentes, cossenos e aritmética de alta precisão. Por fim, pode-se dizer que o padrão é de certa forma flexível (desde que sejam respeitadas as restrições impostas), permitindo realizar diferentes aperfeiçoamentos e ampliações na sua própria proposta dependendo dos requerimentos da aplicação.

Esse padrão especifica os formatos básicos e estendidos para números em ponto flutuante, as operações de soma, subtração, multiplicação, divisão, raiz quadrada, resto, assim como funções de comparação. O mesmo padrão especifica as conversões entre inteiros e formatos em ponto flutuante, assim como as conversões entre diferentes formatos em ponto flutuantes, ou entre os formatos básicos de números em ponto flutuante e cadeias de caracteres decimais (*strings*). Adicionalmente, o mesmo especifica as exceções e sua manipulação, incluindo NaN (*Not a Number*), os formatos de cadeias de bits decimais e inteiros, a interpretação do sinal e campo de bit implícito da representação de NaN e, por fim, a conversões de binário para decimal e vice-versa.

Desta maneira, o formato IEEE Std. 754 é uma representação em ponto

flutuante de um número em formato binário [32], sendo este representado por uma cadeia de bits caracterizado por três componentes principais: um bit de sinal S, um exponente E com Ew *bits* e uma mantissa M com Mw bits, como mostrado na figura 2.9 A mantissa representa a magnitude do número e uma constante *bias* é adicionada ao exponente no intuito de poder representar e trabalhar com exponentes negativos.

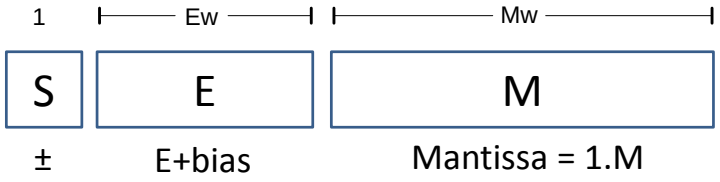


Figura 2.9. O padrão IEEE-754

O sinal S pode ter então dois valores, sendo um 0 usado para representar um número positivo e um 1 para representar um número negativo. O expoente E tem seu próprio sinal e seu valor é armazenado em excesso, seguindo a seguinte expressão matemática

$$2^{q-1} - 1,$$

onde q é o número de bits. Por exemplo, tendo um expoente com 8 bits, onde a representação em excesso seria 127 e o valor armazenado subtraindo o excesso será o verdadeiro representado no expoente. Assim, no caso de um excesso de 127 e expoente com valor igual a 9, o valor verdadeiro representado seria então dado pelo valor: $-118 = (9 - 127)$.

Por fim, a mantissa M representa um número fracionário normalizado, assim o bit mais a esquerda da mantissa é sempre 1. Devido a que o valor do bit nunca varia o seu armazenamento em memória não é requerido; embora, implicitamente este sempre faz parte da representação (sendo o bit implícito). Neste caso, a mantissa M se calcula segundo a equação 2.3

$$M = m_{22} \times 2^{-1} + m_{21} \times 2^{-2} + \ldots + m_1 \times 2^{-22} + m_0 \times 2^{-23}, \qquad (2.3)$$

e seu número real representado pela cadeia binária é obtido a partir da equação 2.4

$$N = (-1)^s \times (1 + M) \times 2^{E-Bias}. \qquad (2.4)$$

Este padrão permite ao projetista trabalhar tanto com precisão simples (32 bits) e duplo (64 bits), sendo que para precisão dupla (o diferentes de simples) os valores de mantissa, exponente e bias devem ser modificados. Por outro lado o mesmo padrão pode ser adaptado à medida dos requisitos de precisão de uma aplicação específica. Um formato de alta precisão indica assim pouco erros de quantização na implementação final, enquanto que um formato com precisão baixa indica implementações de alta velocidade e reduções em área e consumo de potência [31].

2.3 Solução Numérica de Sistemas Matriciais

2.3.1 A Decomposição

Nesta seção serão comentados três métodos de fatoração (também denominados de decomposição ou eliminação): (a) Eliminação *Gauss-Jordan*, (b) Método de Decomposição QR e (c) Método de Decomposição de Schur. Observa-se que a eliminação de *Gauss-Jordan* é geralmente usada com matrizes quadradas não singulares (ver definições no apêndice B). Por outro lado, a decomposição QR é mais geral, podendo ser aplicado para qualquer matriz quadrada ou retagular onde $m \geq n$, sendo m as linhas e n as

colunas. Para matrizes quadradas n denota o tamanho da matriz assim como $n = 4$ denota uma matriz de 4×4.

A Eliminação de Gauss-Jordan

A eliminação de Gauss-Jordan é uma versão da eliminação de Gauss que zera os elementos por cima e por baixo do elemento pivô, conforme ele percorre a matriz [70, 71]. Este método (chamado assim devido a *Carl Friedrich Gauss* e *Wilhelm Jordan*) é um algoritmo da álgebra linear para determinar as soluções de um sistema de equações lineares, assim como para obter matrizes inversas. Um sistema de equações se resolve pelo método de Gauss quando se obtêm suas soluções mediante a redução do sistema, transformando-o em outro equivalente, em que cada equação tem uma incógnita a menos do que a anterior. Quando se aplica este processo, a matriz resultante se conhece como *forma escalonada* da matriz. Em outras palavras, a eliminação de Gauss-Jordan transforma a matriz em uma *forma escalonada reduzida*, enquanto a eliminação de Gauss transforma na *forma escalonada*.

O procedimento de eliminação de Gauss-Jordan para resolução do sistema linear $Ax = b$ é descrito a seguir:

1. Formar a matriz aumentada $[A|B]$.
2. Obter a forma escalonada reduzida por linhas $[C|D]$ da matriz aumentada $[A|B]$ utilizando as operações elementares nas linhas (apêndice B).
3. Para calcular cada linha não-nula da matriz $[C|D]$ resolver a equação correspondente para a incógnita associada ao primeiro elemento não-nulo naquela linha. As linhas com todos os elementos iguais a zero

podem ser ignoradas, pois a equação correspondente será satisfeita por quaisquer valores das incógnitas.

A Decomposição QR

A decomposição QR é uma operação elementar que decompõe uma matriz A no produto de uma matriz ortogonal por uma matriz triangular superior. A decomposição QR da matriz A é assim definida como $A = Q \times R$, em que Q é uma matriz ortogonal, ou seja, $Q^T \times Q = Q \times Q^T = I$, $Q^{-1} = Q^T$ e R é uma matriz triangular superior, assim como ilustrado na equação 2.5.

$$Q = \begin{pmatrix} Q_{11} & Q_{12} & Q_{13} & Q_{14} \\ Q_{21} & Q_{22} & Q_{23} & Q_{24} \\ Q_{31} & Q_{32} & Q_{33} & Q_{34} \\ Q_{41} & Q_{42} & Q_{43} & Q_{44} \end{pmatrix}; R = \begin{pmatrix} R_{11} & R_{12} & R_{13} & R_{14} \\ 0 & R_{22} & R_{23} & R_{24} \\ 0 & 0 & R_{33} & R_{34} \\ 0 & 0 & 0 & R_{44} \end{pmatrix}. \quad (2.5)$$

Os algoritmos mais comuns para tratar a decomposição QR são: (1) *Givens rotation*, (2) *MGS orthogonalization* e (3) *Householder transformation*. Neste trabalho foi considerado para implementação em *hardware*, o algoritmo *MGS orthogonalization*, devido a complexidade computacional da implementação em *hardware* do *Householder transformation* assim como a precisão do *Givens rotation* [24].

O Gram-Schmidt Modificado:

A decomposição QR para quaisquer matriz pode ser realizada mediante o algoritmo de ortogonalização de Gram-Schmidt modificado. Este procedimento tem uma grande vantagem com relação ao algoritmo de rotações de Given, devido a que reduz o consumo de recursos de memória [24].

Neste algoritmo o termo a_{ij} denota a linha i e a coluna j da matriz A e a_i denota a coluna vetor da matriz A. A decomposição QR da matriz A, $A = QR$, pode ser obtida da forma apresentada no exemplo a seguir. Para simplificar, a matriz utilizada tem tamanho 3×3. Primeiro, a primeira coluna vetor a_i é normalizada para obter

$$r_{11} = \|a_1\|_2 = \sqrt{(a_{11})^2 + (a_{21})^2 + (a_{31})^2},$$

a qual representa os elementos na linha 1 e coluna 1 de R. Os valores de q_1 (que é a primeira coluna vetor de Q), pode ser calculada de r_{11}.

$$q_{11} = \frac{a_{11}}{r_{11}}, q_{21} = \frac{a_{21}}{r_{11}}, q_{31} = \frac{a_{31}}{r_{11}}. \tag{2.6}$$

Segundo, os valores de r_{12} e r_{13} podem ser calculados usando o coluna vetor 1 da matriz Q e a segunda e terceira coluna vetor da matriz A

$$\begin{aligned} r_{12} &= q_1^T a_2 = q_{11}a_{12} + q_{21}a_{22} + q_{31}a_{32} \\ r_{13} &= q_1^T a_3 = q_{11}a_{13} + q_{21}a_{23} + q_{31}a_{33}. \end{aligned} \tag{2.7}$$

Depois de obter os valores de q_1, r_{12} e r_{13} das equações 2.6 e 2.7, a matriz A pode ser transformada na matriz A^{p1}

$$\begin{aligned} a_{11}^{p1} = 0, a_{11}^{p1} = a_{12} - r_{12}q_{11}, a_{13}^{p1} = a_{13} - r_{13}q_{11} \\ a_{21}^{p1} = 0, a_{22}^{p1} = a_{22} - r_{12}q_{21}, a_{23}^{p1} = a_{23} - r_{13}q_{21} \\ a_{31}^{p1} = 0, a_{32}^{p1} = a_{32} - r_{12}q_{31}, a_{33}^{p1} = a_{33} - r_{13}q_{31}. \end{aligned} \tag{2.8}$$

Assim, os valores de q_1, r_{11}, r_{12} e r_{13} e a matriz A^{p1} (ver equação 2.8) são derivados dos passos anteriores. Para obter os valores restantes os passos anteriores são repetidos com A^{p1} para calcular q_2, r_{22}, r_{23} e a matriz A^{p2}. A segunda coluna vetor da matriz A^{p1} é normalizada para obter o valor de r_{22}. Exemplo

$$r_{22} = \left\| a_2^{p1} \right\|_2 = \sqrt{(a_{12}^{p1})^2 + (a_{22}^{p1})^2 + (a_{32}^{p1})^2}.$$

Desta forma, q_2 e r_{23} podem ser calculados tal como mostrado nas equações 2.9 e 2.10

$$q_{12} = \frac{a_{12}^{p1}}{r_{22}}, q_{22} = \frac{a_{22}^{p1}}{r_{22}}, q_{32} = \frac{a_{32}^{p1}}{r_{22}} \tag{2.9}$$

$$r_{23} = q_2^T a_3^{p1} = q_{12} a_{13}^{p1} + q_{22} a_{23}^{p1} + q_{32} a_{33}^{p1}. \tag{2.10}$$

A matriz A^{p2} é obtida da seguinte equação 2.11.

$$\begin{aligned} a_{11}^{p2} = 0, a_{12}^{p2} = 0, a_{13}^{p2} = a_{13}^{p1} - r_{23} q_{12} \\ a_{21}^{p2} = 0, a_{22}^{p2} = 0, a_{23}^{p2} = a_{23}^{p1} - r_{23} q_{22} \\ a_{31}^{p2} = 0, a_{32}^{p2} = 0, a_{33}^{p2} = a_{33}^{p1} - r_{23} q_{32} \end{aligned} \tag{2.11}$$

Similarmente, repetindo os passos na matriz A^{p2}, r_{33} pode ser calculado normalizando a terceira coluna vetor da matriz A^{p2} e obter os valores de q_{13}, q_{23} e q_{33}. Finalmente, $A = QR$ é obtida na qual Q e R estão dados pela equação 2.12.

$$Q = \begin{pmatrix} q_{11} & q_{12} & q_{13} \\ q_{21} & q_{22} & q_{23} \\ q_{31} & q_{32} & q_{33} \end{pmatrix}, R = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ 0 & r_{22} & r_{23} \\ 0 & 0 & r_{33} \end{pmatrix} \tag{2.12}$$

Resumindo, os principais métodos de decomposição e de eliminação são o QR e a eliminação Gaussiana para se obter soluções de sistemas lineares. A Tabela 2.3 mostra algumas das vantagens e desvantagens do método QR para tratar matrizes não quadradas, assim como não singulares para obter a decomposição. Entre outras vantagens do método QR pode se citar o fato de permitir tratar de sistemas esparsos e, adicionalmente, seu número de FLOPs é menor que o da eliminação Gaussiana.

Tabela 2.3. Comparação entre os diferentes métodos de decomposição e/ou eliminação de matrizes

	FLOP (Operações em ponto flutuante)	Esparsa	Quadrada	Restrições
Eliminação Gaussiana	$\frac{2n^3}{3} - \frac{n^2}{2} - \frac{n}{6}$	Não	Sim	Não Singular
Decomposição QR	$2mn^2$	Sim	Naõ necessário	$m \geq n$

2.3.2 O Cálculo da Raiz Quadrada de uma Matriz

A Decomposição de Schur

Se existe uma matriz unitária $U \in C^{m \times m}$ tal que $B = U^H AU$, então B é **equivalente unitariamente** a $A \in C^{m \times m}$. Caso U seja real, então B goza da propriedade de **equivalência ortogonal real** em relação a A.

***Teorema* 2.1** *(Teorema de Schur). Seja $A \in C^{m \times m}$ com m autovalores $\lambda_1, \lambda_2, ..., \lambda_m$, distintos ou não, e em qualquer ordem dada, então existe uma matriz unitária $U \in C^{m \times m}$ como apresentado na equação 2.13, sendo triangular superior (ver definições no apêndice B), com entradas diagonais $T\left(i,i\right) = \lambda_i$, $i = 1 : m$. Ou seja, toda matriz quadrada é equivalente unitariamente a uma matriz triangular superior (ou triangular inferior).*

$$U^H AU = T \tag{2.13}$$

A Demonstração do Teorema de Schur pode ser observada no apêndice B.

A Raiz Quadrada de uma Matriz

A raiz quadrada de uma matriz A de $n \times n$ é uma matriz X de forma tal que satisfaz a seguinte equação (2.14).

$$X^2 = A, \tag{2.14}$$

Tal matriz X é chamada de *raiz quadrada de* A, sendo que a mesma equação pode ser escrita como $X = A^{1/2}$. Diferente da raiz quadrada de um escalar, a raiz quadrada de uma matriz pode não existir ou ser única. Se A tem ao menos $n-1$ autovalores não zero, então A tem uma raiz quadrada. Já que o número de possíveis raízes quadradas de uma matriz dada A varia dependendo da singularidade da matriz de dois até infinito (considerando a matriz identidade), uma matriz A real tem uma raiz quadrada real se a matriz dada A não tem autovalores reais negativos. No caso de matrizes reais simétricas de $n \times n$, onde $z^T Az \geq 0$ é positivo para qualquer vetor coluna não zero z de n números reais (isso é uma matriz semi-definida positiva), existirá uma única raiz quadrada simétrica semi-definida positiva. Isto significa que para se obter uma única raiz quadrada $A^{1/2}$, para a qual cada autovalor tem uma parte real positiva, a matriz dada A não deve ter autovalores reais negativos [72] [73].

A maioria de propostas numéricas para calcular a raiz quadrada de uma matriz são baseadas no método de Newton, sendo que vários métodos computacionais são baseados na igualdade $f(XAX^{-1}) = Xf(A)X^{-1}$. Se X pode ser encontrada de tal forma que $B = XAX^{-1}$ tem a propriedade que $f(B)$ é facilmente avaliada, então um método obvio resulta. Quando A é diagonizável, B pode ser tomada como diagonal e a avaliação de $f(B)$ é trivial. Para uma matriz A geral, se X é restringida para ser unitária então A pode ser reduzida a uma fatoração de Schur da forma $A = QTQ^{-1}$, onde Q é unitária e T é triangular superior. Dado que Q é unitária, Q^{-1} é igual à transposta conjugada de Q. Esta decomposição é computada pelo algoritmo de decomposição QR [74][75] apresentado em 2.3.1.

Uma forma recursiva baseada no teorema de fatoração de Schur foi proposta por Björck e Hammarling em [23], conhecida como método de Schur, que presenta uma alta estabilidade e precisão. O método de Schur vem da decomposição de Schur (ver 2.3.2) a qual transforma a matriz dada A de $n \times n$ em $A = QTQ^{-1}$. Desta maneira, o problema é reduzido ao cálculo da raiz quadrada da matriz triangular superior T.

Embora essa forma recursiva permite manter uma estabilidade e precisão altas, a sua implementação em *hardware* foi por muito tempo desconsiderada devido a sua complexidade e inviabilidade da sua implementação, deixando apenas as aplicações do algoritmo para implementações em *software*. Com o avanço na complexidade das novas tecnologias, especialmente os FPGAs, algoritmos para processamento de sinais digitais vem ganhando uma grande força na sua implementação em *hardware* dedicado para acelerar os diferentes meios onde são requeridos.

2.4 Considerações Finais do Capítulo

Neste capítulo foi introduzida a tecnologia de *hardware* reconfigurável, especialmente os FPGAs. Os dispositivos FPGAs permitem um alto grau de paralelismo tornando-os bons candidatos para aplicações embarcadas onde soluções baseadas em *software* não alcançam os requisitos estabelecidos. A grande maioria destas implementações são voltadas para soluções onde há uma demanda pela computação de alto desempenho e baixo consumo de potência, como no caso onde existem operações matriciais.

Com essa versatilidade do *hardware* reconfigurável e o alto grau de paralelismo espera-se que as implementações em FPGAs permitam ganhos em desempenho, custo e precisão, logrando estimular o uso desta tecnologia em diferentes ambientes de pesquisa. Adicionalmente, arquiteturas baseadas em arranjos sistólicos são candidatas interessantes quando necessário grande poder de processamento em paralelo, como no caso de sistemas matriciais, o que torna as FPGAs plataformas ideais de desenvolvimento. Esses arranjos sistólicos tem como vantagem seu simples e regular desenho que pode acelerar o fluxo computacional dos algoritmos implementados.

A representação numérica utilizada no desenvolvimento deste trabalho também faz parte importante, pois aportam parâmetros como precisão além de outros fatores que afetam diretamente a implementação em *hardware* como o consumo de recursos associado ao comprimento da palavra, etc. Este capítulo fez uma descrição do formato IEEE−754 para representação em ponto flutuante e mostrou como é usado para representar a notação cientifica e poder aproveitar uma representação numérica com intervalos maiores e de forma dinâmica. Nos capítulos seguintes será mostrado como alguns erros associados aos cálculos afetam esta representação e consequentemente a acurácia dos resultados obtidos.

Também, foram descritas as diferentes decomposições matriciais usadas tanto para resolver sistemas lineares quanto para encontrar a solução de funções matriciais como no caso da raiz quadrada da matriz. O estudo mostrou que cada método tem suas vantagens e desvantagens, em que, parâmetros tais como a acurácia do método, a complexidade das operações matriciais, o número de operações diferentes a implementar (e.g., soma, divisão, multiplicação, raiz quadrada), assim como a quantidade de acessos à memória, entre outras, permitem escolher o tipo de método a ser usado.

Conjuntos de equações lineares da forma $Ax = b$ ou o cálculo da raiz quadrada de uma matriz $X = A^{1/2}$ aparecem em muitas aplicações, onde, independente da aplicação, seja envolvendo problemas de predição com o filtro de Kalman ou estimação dos canais de comunicação MIMO, o problema é sempre encontrar a solução numérica do sistema matricial. A eliminação de Gaussiana, fazendo uso de algumas restrições, permite tratar sistemas matriciais onde a sua matriz de entrada deve ser simétrica e não singular, além de definida positiva, como os casos das matrizes de covariância. Este também é um critério para realizar uma implementação mediante a decomposição de Cholesky que apresenta maior desempenho computacional.

A decomposição QR, diferente dos outro métodos mostrados permite tratar sistemas matriciais quadrados $n \times n$ ou retangulares $m \times n$ sendo $m > n$, com m igual as linhas e n as colunas. Decomposições como QR, Cholesky apresentam grande aplicabilidade no desenvolvimento de sistemas para processamento de sinais digitais embora também apresentem um alto grau de complexidade, vem ganhando grande interesse na comunidade devido aos avanços tecnológicos que permitem implementações mais complexas.

As implementações em arquiteturas reconfiguráveis dos algoritmos estudados permitiram ganhar em desempenho assim como na acurácia deixando

um suporte para que futuras propostas possam usá-las como base no desenvolvimento de sistemas embarcados. Finalmente, os conceitos apresentados neste capítulo são importantes para o entendimento das implementações de *hardware* realizadas no contexto deste trabalho. Estas implementações são descritas nos capítulos seguintes.

Capítulo 3 IMPLEMENTAÇÃO DAS ARQUITETURAS PARA SOLUÇÃO DE SISTEMAS DE EQUAÇÕES LINEARES E INVERSÃO DE MATRIZES

Este capítulo apresenta as implementações realizadas envolvendo a solução de sistemas lineares incluindo os resultados de síntese, desempenho e propagação do erro. O capítulo está organizado da seguinte forma: (1) comentários prévios relacionados à importância das operações matriciais são expostos nesta primeira seção, (2) seguindo, uma seção será utilizada para descrever o algoritmo de eliminação Gaussiana usado nas implementações propostas neste trabalho, (3) uma seção é dedicada para a revisão bibliográfica sobre o estado da arte das implementações em *hardware* do algoritmo e por fim, (4) em uma última seção, é realizada uma descrição das implementações para inversão de matrizes, assim como para a solução de sistemas lineares, realizando uma discussão sobre os pontos mais relevantes associados aos resultados obtidos.

3.1 Comentários iniciais

Frequentemente, a maioria dessas aplicações em ciência e engenharia envolve algoritmos que devem lidar com estruturas de dados matriciais e suas respectivas operações. Uma das operações mais importantes e computacionalmente custosas é a inversão de matrizes para a qual muitas abordagens numéricas bem conhecidas são aplicadas, principalmente em plataformas de *software*. Todavia, a complexidade computacional da inversão de matrizes é uma pergunta aberta, soluções em *software* sub-cúbicas tais como

$O(n^{2.807})$ Strassen e $O(n^{2.376})$ Coppersmith-Winograd aplicadas para inverter matrizes $n \times n$ são de grande interesse teórico mas não são usadas na prática devido a que sua aceleração é apenas visível para matrizes de grande porte em *software*, o que torna inadequado para implementações em *hardware* (e.g. Capítulo 12 de [11]).

Algoritmicamente, métodos simples tal como a Eliminação de GJ *Gauss-Jordan*, são importantes para o desenvolvimento de implementações com arquiteturas, embora conhecida sua alta complexidade computacional ($O(n^3)$). A implementação em *hardware* destes algoritmos é interessante, em termos de se poderem evitar os problemas relacionados ao gargalo de *von Neumann*, principalmente os relacionados com *ler* instruções *desde* a memória RAM. No caso de operações matriciais, que são implementadas em *hardware*, apenas operações de *escrever/ler* dados devem ser executadas, sem a complexidade dos passos de *decodifição/execução* relacionadas às instruções de execução, as quais estão fortemente ligadas ao modelo de *von Neumann*. Neste caso, deve-se prover um escalonamento adequado dos dados, mediante contadores, sincronizados adequadamente, como proposto em [44].

Muitos algoritmos de inversão de matrizes têm uma complexidade cúbica mas apresentam uma simplicidade algorítmica onde um *hardware* reconfigurável específico pode prover uma solução muito atraente [76, 77]. O uso de métodos de decomposição e eliminação devém ser introduzidos para a inversão de matrizes de grande porte, uma vez que as abordagens analíticas resultam em arquiteturas não escaláveis pelo uso de determinantes. Alguns métodos de decomposição/eliminação tais como Eliminação GJ e decomposição QR (QRD) são tradicionalmente usados por causa da sua simplicidade ou estabilidade (ou precisão do algoritmo), além das decomposições LU e *Cholesky*. A eliminação GJ apresenta uma maior simplici-

dade dada as operações básicas necessárias para sua implementação. Por outro lado, o método de *Cholesky* está limitado para matrizes positivas e não singulares, enquanto que o QRD pode ser usado em quase qualquer classe de matrizes [78].

Devido a que em algumas representações matemáticas existem modelos envolvendo a solução de sistemas de equações lineares (modelos matriciais), há uma dificuldade relacionada a qual seria o algoritmo mais adequado para operar com esse tipo de representações, sem demandar alto custo computacional, e mantendo precisão nas soluções. Por causa disso, o maior desafio em análise numérica está relacionado ao desenvolvimento de algoritmos eficientes para a realização de cálculos matriciais.

A eliminação Gaussiana é uma das mais antigas técnicas propostas na álgebra linear, e ainda hoje é um dos métodos numéricos mais populares para a solução de sistemas de equações lineares. A eliminação Gaussiana é também uma técnica usada para calcular a inversa da matriz quando a matriz, dada A, é densa e linearmente independente.

3.2 Algoritmo de Eliminação GJ

O algoritmo de eliminação GJ é um método baseado na Eliminação Gaussiana que coloca zeros acima e abaixo de cada pivô dando deste modo a matriz inversa. Seja A uma matriz $n \times n$, I a matriz identidade $n \times n$ e X a matriz de incógnitas $n \times n$. A solução do sistema linear $A \times X = I$ será o resultado $X = A^{-1}$. Este método, chamado assim devido a Carl Friedrich Gauss e Wilhelm Jordan, é um algoritmo da álgebra linear para determinar as soluções de um sistema de equações lineares, assim como para obter matrizes inversas.

O método aplica primeiramente a eliminação Gaussiana na matriz aumentada $n \times 2n$, $\tilde{A} = [A\ I]$, dando como resultado uma matriz da forma $[U\ H]$ na qual U é triangular superior. Segundo, $[U\ H]$ é transformado por meio de operações elementares de linhas em uma matriz aumentada da forma $[I\ K]$. Por fim, extraindo o componente direito da matriz aumentada obtém-se K que é a matriz inversa: $A \times K = I$ [70, 71]. A seguir são descritos os quatro principais passos do algoritmo 1 de Gauss-Jordan com *pivô parcial* que começam na linha $i = 1$ e com a matriz aumentada $[A\ I]$:

1. Pivô Parcial: localizar o pivô, que é o maior elemento na coluna i. Considere os elementos em linhas i até n, então é trocada a linha i com a linha que contem o pivô.

2. Triangularização do sistema: Eliminar todos os elementos abaixo da diagonal na coluna i, subtraindo cada linha abaixo da linha pivô pelo múltiplo da linha pivô. Então, incrementa-se i e repete-se o passo 1 (se $i < n$), até se obter na matriz esquerda uma forma triangular superior.

3. Diagonalização: Eliminar todos os elementos acima da diagonal na coluna i, subtraindo cada linha acima da linha pivô por um múltiplo da linha pivô. Então, diminuir i e repetir o passo até que todos os elementos acima da diagonal sejam zero e o componente esquerdo da matriz aumentada se torne uma *matriz diagonal*.

4. Normalização: Dividir cada linha i pelo recíproco do pivô assim os elementos da diagonal tomam valor de 1. Então o componente esquerdo da matriz aumentada será a matriz identidade e a parte direita será a inversa da matriz.

Algoritmo 1: Pseudocódigo para a eliminação de Gauss-Jordan com pivô parcial

Input: A

Saidas A^{-1};

$[n] = size(A)$;

$I = identity(n)$;

Colocar a matriz na forma de matriz aumentada $B = [AI]$;

Pivô parcial: **for** $j = 1...n$ **do**

  for $i = 2...n$ **do**

    Se $B(j,j) < B(i,j)$ trocar as linhas $B(j) \leftrightarrow B(i)$;

  end

end

Triangularização do sistema: **for** $j = 1...n$ **do**

  for $i = 2...n-1$ **do**

    Encontro o elemento pivô $\Longrightarrow 1/B(j,j)$;

    Multiplico o elemento pivô com a linha $1/B(j,j) * B(i)$;

    Soma de linas $B(j) + B(i)$ e o resultado é $\Longrightarrow B(i)$

  end

end

Normalização: **for** $i = 1...n$ **do**

  Multiplico toda a linha $1/B(i,i) * B(i)$;

end

Diagonalização: **for** $j = n...1$ **do**

  for $i = n-1...2$ **do**

    Encontro o elemento pivô $\Longrightarrow 1/B(j,j)$;

    Multiplico o elemento pivô com a linha $1/B(j,j) * B(i)$;

    Soma de linas $B(j) + B(i)$ e o resultado é $\Longrightarrow B(i)$

  end

end

3.3 O estado da arte: Implementações em *hardware* reconfigurável do algoritmo de eliminação Gaussiana

Vários trabalhos reportam implementações de inversões de matrizes usando métodos diferentes tais como *Cholesky* [79], *LU* [80] e *Gauss-Jordan* [81, 82]. Existem também algumas arquiteturas VLSI para inversões de matrizes usando o método de fatoração *QR* que não é especialmente desenhado para FPGAs. Em [83] é apresentado um algoritmo para realizar uma inversão de matrizes usando o método de QRD-GR com uma arquitetura VLSI em arranjo sistólico. Adicionalmente, em [84] é apresentado um algoritmo usando QRD-MGS.

Trabalhos mais recentes para matrizes de pequeno porte, usando FPGAs, não são facilmente estendidos a matrizes de grande porte como apresentado em [85], onde é discutida uma solução prática para matrizes maiores do que 4×4. Em [86], o trabalho foi feito para matrizes de 16×16 junto com a sua respectiva análise, porém, a implementação em FPGA é mostrada apenas para matrizes de 4×4. Em [87] é apresentada uma arquitetura para inverter matrizes usando a fatoração *QR* baseada no algoritmo (RLS). Essa proposta usa o algoritmo *Square Givens Rotations* e seu desenho em arranjo sistólico foi implementado com uma representação em ponto flutuante de 20 *bits* de comprimento.

Contudo, a maioria de trabalhos reportam apenas representações em ponto fixo e arquiteturas que não podem ser estendidas a matrizes de grande porte. Uma exceção é a proposta apresentada em [82] na qual uma arquitetura é proposta usando uma representação em ponto flutuante com dupla precisão e usando memórias externas para alcançar os requisitos propostos. Entretanto, nesse trabalho não foi apresentado um estudo em termos da precisão tendo em conta o aumento nas dimensões da matriz, ocultando

a importância de levar em consideração a propagação do erro quando se trabalha com matrizes.

No contexto dos sistemas lineares, existem poucos trabalhos que podem resolver sistemas de equações lineares, e ainda menos propostas capazes de lidar com sistemas de grande porte. Em [88] e [89], os autores apresentam uma proposta para resolver um sistema linear de equações sem o uso da operação de divisão, chamada de *division-free parallel architecture* (DFPA), a qual é capaz de lidar com sistemas grandes (cerca de 96 equações). Entretanto, os resultados sobre análises da propagação do erro, tendo em conta a variação na dimensionalidade dos sistemas envolvidos, não foram apresentados.

Resumindo, pode-se ver na tabela 3.1 as diferentes implementações feitas em FPGAs com diferentes métodos propostos na literatura para resolver sistemas de equações lineares. Isto mostra uma carência de *hardware* dedicado que permita encontrar uma solução numérica aos problemas envolvendo sistemas lineares.

Tabela 3.1. Comparação entre as referências encontradas para solução de sistemas matriciais

Método		Ano	Vs. *Software*	Max. tamanho	Esparsa	Off-chip RAM	Requerimentos de A e Precisão
Direto	Gauss_Jordan [90]	2012	—	120	Densa	Não	Não-Singular
	SVD [91]	2011	N/A	32x127	—	—	—
Iterativo	Conjugate Gradient, [92, 93, 94, 95]	2006	1.3x	2000	Esparsa	Sim	Definida Positiva, Simétrica
		2007	MAPStation SRC-6	—	—	—	
		2009	MAPStation SRC-6	—	—	—	
		2010	—	58	Densa	Não	
	Minres [40]	2008	8.8x	145	Densa	Não	Simétrica
	Lanczos [96]	2012	—	335	—	—	—
	BICGSTAB [97]	2013	—	—	—	—	—
	BICG	Sem referencia encontrada					
	QMR						
	QMRCGSTAB						
	TFQMR						
	CGS						
	GEMRES						
Ambos	LAPACKrc (CG) [98]	2009	Hibrido	—	—	—	—

3.4 Arquiteturas propostas para inversão de matrizes

3.4.1 Uma adequada implementação em FPGA de uma inversão de matrizes em ponto flutuante baseada na Eliminação de GJ

A arquitetura apresentada nesta subseção tem como referência o trabalho [99], onde os elementos da matriz foram representados usando o sistema padrão de ponto flutuante $IEEE-754$, com uma configuração para precisão simples. Para realizar os cálculos foram usadas as bibliotecas de ponto flutuante desenvolvidas no laboratório LEIA da UnB [100]. Os blocos disponíveis de memória RAM interna, no dispositivo utilizado (Virtex-5), foram usados para armazenar os valores dos componentes da matriz na forma de vetores, os quais são acessados em uma ordem específica.

Na figura 3.1 é apresentada a estrutura geral da arquitetura proposta para resolver inversões de matrizes, a qual representa uma proposta que usa adequadamente as capacidades da FPGA, seguindo o algoritmo descrito na seção 1. A arquitetura geral é descrita em (3.2a), a qual é composta de um circuito chamando de *circuito de controle*, uma unidade de memória RAM e três unidades aritméticas de ponto flutuante chamadas de *Mult*, *Add/Sub* e *Div* incluidas na unidade geral de eliminação de matrizes. As unidades de *Mult* e *Add/Sub* são compostas de 10 multiplicadores e 10 somadores, respectivamente, usando os circuitos de ponto flutuante de [100], sendo o número de multiplicadores e somadores limitado apenas pelos recursos *hardware* disponíveis no FPGA selecionado. O circuito de controle geral é mostrado na figura (3.2b), onde pode ser observada a sua estrutura composta de um conjunto de FSMs (máquinas de estados finitos) chamadas de módulos: *encontrar pivô*, *troca de linhas* e uma unidade chamada de *eliminação de matriz* que contém os blocos de *triangularização*, *diagonalização* e *normalização*.

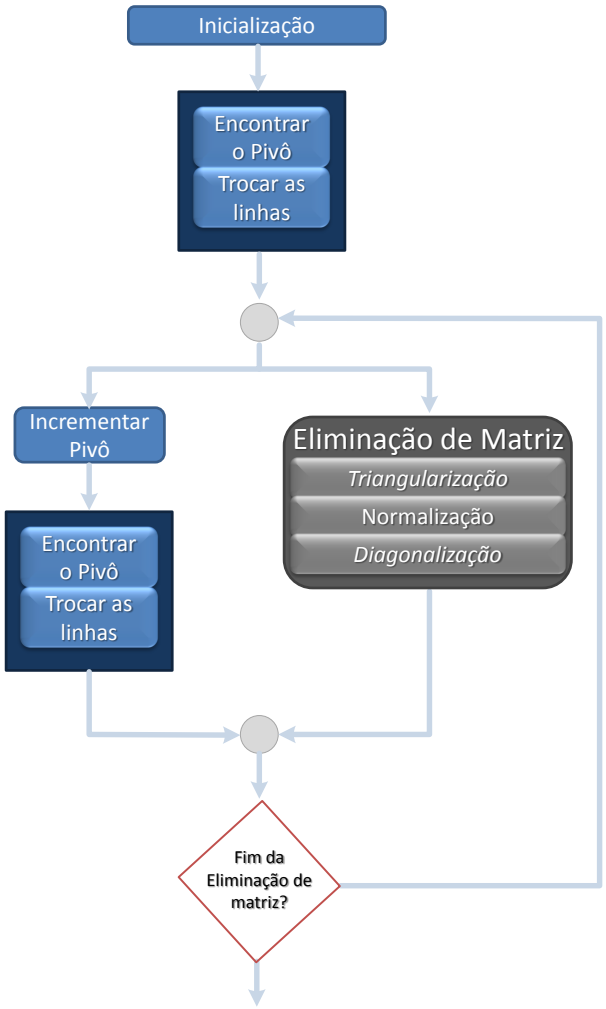


Figura 3.1. Gráfico sequencial da arquitetura proposta

O circuito de controle tem a tarefa de controlar os acessos às memórias RAM internas, assim como controlar os acessos aos dados das diferentes unidades aritméticas, tais como (*Div*, *Mult* e *Add/Sub*). Adicionalmente, o mesmo sistema de controle visa distribuir esses dados segundo sejam requisitados por cada unidade. O algoritmo mapeado nesta arquitetura começa por um processo de inicialização (ver figura. 3.1) onde os parâmetros básicos são atribuídos a cada unidade.

Os módulos *encontrar pivô* e *troca de linhas* (ver figura. 3.2b), trocam as linhas respectivas sempre deixando o pivô na linha i e coluna j, sendo encontrado o valor do pivô correspondente X tal como mostrado na equação 3.1.

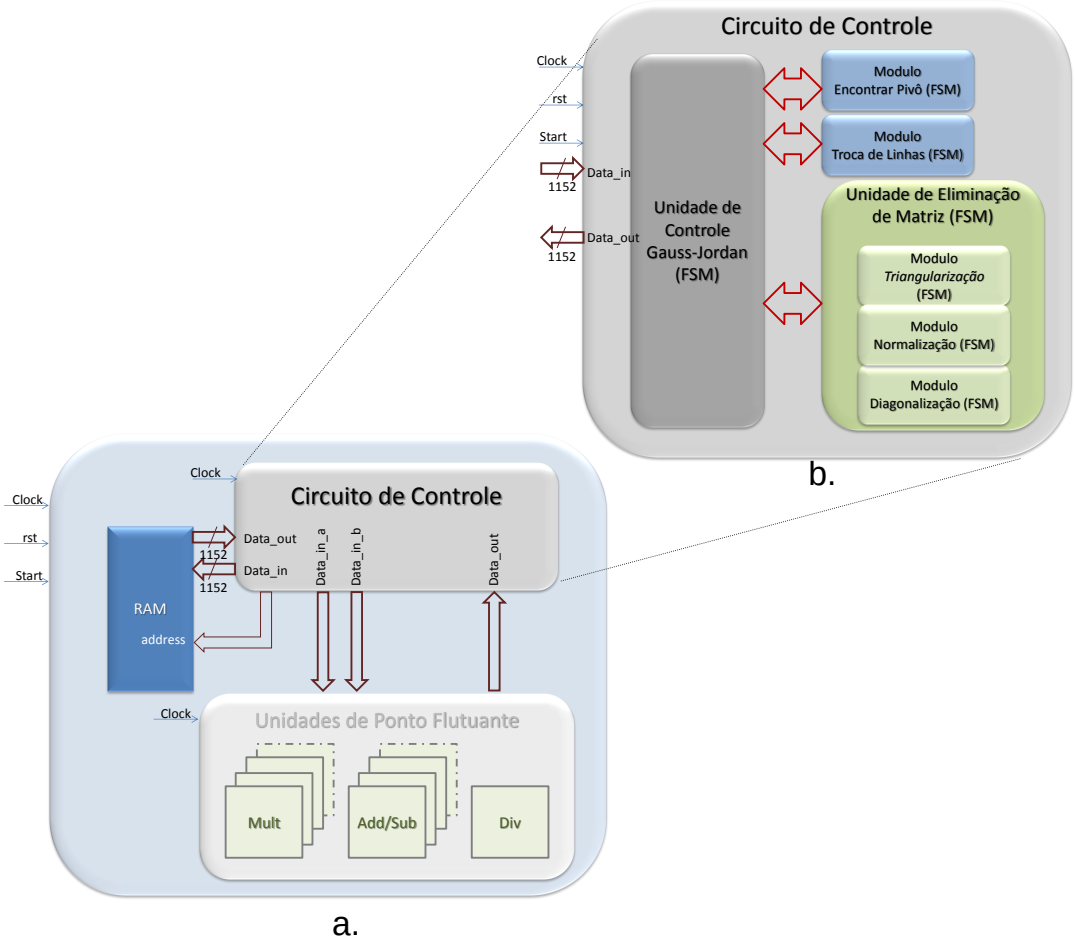


Figura 3.2. Estrutura geral do sistema: a). O sistema GJ total b). A arquitetura do circuito controlador

$$X = \frac{A(i,p)}{A(p,p)} \tag{3.1}$$

onde $A(p,p)$ é o elemento pivô e $A(i,p)$ são todos os elementos abaixo do pivô.

Depois do processo inicial, o sistema passa para o seguinte passo, envolvendo o módulo de *eliminação de matriz* e o *incrementar pivô* (vide figura. 3.1). Neste caso, o módulo *eliminação de matriz* é responsável por realizar os cálculos seguindo a equação 3.2.

$$A(i,j) = A(i,j) - A(p,j) * X, \tag{3.2}$$

onde o novo valor de $A(i,j)$ corresponde à subtração do elemento embaixo do pivô com o elemento pivô multiplicado pelo líder encontrado, alcançando desta forma um pivô parcial. É importante observar que os cálculos realizados são feitos em toda a matriz aumentada (ver a descrição do algoritmo na seção 3.2).

O processo descrito anteriormente encontra-se em um laço de repetição (vide figura. 3.1) o qual envolve ambas as partes do algoritmo: *triangularização*, *normalização* e *diagonalização*. Para se conseguir isto são usadas as diferentes unidades aritméticas de ponto flutuante: *div*, *mult* e *add/sub* (ver figura. 3.2a). Assim pode ser observado que o número de multiplicações necessárias utilizadas no módulo de *eliminação de matriz* se incrementa com o tamanho da matriz, o que está relacionando com sua complexidade computacional ($O(n^3)$) segundo a equação 3.3,

$$\sum_{i=1}^{n-1}(i+1)i + \sum_{i=1}^{n-1} i + 2 \times n \times \sum_{i=1}^{n-1} i + n^2, \tag{3.3}$$

onde n representa o tamanho do matriz.

O processo de normalização é executado no intuito de transformar a matriz diagonal atual em uma matriz identidade. Este procedimento garante que no lado direito da matriz aumentada estará a matriz inversa. A tarefa de normalização é desenvolvida pelo módulo *normalização*, o qual se encontra dentro da *eliminação de matriz* (ver figura. 3.2b). Neste caso, a equação 3.4 é executada para calcular o reciproco de cada elemento da diagonal,

sendo que a equação 3.5 é aplicada para obter uma matriz identidade, na parte esquerda da matriz aumentada.

$$R = \frac{A(p,i)}{A(p,p)} \tag{3.4}$$

$$A(p,j) = A(p,j) \times R \tag{3.5}$$

3.4.1.1 Características importantes da arquitetura

Vários aspectos de optimização foram introduzidos nesta proposta tendo em conta: (a) os acessos à memória RAM, (b) a operação de troca de linha e (c) o paralelismo das operações aritméticas de ponto flutuante. Neste caso, os principais aspectos de optimização são os seguintes:

- *O acesso a memória RAM*: As memórias internas do FPGA selecionado permitem ao sistema alcançar operações de *escrita/leitura* sobre palavras de 1152 *bits* de comprimento. A implementação em *hardware* proposta toma vantagem da capacidade de paralelismo do FPGA. A figura. (3.2a) apresenta os barramento de dados *Data_out* e *Data_in* com este comprimento para operações de *escrita/leitura desde/para* as memórias RAM.

- *A operação de troca de linha*: A operação de intercâmbio de linhas foi otimizada evitando as operações de escrita física, que são computacionalmente custosas. Neste caso, um registrador chamado de *pos_memória* foi usado (vide figura. 3.3, representando uma configuração usando como exemplo uma matriz de 4×4). Assim, cada

conjunto de 6 bits representa e armazena a posição real dos primeiros elementos da linha da matriz na memória, uma vez que 36 posições representam o número de linhas da maior matriz invertida, sendo que para trocar duas linhas da matriz é somente necessário a troca dos valores dos registradores.

Por exemplo, se o número atribuído na posição 1 corresponde ao valor de zero, indicado pelos 6 *bits*, significa que o valor da primeira linha da matriz está armazenado na posição zero da memória RAM (como apresentado na figura 3.3). Fato este que atribui os valores de cada posição das linhas da matriz no registro *pos_memoria*, sem ter que passar por um armazenamento físico, ou ordenamento na memória RAM, evitando assim uma operação de escrita na mesma.

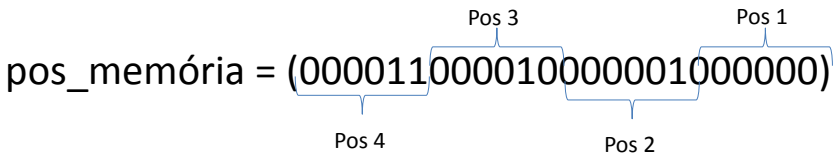


Figura 3.3. Registrador pos_memória

- *O paralelismo das operações aritméticas de ponto flutuante*: Dez multiplicadores e dez somadores em ponto flutuante foram usados, desenvolvendo as operações requeridas no módulo de *eliminação de matriz*, sendo o número de multiplicadores e somadores limitado apenas pelos recursos *hardware* disponíveis no FPGA selecionado. Isto representa o nível de paralelismo, permitindo desenvolver operações sobre dados de memória de 1152 bits (que representam até 36 dados em ponto flutuante de precisão simples). No caso de ter 36 números (por exemplo, uma matriz de 36×36 elementos) sete processos em paralelo devem ser realizados, e na última operação apenas é necessário utilizar 2 dos 10 multiplicadores e somadores disponíveis (ver figura. 3.4). De fato, existem dois conjuntos de cinco multiplicadores e somadores, um para cada parte da matriz aumentada.

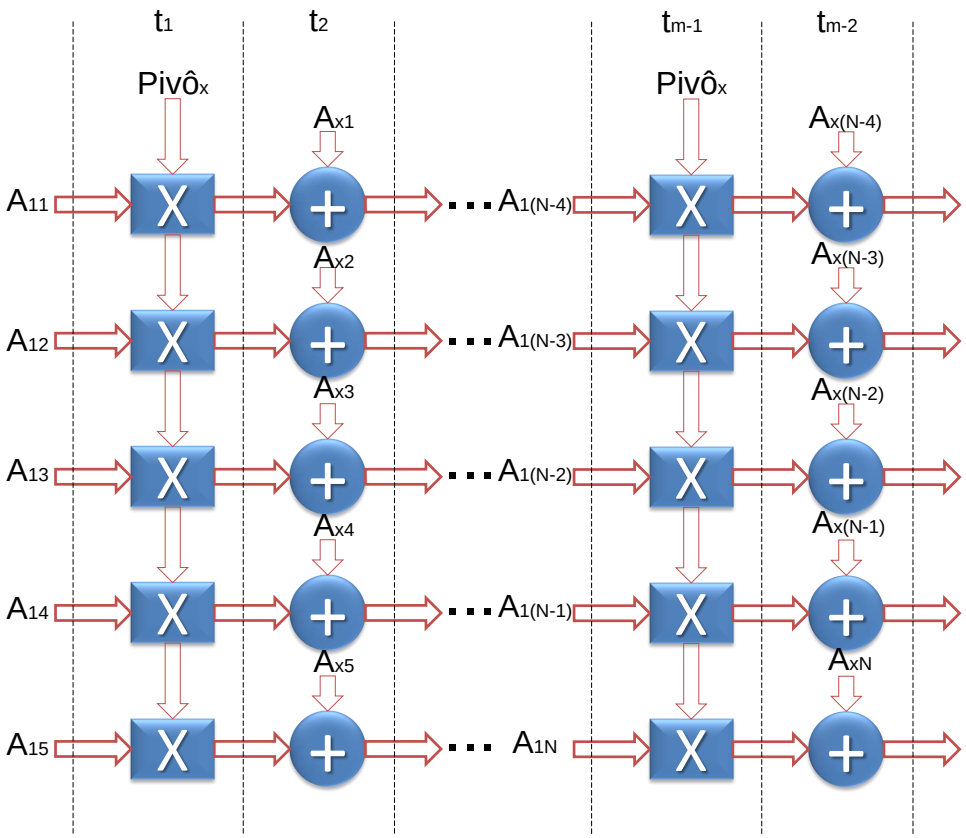


Figura 3.4. Estrutura paralela para t_i ciclos usando um conjunto de cinco multiplicadores e cinco somadores

3.4.1.2 Resultados da implementação

O *software* de descrição de *hardware* usado neste projeto foi o *Xilinx Integrated Software Environment* (ISE) 10.1. O FPGA usado foi a Virtex-5 XC5VLX110T. A escolha do dispositivo neste caso, o Virtex-5, foi devido à disponibilidade do kit no nosso grupo de pesquisa.

A figura. 3.5 descreve o comportamento do erro médio (ME) para diferentes tamanhos de matrizes quadradas na faixa de $n = 4$ até $n = 36$ usando 100 amostras para cada tamanho n, usando MatLab como base para a estimação estatística do erro, tendo em conta os resultados obtidos. Os dados para cada tamanho de matriz foram gerados usando o comando $A = double(rand(x, x))$; que define a matriz com números reais em preci-

são dupla no intervalo de [0, 1], tal como proposto em [82].

As variações do erro podem ser explicadas pelo erro de truncamento do multiplicador. Por exemplo, em [100] o erro do multiplicador para uma precisão simples é de 10^{-7}. Assim, quando são representados números com uma precisão finita, não todo número no intervalo disponível pode ser representado exatamente. Se um número não pode ser representado exatamente pelo tipo específico de dado e escala, um método de arredondamento é usado para normalizar o número em questão, em uma forma representável. Embora, a precisão diminuía sempre em uma operação de arredondamento, o custo da operação e a quantidade de *bias* que é introduzido depende do método de arredondamento em si.

A figura. 3.5 também mostra um método de regressão polinomial de ordem cinco aplicado aos resultados de erro, ilustrando a tendência do erro para aumentar enquanto o tamanho da matriz cresce. As oscilações do erro sobre a curva podem ser devidas ao número de multiplicações necessárias para a inversão da matriz e, por sua vez, como uma forte responsável pela tendência do erro para aumentar, enquanto o tamanho da matriz aumenta. Contudo, é importante ver que para matrizes de 20×20 (ou menores) a precisão é menor que 10^{-4}.

A figura. 3.6 mostra o tempo usado pela FPGA para calcular a inversa da matriz usando um *clock* de 50MHz, comparado contra o código em C usando o Eclipse CDT em um PC com processador AMD Turion-X2 Dual Core Mobile RM−72 2.10 GHz, 4GB de memória RAM e um sistema operativo de 32 bits (Vista). Embora os dispositivos FPGAs suportem altas frequências (ver tabela 3.2), o *clock* de 50MHz foi escolhido por ser o *clock* disponível na placa de desenvolvimento. Neste caso, apenas uma aplicação foi executada durante cada prova a fim de evitar uma sobrecarga da CPU. Na figura 3.6 pode ser observado que o desempenho é muito

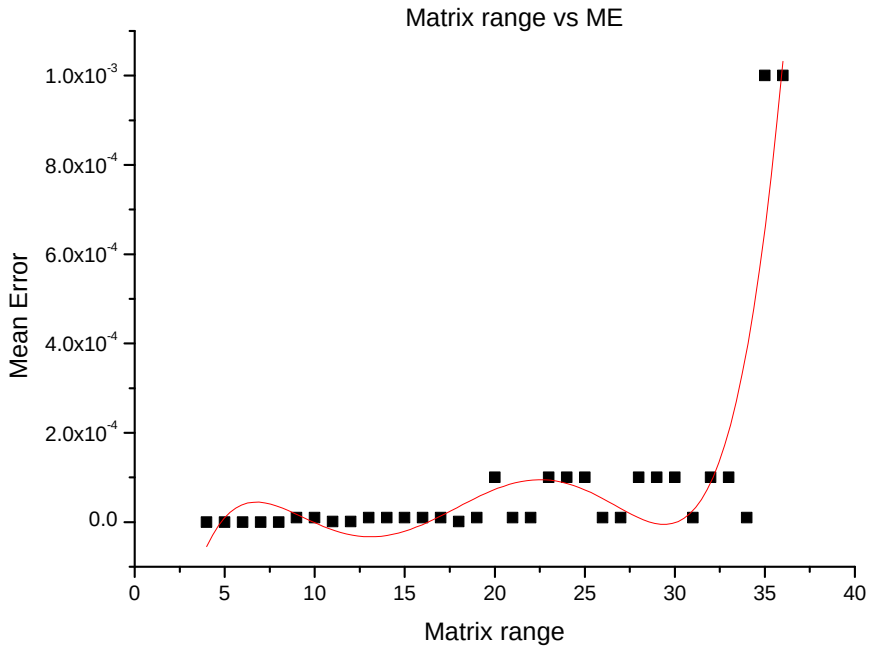


Figura 3.5. Curva para o Erro médio

melhor para matrizes maiores que 20×20 quando comparado com o PC, embora o FPGA esteja usando um *clock* de 50Mhz. Isto mostra claramente o efeito positivo de se mapear este tipo de algoritmos em arquiteturas de *hardware* apropriadas, as quais não têm as restrições do modelo de von Neumann

A tabela 3.2 resume os recursos ocupados pelas principais unidades projetadas na implementação, após o roteamento e posicionamento, para matrizes de 4×4 e 36×36. A frequência máxima que a arquitetura proposta suporta para uma matriz de 4×4 é de 301.416MHz e para uma matriz de 36×36 é de 166.984MHz. A mesma tabela também mostra que este dispositivo pode suportar matrizes com tamanhos maiores devido ao baixo consumo de recursos LUTs de aproximandamente 16.96% quando configurado para tratar sistemas de 36×36.

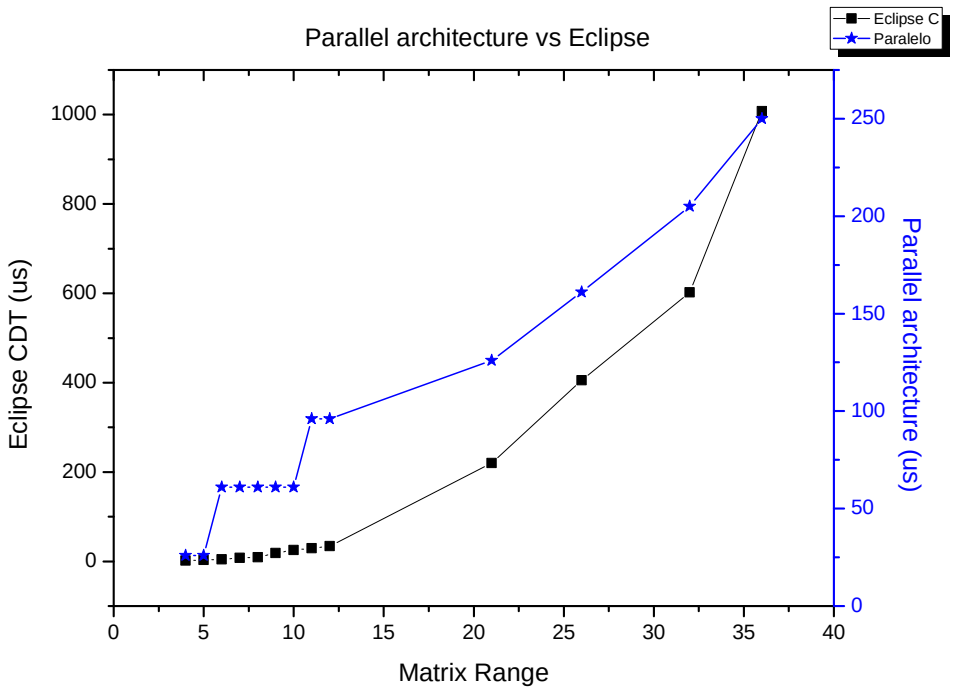


Figura 3.6. Tempo consumido pela implementação em *software* vs arquitetura proposta

Tabela 3.2. Resultados de síntese para as principais unidades com matrizes de n=4 e n=36

Unidade	LUTs (n=4/n=36)	DSP48(E)	Freq. [MHz](n=4/n=36)
Somador	845 (1.22%)		504.286
Multiplicador	146(0.21%)	16(25%)	616.797
Divisor	426(0.61%)	6(9.375%)	598.223
GJ	7131(10.31%)/10306(14.91%)		301.416/166.984
Total	8549(12.36%)/11723(16.96%)	22(34.375%)	301.416/166.984

3.4.2 Implementação em FPGA de inversão de matrizes usando uma representação em ponto flutuante com precisão simples, dupla e customizada

Neste trabalho, a arquitetura proposta e apresentanda na subseção 3.4.1 foi melhorada em vários aspectos, dando origem a uma nova arquitetura apresentada em [101]. As bibliotecas de ponto flutuante usadas são as disponíveis pela *Xilinx* as quais foram configuradas para diferentes precisões: simples, dupla e 40-*bits*. Uma consideração importante a salientar é que

todas as matrizes usadas nos experimentos são inversíveis.

Para propósitos gerais, esta arquitetura é completamente escalável devido a que tanto o tamanho quanto a precisão podem ser modificados mudando apenas alguns parâmetros de configuração, permitindo a inversão de matrizes de diferentes tamanhos, e alcançando a precisão desejada pelo usuário. Devido à falta de desenvolvimento de uma interface para *enviar/receber* os dados da matriz, os mesmos são armazenados nas memórias RAM internas do FPGA permitindo o início do processo de inversão.

A estrutura geral da arquitetura proposta é descrita na figura 3.2. A implementação em *hardware* foi dividida em três módulos e uma unidade: *módulo encontrar pivô*, *módulo troca de linhas*, *módulo de eliminação de matriz* e a *unidade circuito de controle Gauss-Jordan*. O fluxograma do *módulo de eliminação de matriz* é mostrado na figura. 3.7 onde pode ser observado que tanto para o processo *triangularização* como para o processo *diagonalização* o fluxograma é igual, com apenas uma diferença relacionada aos contadores, os quais controlam o caminho dos dados para os diferentes componentes matriciais. Nesta implementação, o processo de *triangularização* começa na primeira posição e termina na posição $n-1$. Por outro lado, o processo *diagonalização* começa logo na segunda posição e termina na posição n. Esses dois processos são desenvolvidos separadamente e de forma sequencial (um começa após o outro terminar).

Os blocos de multiplicação e soma (vide figura. 3.7) são representados pela *unidade de multiplicação* e pela *unidade de soma/subtração*, respectivamente. Essas duas unidades contêm dez operadores de multiplicação em ponto flutuante e dez operadores de soma em ponto flutuante, como apresentado na figura.3.9.

A figura.3.8 apresenta a unidade *circuito de controle* onde pode ser ob-

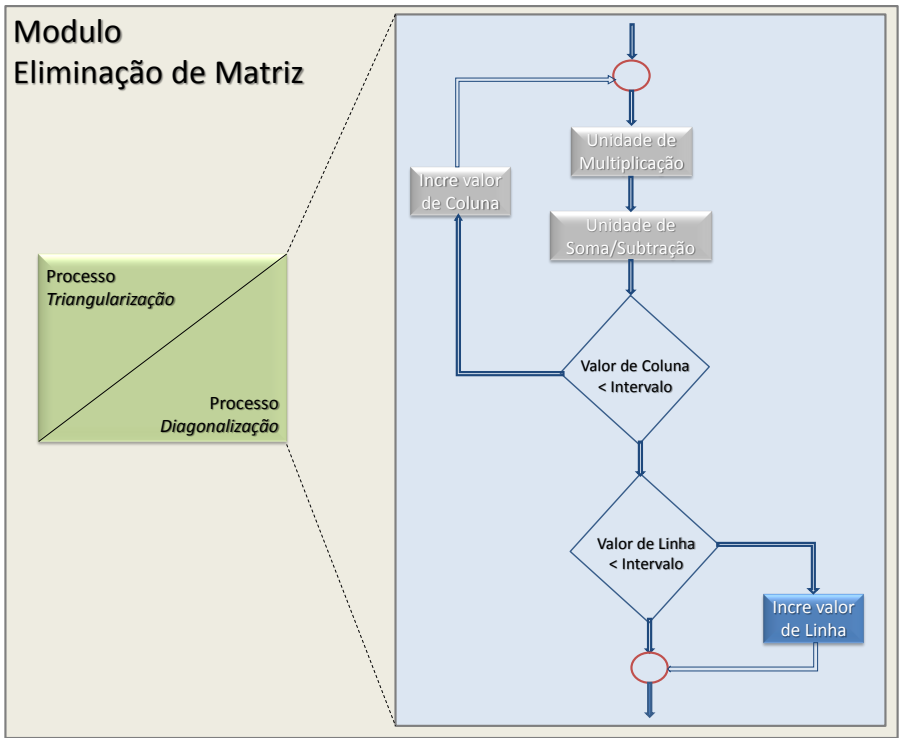


Figura 3.7. Fluxograma do Módulo de Eliminação de Matriz

servada a presença da matriz armazenada na memória interna RAM, do dispositivo FPGA selecionado (*Virtex-5*). Em [99], a arquitetura usa duas matrizes de $n \times n$, armazenadas na memória interna RAM, correspondente à matriz aumentada $[AI]$ do algoritmo de GJ. Assim, uma redução de n^2 elementos de memória foi alcançada com esta configuração.

Adicionalmente foram realizadas mudanças nas unidades de *multiplicação* e de *soma/subtração* em relação ao (descrito na referência [99]), como pode ser observado na figura.3.9. O paralelismo intrínseco dos *FPGAs* é novamente explorado pelas duas unidades devido ao incremento do número de operadores aritméticos disponíveis para executar o algoritmo GJ (passando de cinco multiplicadores em ponto flutuante e cinco somadores em ponto flutuante para dez por cada). O desempenho e área foram importantes otimizações obtidas para esta nova arquitetura, no intuito de alcançar implementações de inversões de matrizes grandes, apenas limitadas pelos recursos *hardware* e a propagação do erro associado.

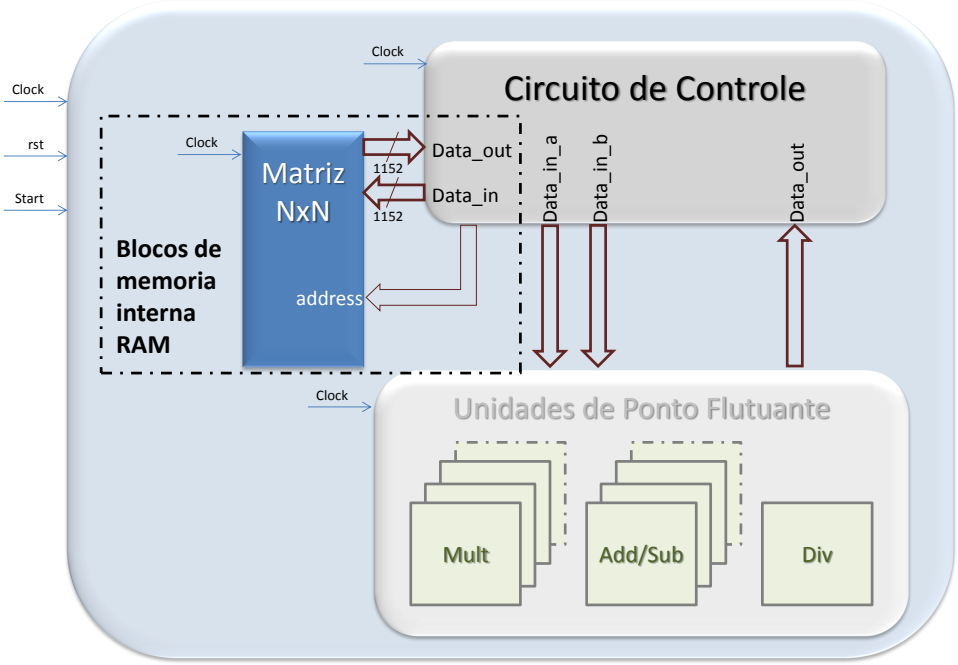


Figura 3.8. Estrutura da unidade circuito de controle

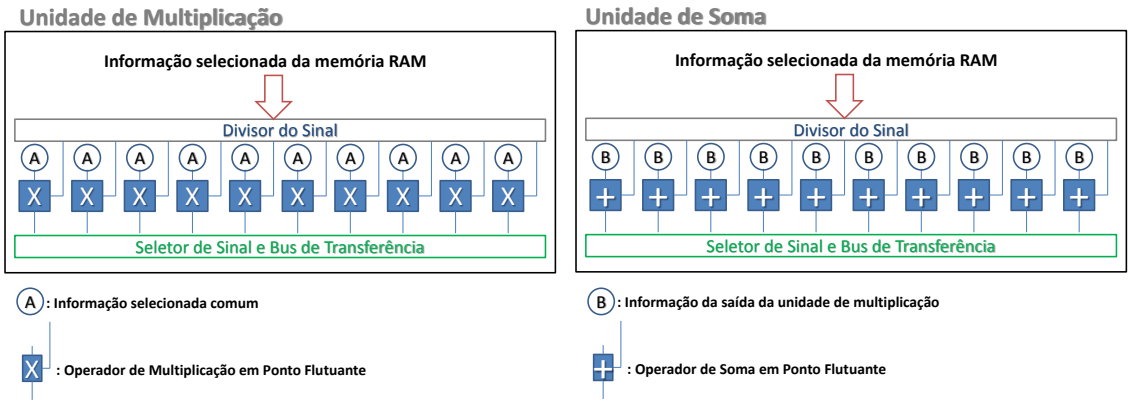


Figura 3.9. Organização estrutural das unidades de *multiplicação* e de *soma/subtração*

A abordagem atual usa a proposta apresentada em [77] que reduz pela metade os recursos de memória usados em trabalhos anteriores e também reduz o número de unidades de ponto flutuante necessárias. Para conseguir isto, a matriz identidade usada na parte direita da matriz aumentada não é armazenada na memória pela razão de que a informação necessária está contida na parte esquerda da matriz aumentada. O comportamento da informação (fluxo de dados) nos processos *Forward Elimination* e *Back Substitution* são bem conhecidos assim como os seus respectivos valores

relacionados com a matriz identidade (o lado direito da matriz aumentada). Devido ao fato de que estes processos levam a matrizes triangulares, todos os valores de zero, do lado esquerdo da matriz aumentada, são substituídos pelos valores que seriam armazenados na parte direita. Para realizar todo o processo de inversão, o processo de normalização foi executado junto com *Forward Elimination*, apenas no final de cada laço. Por exemplo, a figura 3.10 apresenta o processo *Forward Elimination* e a organização da memória RAM, tal como acima explicado, apenas para uma coluna da matriz.

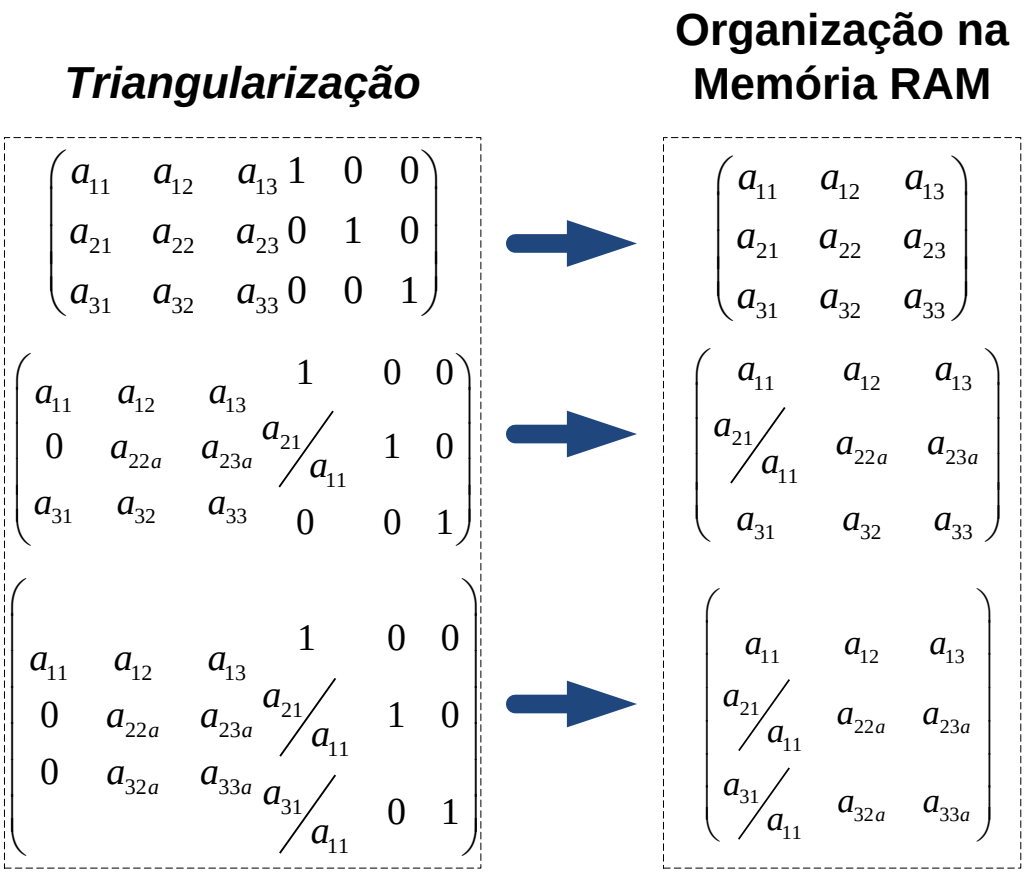


Figura 3.10. Processo de *triangularização* e *organização da memória RAM* para uma coluna da matriz

3.4.2.1 Resultados da implementação

A ferramenta de síntese usada foi o *Xilinx Integrated Software Environment* (ISE) 10.1 e o dispositivo FPGA usado foi Virtex−5 $XC5VLX330T$. Na

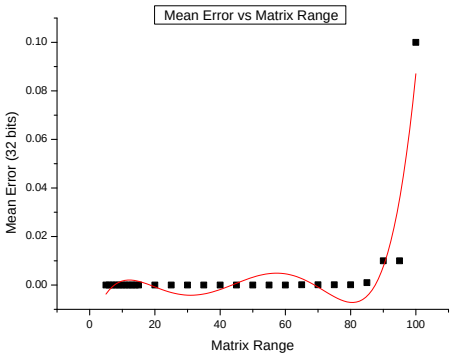
figura 3.11 é mostrado o erro médio para cada umas das precisões usadas: simples, dupla e 40-bits(neste caso, 9-bits para o expoente e 30-bits para a mantissa), com matrizes quadradas de tamanhos $n = 5$ até $n = 120$ usando 1000 amostras para cada n e os resultados foram comparados estatisticamente contra o MatLab. Os dados para cada tamanho de matriz foram gerados usando o comando de MatLab $A = double(rand(x, x))$; que define a matriz com números reais, em dupla precisão com valores entre [0,1].

Pode-se ver a importância de se ter em conta a precisão necessária a ser atingida pela arquitetura segundo os parâmetros da especificação. Assim, a figura.3.11 permite ver a precisão necessária para poder alcançar os requisitos ou restrições dados pelas especificações e também conseguir realizar a inversão da matriz com tamanhos desde 5 até 120. Esses resultados mostram a necessidade de se escolher a precisão conforme o tamanho da matriz a ser invertida, o que pode ser selecionado nesta arquitetura. A seleção da precisão é feita pelo tamanho da palavra e, neste caso, é analisada sua influencia mediante as tabelas de erro associado apresentadas (ver 3.11).

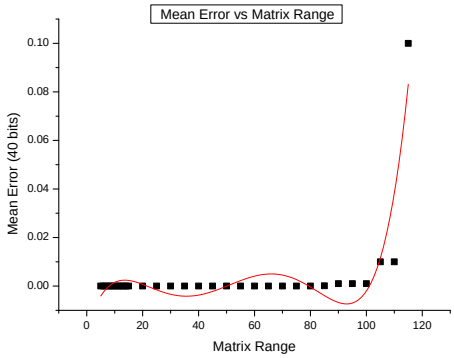
Neste contexto, é importante mostrar que a escolha de um FPGA adequado para uma aplicação deve depender de uma análise em termos dos recursos internos RAM, assim como em termos do número de DSP disponíveis, como consequência da flexibilidade dada ao usuário de mudar a precisão segundo o tamanho da matriz.

As tabelas 3.3, 3.4 e 3.5 resumem os recursos ocupados na implementação por cada módulo e unidade projetada, depois das etapas de posicionamento e roteamento.

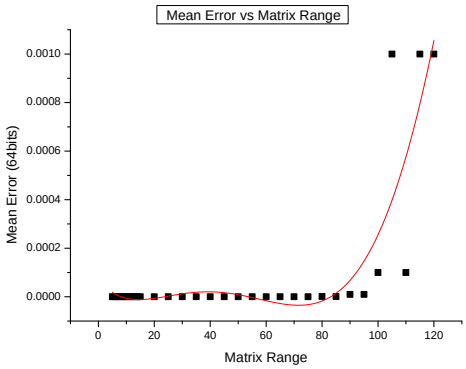
Dado que este dispositivo FPGA tem 192 $DSP48E$ e 324 blocos distribuídos de memória RAM de 36Kb e tendo em conta os recursos ocupados



(a) Precisão simples



(b) 40-bits de precisão



(c) Precisão dupla

Figura 3.11. Erro médio da operação de inversão de matriz com diferentes precisões e com diferentes tamanhos de matrizes.

Tabela 3.3. Resultados de síntese para os módulos principais

Unidade	LUTs	Freq. [MHz]
Troca de Linhas	687(0.32%)	302.414
Encontrar Pivô	893(0.43%)	323.625
Normalização	1042(0.5%)	315.856
Eliminação de Matriz	1233(0.59%)	304.257

da arquitetura, segundo as tabelas 3.3, 3.4 e 3.5), o FPGA usado pode suportar arquiteturas para calcular inversas de matrizes com dimensões maiores.

O número total de memória RAM usado pela implementação em *hardware* da arquitetura depende da precisão. Desta maneira, para uma precisão

Tabela 3.4. Resultados de síntese para a unidade de multiplicação relacionados às diferentes precisões

Precisão	LUTs	Freq. [MHz]	DSP48E
Simples	656(0.31%)	260.484	30(15.62%)
Dupla	1178(0.56%)	211.433	130(67.7%)
40-bits	823(0.39%)	227.894	50(26.04%)

Tabela 3.5. Resultados de síntese para a unidade de soma/subtração relacionados às diferentes precisões

Precisão	LUTs	Freq. [MHz]	DSP48E
Simples	2379(1.14%)	254.647	20(10.41%)
Dupla	4752(2.29%)	200.4	30(15.62%)
40-bits	3058(1.47%)	210.765	20(10.41%)

simples, dupla e 40-bits, os respectivos valores dos blocos de memória RAM usados para armazenar uma matriz de 120×120 são: 60, 120 e 120, respectivamente.

3.5 Arquitetura proposta para solução de sistemas lineares: Uma arquitetura rápida e de baixo custo desenvolvida em FPGAs para resolver sistemas de equações lineares

Nesta seção será descrita a implementação de uma arquitetura para solução de sistemas lineares baseada nos trabalhos [99, 101] dando origem ao trabalho apresentado em [90].

Um sistema linear de equações representado em uma forma matricial pode ser escrito como $Ax = b$. Onde os coeficientes da matriz A assim como o vetor b são conhecidos e os componentes do vetor x são as incógnitas. O método de eliminação de Gaussiana pode ser aplicado apenas quando os componentes da matriz $a_{i,i}$, $i = 1, 2, ..., n$ (usados como divisores) são diferentes de zero. Assim, como requisito da aplicabilidade do método, o

sistema de equações lineares deve ser linearmente independente.

Em muitas aplicações típicas a condição dada em que todos os elementos da matriz $a_{i,i}$, $i = 1, 2, ..., n$ devem ser diferentes de zero é satisfeita pela própria estrutura dos coeficientes da matriz envolvida. Particularmente, esta condição é satisfeita quando a matriz A é simétrica e definida positiva. Ainda se as condições não são satisfeitas o sistema $Ax = b$ pode sempre ser transformado em um sistema equivalente $Bx = g$ com uma matriz simétrica e definida positiva B [102]. Isto pode ser feito por uma simples transformação onde ambos os lados do sistema $Ax = b$ são multiplicados pela transposta da matriz A^T (vide equação 3.6).

$$A^T Ax = A^T b \tag{3.6}$$

Assim, um sistema $Bx = g$ é obtido onde $B = A^T A$ e $g = A^T b$. Esse novo sistema é equivalente ao original $Ax = b$, o que significa que ambos têm as mesmas soluções. Pode ser facilmente demostrado que a matriz $B = A^T A$ do novo sistema é sempre simétrica e definida positiva, e assim o método de eliminação pode ser aplicado sem problemas [102]. Por fim, o processo de *forward elimination* e *back substitution* são respectivamente executados.

3.5.1 Arquitetura de solução de sistemas lineares

A estrutura completa da arquitetura é apresentada na figura. 3.12 onde pode ser observada uma única diferença, em relação à estrutura mostrada na figura.3.2, chamada de MAU (unidade de acesso à memória) sendo esta apenas uma unidade de acesso à memória. Os módulos restantes

assim como as diferentes unidades seguem as descrições apresentadas em 3.4.1 e 3.4.2.

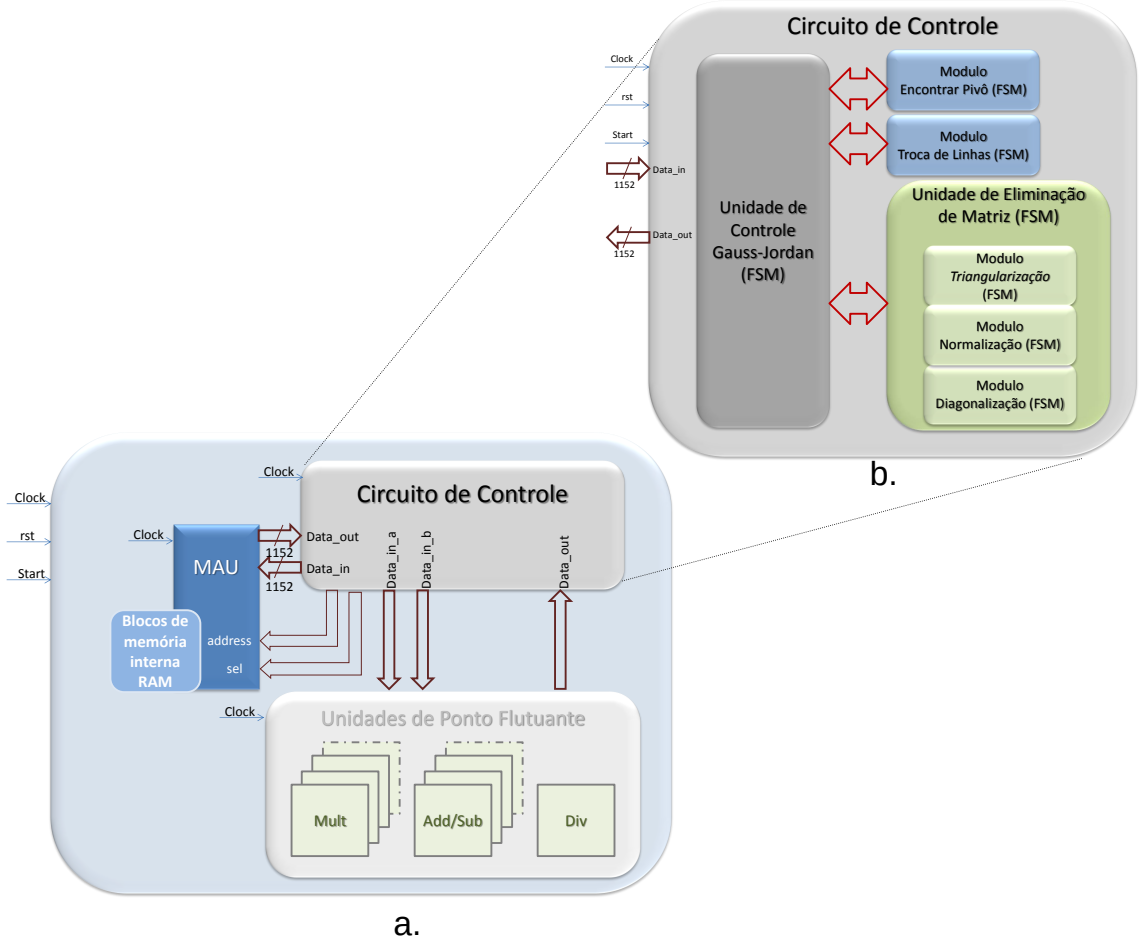


Figura 3.12. Estrutura geral da arquitetura proposta

Para implementar esta arquitetura (delineando o problema de acesso a memória, passos de *leitura/escrita desde/para* a memória RAM), uma unidade de acesso a memória específica foi desenvolvida (MAU) (vide figura.3.13). Esta MAU inclui três módulos: (1) *divisor de sinal*, (2) *seletor de sinal* e (3) *ordenar os endereços*. Estes módulos servem também como um *wrapper* dos blocos de memória internos e incluem n memórias que representam o número total de colunas da matriz B. O *módulo divisor do sinal* e o *módulo seletor de sinal* têm como tarefa *receber/enviar* os dados *desde/para* os blocos de memória internos em uma ordem especifica e, ordenar a tarefa de enviar a cada memória o seu correspondente endereço.

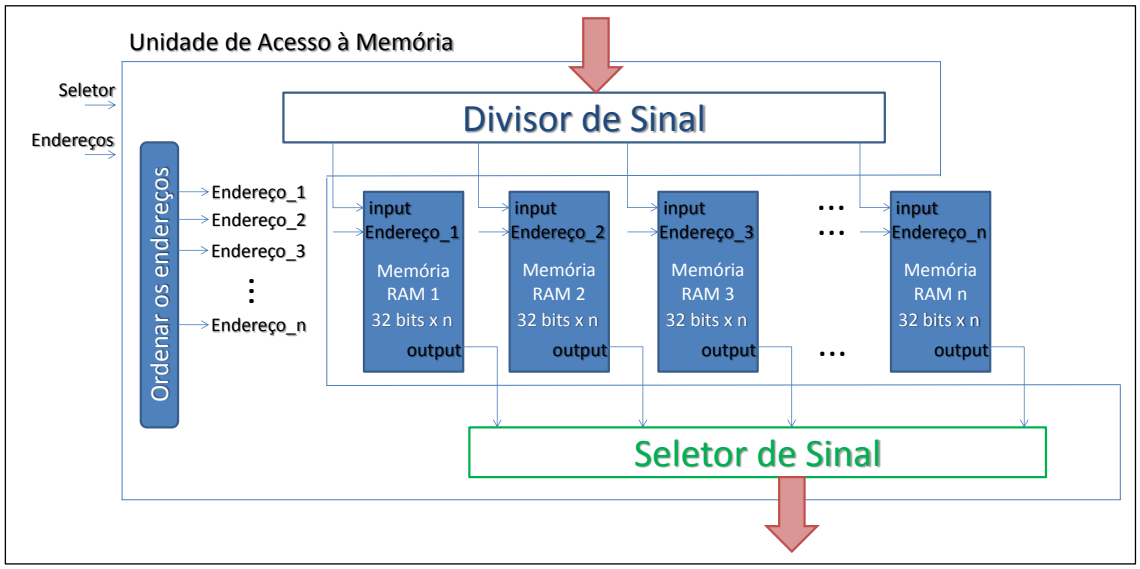


Figura 3.13. A MAU (unidade de acesso à memória) para solução de sistemas lineares

3.5.2 Resultados da implementação

Neste caso, o dispositivo FPGA usado é a Virtex-5 $XC5VLX330T$ junto com o seu *Xilinx Integrated Software Environment* (ISE) versão 10.1. A escolha do dispositivo neste caso, o Virtex5, foi devido à disponibilidade do kit no nosso grupo. Os dados foram gerados da mesma forma como explicados na seção 3.4.1.2. A figura. 3.14 apresenta o tempo em micro segundos (us) usado pela FPGA para obter as soluções do sistema linear de equações (e a inversa da matriz) usando um *clock* de 100Mhz.

Os recursos totais usados pela arquitetura, configurada para um sistema de 10 equações, são apresentados na tabela. 3.6.

Tabela 3.6. Resultados de síntese para a arquitetura solução de sistemas lineares configurado para $n = 10$

Unidade	LUTs	DSP48Es	Freq. [MHz]
Solução de Sistemas Lineares	9441(4.55%)	40(20.83%)	200

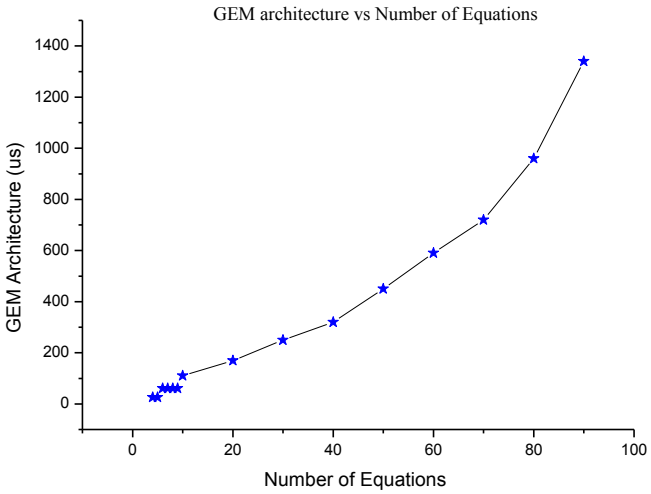


Figura 3.14. Tempo execução da arquitetura solução de sistemas lineares em (μs) para diferentes números de equações

3.6 Simulação baseada em um laço em *hardware* (*Hardware in the loop* - HIL) de um módulo em FPGA para Solução de Sistemas Lineares utilizado em aplicações com sistemas fortemente ligados

As simulações e resultados aqui apresentados foram tomados do trabalho [103]. Atualmente os fornecedores de CPUs estão fazendo um grande esforço para integrar múltiplos processadores em um único chip para lidar com as dificuldades dadas pela lei de Moore, sem a indevida dissipação de energia. Neste contexto, esses processadores são conhecidos como MPSoCs (*multiprocessor system-on-chip*). Alguns desses múltiplos processadores envolvem unidades de processamento gráfico (GPUs) e/ou FPGAs, trabalhando em um processador de proposito geral (GPP). Portanto, um enfoque de *co-design em software/hardware* baseado em FPGA é um meio cada vez mais popular para ajudar o desempenho dos processadores de proposito geral (GPPs), executando tarefas intensas e complexas, trabalhando como um acelerador de *hardware* anexado. As FPGAs junto com outros aceleradores (por exemplo, DSP - processadores digitais de sinais) podem processar tarefas dadas pelos GPPs, enviando os resultados de volta

quando a tarefa seja completada [104].

Por um lado, a programação das FPGAs usando uma linguagem de descrição de *hardware* (HDL) é muito demorada e necessita pelo menos um conhecimento em design de circuitos, o que torna a aplicação desta tecnologia difícil. Ao longo da última década, os pesquisadores vem fazendo grandes esforços para gerar uma nova classe de ferramentas (Programas de alto nível) e linguagens para o design de FPGAs que permitam ao usuário final trabalhar sem lidar com a complexidade dos HDLs. Uma dessas ferramentas é o *Xilinx System Generator Tool* (XSG), o qual é um *software* de alto nível que permite o uso do ambiente MatLab/Simulink para criar e verificar os designs de *hardware* para as FPGAs da Xilinx [105].

3.6.1 Processo de Verificação funcional e HIL com XSG

No intuito de desenvolver uma metodologia de verificação funcional, foram obtidos os resultados da solução de um sistema linear de 6 equações. Nesta prova, o modelo de referência foi o conjunto dos resultados obtidos no MatLab. Para alcançar ambos os resultados de simulação assim como os resultados de *hardware* (executando na placa com dispositivo FPGA Virtex-5 $XC5VLX110T$) uma arquitetura desenvolvida em VHDL (em ISE 13.3 da Xilinx) foi provada. Para realizar isto foi desenvolvido um bloco de *hardware co-simulação* usando Simulink e XSG.

Hardware co-simulation (é o nome usado pela Xilinx para se referir a simulação HIL) é uma placa de desenvolvimento de fiação virtual executando em um PC, permitindo ao usuário colocar a sua arquitetura na ferramenta para realizar as simulações. Depois de terminar as configurações e lograr a interface JTAG, o modelo gera uns arquivos específicos do tipo HDL e *netlists*.

A figura. 3.15 apresenta a arquitetura do bloco *solução de sistemas lineares* em XSG junto com os seus blocos FIFO de compartilhamento, permitindo assim a interface com o MatLab usando código M. Neste lugar, o MatLab endereça as informações correspondentes às linhas da matriz. Por exemplo, a primeira operação de escrita corresponde a escrever $a_{i,1}$ (para i = 1 até 6) dados, seis operações de escrita são executadas para toda a operação de escrita da matriz (vide figura 3.15). Adicionalmente, pode ser observado que seis linhas de controle foram introduzidas pela ferramenta XSG (ver as linhas de retroalimentação com os blocos *assert*, as quais registram a informação tais como a taxa de amostragem).

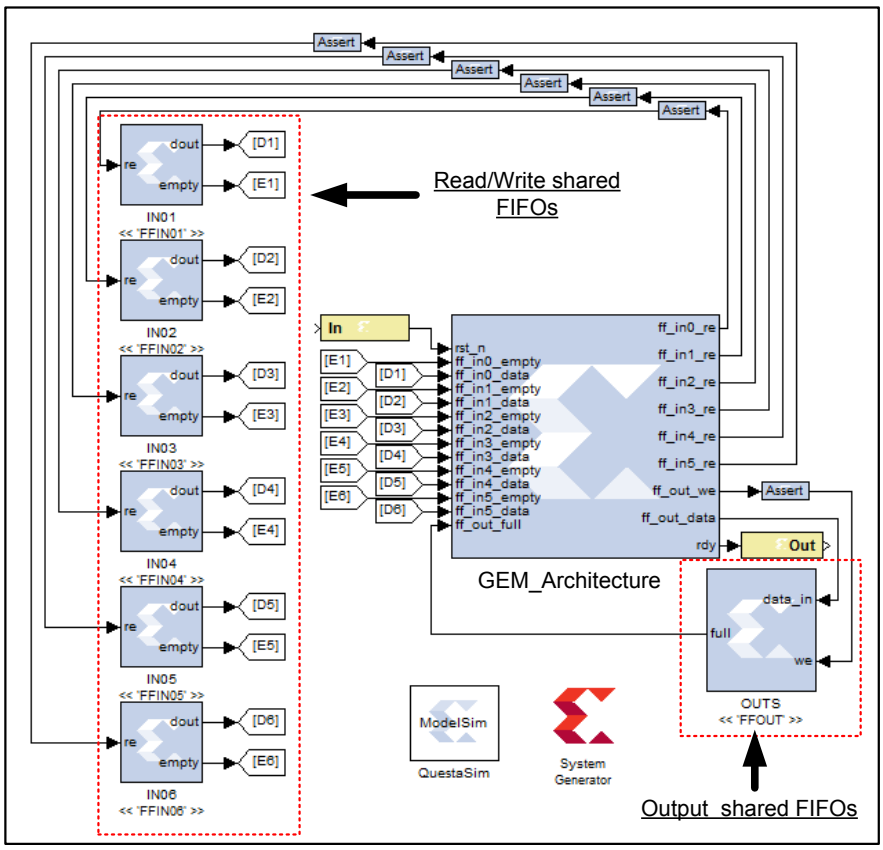


Figura 3.15. Simulação em *software* do bloco SSL com MATLAB usando XSG

O sinal *In* representa o sinal *resete* do SSL (bloco *solução de sistemas lineares*) (vide figura 3.17), enquanto que o sinal *out* representa o sinal *ready* de toda a arquitetura (ver figura 3.17). O FIFO compartilhado de saída é usado pelo bloco SSL para as operações de escrita dos resultados

além de permitir a Matlab ler os resultados. Pode ser observado que o bloco SSL controla a saída do FIFO compartilhado, usando uma linha *enable* para a operação de escrita (usando também um bloco *assert*).

A figura 3.16 descreve toda a arquitetura com a interface para o Simulink, na qual seis FIFOs de compartilhamento foram introduzidos permitindo ao Simulink as operações de escrita para todo o bloco SSL. Adicionalmente, um FIFO de compartilhamento é usado para executar as operações de leitura dos resultados dados pelo bloco SSL. A arquitetura toda assim como o fluxo de *design* permite avaliar o sistema em tempo real.

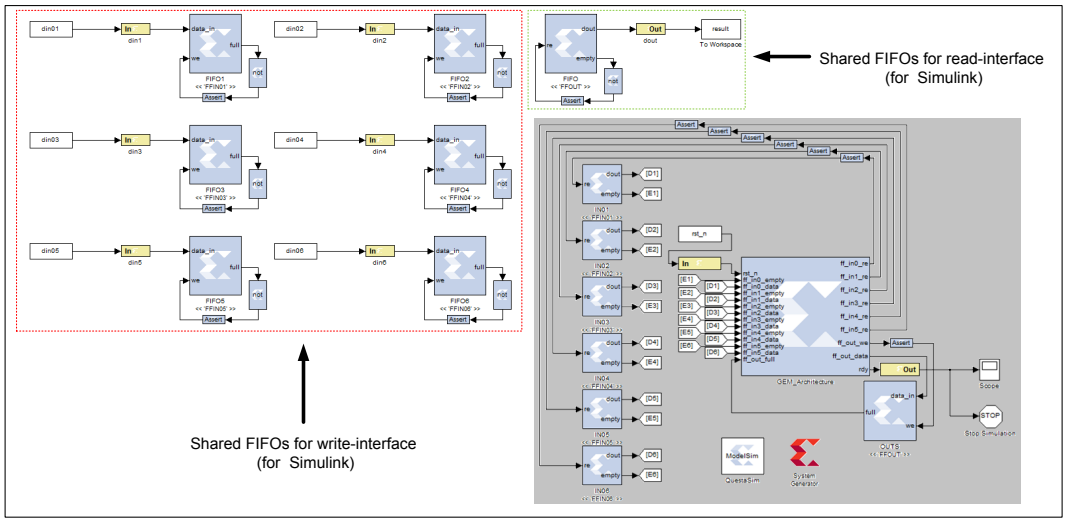


Figura 3.16. Simulação em *Software* do bloco SSL com Simulink usando XSG

O XSG permite também baixar o *bitstream* e a comunicação com o *design* executando por meio de um conjunto de funções que podem ser diretamente chamadas desde o código M em Matlab. Por fim, uma nova biblioteca foi criada e um bloco *hardware co-simulação* foi adicionado, ver figura. 3.17.

Quando um bloco *hardware co-simulação* é adicionado o *chip* de configuração e a transmissão de dados é completada. No nosso processo de *co-simulação* as entradas (que são representadas como matrizes) são geradas desde o espaço de trabalho do Matlab. A inversa da matriz assim como

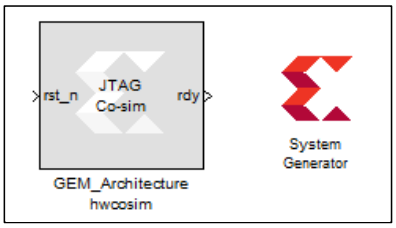


Figura 3.17. *Hardware co-simulation* do bloco SSL com MATLAB/Simulink usando XSG

a solução do sistema linear denso são enviadas de volta e apresentadas no mesmo espaço de trabalho. Esses dados da matriz são transmitidos para o FPGA (desde o espaço de trabalho de MATLAB) através da conexão JTAG, e os resultados são alimentados de volta para o espaço de trabalho de MATLAB usando a mesma interface.

O bloco SSL foi verificado comparando seus resultados para uma precisão simples em ponto flutuante do sistema de 6 equações lineares, ver [103], e o total de recursos usados pela arquitetura toda para esse sistema linear é apresentado na figura.3.18. O bloco *Resources Estimator* do XSG foi usado para ler os valores dos recursos reportados pelo *xflow* no *Xilinx ISE* 13.3.

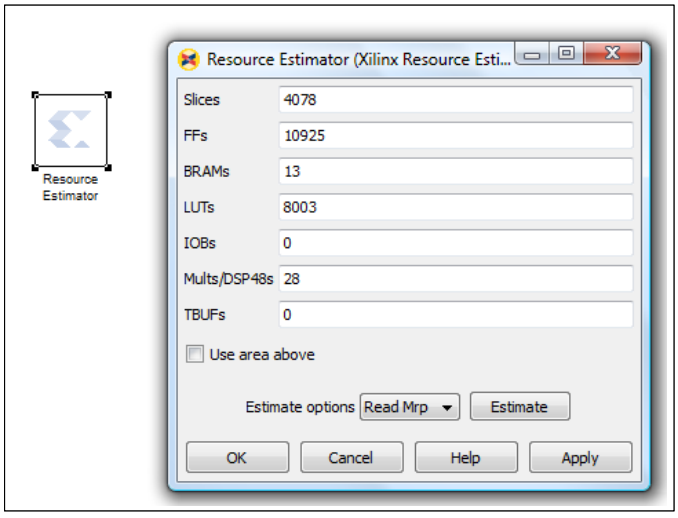


Figura 3.18. Resultados de síntese para a arquitetura SSL com $n = 6$

Os blocos RAM ocupados pela implementação de toda a arquitetura dependem da precisão e o número de equações. Assim, para uma precisão

simples os blocos RAM usados foram treze (esse dispositivo especifico tem 148 blocos RAM organizados como blocos de 36kb).

3.7 Aplicações do bloco SSL

Em colaboração com uma equipe do laboratório de eletrônica do GRACO na UnB, foi elaborado em conjunto um trabalho para acelerar as predições da estrutura tridimensional das proteínas, dando como resultado o trabalho [106]. Neste trabalho foi proposta uma implementação em *hardware* de uma rede neural tipo GMDH e seu uso para predizer de forma aproximada a estrutura tridimensional das proteínas. O bloco SSL foi usado especificamente para acelerar o método dos mínimos quadrados. Ambos, o *hardware* GMDH e o *hardware* SSL, foram escritos em VHDL e provados no dispositivo FPGA. Nesta seção é descrita a parte relacionada com o bloco SSL deixando ao leitor uma revisão da implementação do GMDH na referencia [106].

3.7.1 *Co-verification software-hardware*

Os projetos desenvolvidos neste trabalho foram testados usando o fluxo de projeto empregando o *socket* de comunicação TCP/IP entre o PC e o FPGA onde o GMDH e o SSL foram programados. Enquanto o neurônio GMDH foi realmente programando no FPGA, o SSL executa na ferramenta de simulação ModelSim na qual a comunicação foi captada através de um modelo de nível de transação. Esta captação foi alcançada usando uma biblioteca de código M que lida com os detalhes da comunicação do desenho assim, isolando o treinamento do GMDH e os algoritmos de execução da complexidade envolvida nessas operações [107].

Para desenvolver essa comunicação TCP/IP foi usado um *soft-processor*, o MicroBlaze. O neurônio GMDH e o SSL foram usados como periféricos do processador, ligados ao barramento PLB, como descrito na figura 3.19. Isto foi possível devido ao uso de *wrappers* em VHDL que encapsulam a lógica do usuário como um periférico no MicroBlaze. O processo de criar tais periféricos é assistido pelo *Xilinx Platform Studio integrated environment* (XPS) [107].

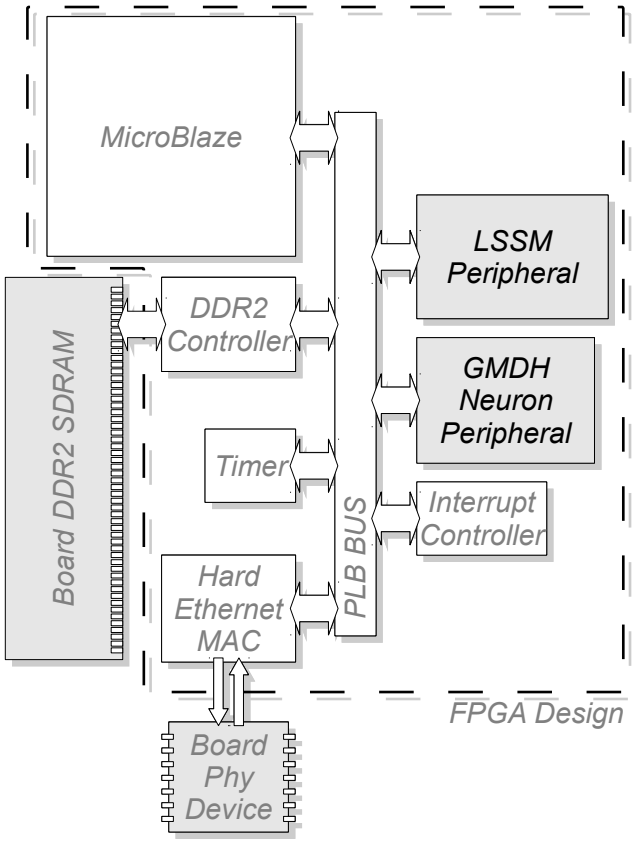


Figura 3.19. Design do neurônio GMDH e do bloco SSL como periféricos do *MicroBlaze soft-processor* anexo ao barramento PLB. [106]

Quando o método de treinamento precisa dos cálculos dos pesos dos neurônios, a biblioteca de código M chama de maneira transparente o processo de mínimos quadrados desenvolvido pelo bloco SSL simulado na ferramenta ModelSim [107].

3.7.2 O bloco de solução de sistemas lineares (SSL)

Para se ajustar aos requerimentos deste projeto, o bloco SSL foi adaptado para calcular os coeficientes de neurônio GMDH, resolvendo um sistema de 6 equações e 6 incógnitas. A figura 3.20 apresenta o bloco SSL contendo: (a) o bloco RAM onde os elementos da matriz são armazenados; (b) a unidade de eliminação Gaussiana (GEU) que iterativamente escreve e lê desde/para o bloco RAM até que todas as incógnitas são encontradas; (c) o conjunto de controladores FIFO, responsáveis pelas conexões com os 6 FIFOs linha e com o FIFO de saída e; (d) uma máquina de estados finita (FSM) responsável pela coordenação da transferência de dados entre os blocos [107].

Os coeficientes do neurônio são calculados usando a equação 3.7. O SSL desenvolve esse cálculo tomando como entradas a matriz formada pela equação 3.8. $[\mathbf{M},\ \mathbf{p}]$ sendo uma matriz 6×7 enviada ao bloco [107].

$$\mathbf{a} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{b} \tag{3.7}$$

$$[\mathbf{M},\ \mathbf{p}] = \left[(\mathbf{X}^T\mathbf{X}),\ \mathbf{X}^T\mathbf{b}\right] \tag{3.8}$$

3.8 Considerações Finais do Capítulo

Neste capítulo foram apresentadas as implementações em *hardware* reconfigurável baseadas no algoritmo de Eliminação de Gaussiana. As arquiteturas conseguem tratar com a função de inversão de matriz assim como

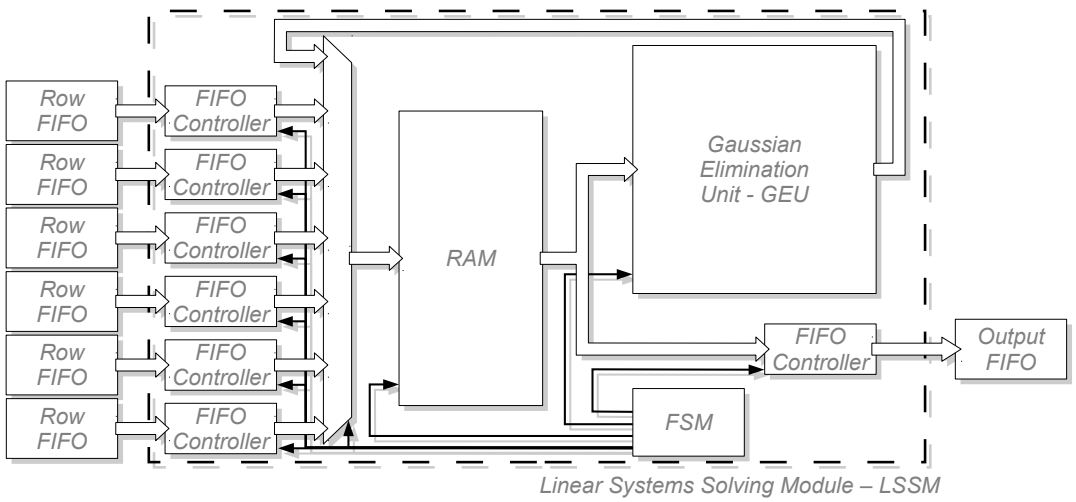


Figura 3.20. Bloco SSL- usado para acelerar o método dos mínimos quadrados. [106]

também a solução de um sistema linear de equações, permitindo a parametrização do sistema com relação ao tamanho da matriz e/ou sistema linear de equações.

Na implementação da arquitetura para inversão de matrizes foram utilizadas representações em ponto flutuante de precisão simples, dupla e customizada de 40-bits. Ao passo que para implementar a solução de um sistema linear de equações foi usada apenas uma representação em ponto flutuante com precisão simples. O sistema pode tratar sistemas lineares como matrizes densas, relacionadas a sistemas fortemente acoplados. A arquitetura foi divida em três fases principais de acordo com o método de eliminação Gauss-Jordan: (a) Eliminação Gaussiana (pivô parcial e *forward elimination*), (b) diagonalização (*back substitution*) e (c) normalização. As três fases são executadas por quatro circuitos internos: (1) Troca de linhas, (2) Encontrar o Pivô, (3) Eliminação da Matriz e (4) Circuito de Controle de Eliminação Gaussiana.

As comparações do desempenho *hardware/software* foram realizadas considerando tanto a arquitetura desenvolvida quanto a execução em *software* do algoritmo de eliminação Gauss-Jordan enquanto o tamanho do sistema

matricial aumentava. A verificação da arquitetura para solução de um sistema linear de equações foi realizada implementando uma arquitetura que suporta um sistema linear de equações de 6×6. Adicionalmente, um analise experimental do erro foi desenvolvido usando como comparador estatístico com precisão dobrada o MatLab.

Os resultados mostram uma tendência do erro a crescer enquanto o tamanho do sistema matricial tratado aumenta em sua dimensionalidade, o que é explicado pelo número de multiplicações necessárias para realizar o processo de inversão e/ou encontrar a solução do sistema linear de matrizes. Implementações do processo de normalização no final assim como no meio dos processos de eliminação de matriz demostraram variações da propagação do erro assim como um aumento na quantidade de multiplicações necessárias na implementação.

Os recursos utilizados (sejam LUT, DSP e RAM), mostram que o dispositivo sobre o qual foi realizada a implementação, neste caso Virtex5, pode suportar matrizes com tamanhos maiores devido ao mínimo consumo de recursos usados pela arquitetura. Esse aumento do tamanho do sistema está condicionado ao número de blocos DSP assim como ao número de blocos de memória interna disponíveis no dispositivo FPGA selecionado.

Para acelerar a verificação funcional e por sua vez, evitar esforços de configuração de interfaces, etc., desenvolvendo uma simulação HIL (*Hardware-in-the-loop*), foi utilizado também o *Xilinx System Generator* anexo ao Matlab/Simulink. A arquitetura foi transformada em um bloco o que permitiu ser usada como qualquer bloco em Matlab/Simulink com a ferramenta XSG em duas formas: como simulação em *software* ou como simulação HIL. Também foi demostrada a aplicação da arquitetura apoiando o co-processamento e a aceleração em *hardware* da implementação de uma rede neuronal tipo GMDH.

Capítulo 4 IMPLEMENTAÇÕES EM *HARDWARE* DAS ARQUITETURAS BASEADAS EM FLUXO DE DADOS

Neste capítulo são apresentadas as principais arquiteturas ou blocos operacionais construídos para lidar com a solução da função raiz quadrada de uma matriz simétrica. Três principais blocos operacionais são descritos: (a)*arquitetura de decomposição QR baseada no método de ortonormalização modificado de Gram-Schmidt*, (b) *arquitetura de multiplicação matriz-matriz* e (c)*arquitetura multiplicação diagonal transposta matriz*. Adicionalmente, para lidar com processos de redução, uma *arquitetura de união multiplicação-soma-acumulação em ponto flutuante* foi desenvolvida (e apresentada também neste capítulo) junto com as arquiteturas *soma/subtração* e *multiplicação* em ponto flutuante. No intuito de facilitar a explicação das diferentes arquiteturas matriciais propostas, as figuras respectivas foram desenhadas implementando apenas matrizes de 4×4.

O capítulo começa com uma descrição da metodologia de verificação validação abordada no desenvolvimento desta arquitetura, seguido da apresentação do fluxo dos dados em ponto flutuante, mostrando os problemas de denormalização/normalização dos dados. São apresentadas também as diferentes arquiteturas de ponto flutuante projetadas (*soma*, *multiplicação*), tomando como base a proposta apresentada em [108]. Adicionalmente, uma seção é dedicada à arquitetura FP-FMAC (*união multiplicação soma acumulação em ponto flutuante*). Por fim, são descritos os principais circuitos desenvolvidos para realizar as operações internas da arquitetura para a raiz quadrada.

4.1 Metodologia usada para Verificação e Validação

O projeto de sistemas embarcados geralmente começa com um conjunto de requisitos aos quais o projeto deve atender, terminando com o produto final, uma vez alcançados todos esses requisitos de funcionamento. A figura 4.1 descreve os passos necessários de verificação no processo de projeto em FPGAs. É importante salientar como a verificação é aplicada em cada etapa, a fim de garantir o correto funcionamento do circuito.

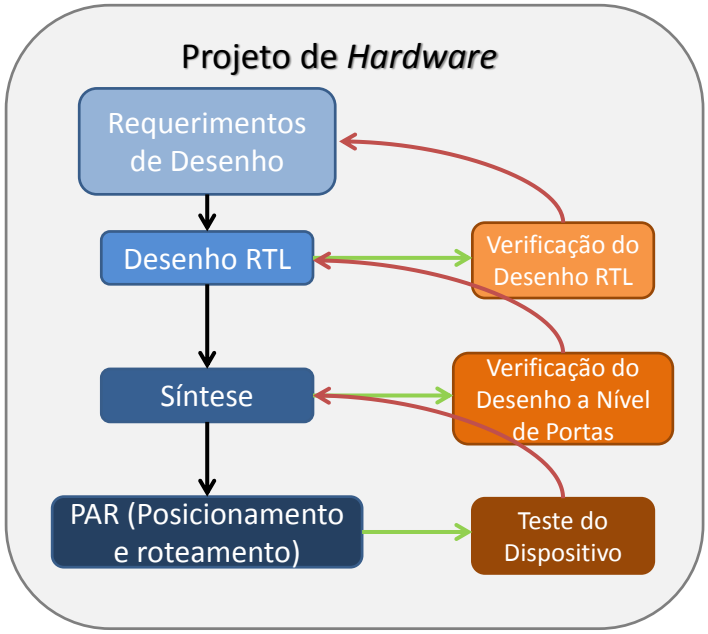


Figura 4.1. Fluxo de projeto e verificação no processo de desenho em FPGAs

A verificação em sistemas embarcados faz referência a ferramentas e técnicas usadas para se certificar de que um sistema não contem problemas de *hardware* ou de *software*. A verificação do *software* destina-se a executar o *software* e observar o seu comportamento; enquanto que a verificação de *hardware* envolve, certificando-se, que o *hardware* se desempenha corretamente, na presença dos estímulos externos ou do *software* sendo executado.

A forma mais antiga de verificação de sistemas embarcados consiste em projetar/implementar o sistema, executando o *software* e torcer pelo me-

lhor resultado. Se por acaso não funcionava, a ideia era tentar modificar o possível tanto em *software* quanto em *hardware* para se ter o sistema funcionando. Desafortunadamente, descobrir o que não está correto, enquanto o sistema está em execução, nem sempre é uma tarefa fácil. Controlar e observar o sistema enquanto ele está sendo executado não pode mesmo ser possível. Ao longo dos anos o fluxo de projeto de sistemas embarcados vem sendo modificado, devido ao incremento na complexidade dos mesmos (ver figura 4.2), isto no intuito de reduzir o *gap* entre os engenheiros de *software* e *hardware* [109].

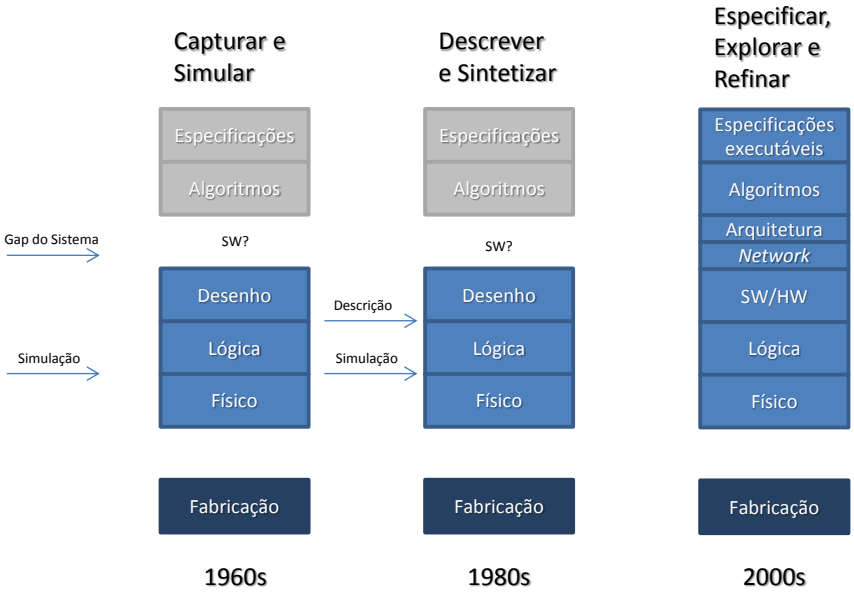


Figura 4.2. Evolução no fluxo de projeto em sistemas embarcados durante os últimos 50 anos. Tomado de [109].

Para lidar com as dificuldades de depuração de um sistema embarcado muitas ferramentas e técnicas foram introduzidas para ajudar os engenheiros a obter sistemas integrados de trabalho mais cedo e de forma mais sistemática. O ideal é que toda essa verificação seja feita antes que o *hardware* seja construído. Quanto mais cedo os problemas de processo são descobertos mais fáceis e mais barato será corrigi-los. Neste caso a verificação responde à pergunta: "Será que o dispositivo que nós construímos vai funcionar"?

Enquanto a verificação em sistemas embarcados objetiva detectar os *bugs* de *hardware* ou de *software* a validação em sistemas embarcados faz uma análise de se o sistema atende ou excede os requisitos. Isto é feito mediante diferentes ferramentas e técnicas, confirmando parâmetros tais como funcionalidade, desempenho, potência. Assim, a validação confirma se (a) a arquitetura proposta é correta e (b) o sistema embarcado apresenta o desempenho ideal.

Algumas projetos se beneficiam da implementação de estruturas de *hardware* complexas com várias características, tais como topologias paralelas ou de *pipeline*. Isto faz difícil uma verificação e, consequentemente, uma validação real dos circuitos. É bem conhecido que uma verificação exaustiva do correto funcionamento da implementação, especialmente de operadores matemáticos, analisando todas as combinações possíveis que permitam avaliar por completo o comportamento do circuito ante quaisquer tipos de entradas, não é aceitável devido ao alto custo computacional que afeta o chamado *time to market*. Por esse motivo, uma verificação funcional dos circuitos é uma parte importante no desenvolvimento do *hardware*, e deve ser realizada de forma consciente e questionando o que deve ser verificado.

No desenvolvimento deste trabalho e, em especial, neste capítulo é mostrada a metodologia usada para verificar funcionalmente (ou verificação do projeto no nível RTL) os circuitos implementados, assim como também é abordada uma verificação ou avaliação no dispositivo mediante ajuda da ferramenta *System Generator* (XSG) da Xilinx, associada ao *Simulink* de MatLab.

4.1.1 Verificação Funcional

A verificação funcional tenta identificar todas as funcionalidades do sistema descrito em HDL (*Linguagem de Descrição de Hardware*), tal como definidas nos requisitos. Esta forma de avaliação é um exemplo do chamado *black-box testing*, no qual são abstraídos os detalhes da implementação do sistema/módulo implementado [110] (ver figura 4.3).

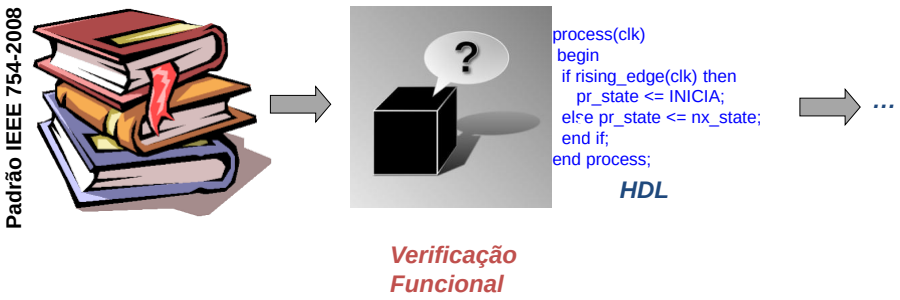


Figura 4.3. Esquema de verificação funcional ou *black-box testing*

Desta forma, é criado um *testbench* analisando o comportamento dos circuitos para diferentes cenários de valores ou intervalos de operação. Estes valores, estarão delimitados pelo conjunto de dados entre $(0, 1)$, onde apenas serão tomados os valores sem uma representação finita, obtendo informação do pior cenário possível para avaliar as arquiteturas, em termos da precisão.

No ambiente de simulação construído no Matlab foi possível gerar os vetores de teste aleatórios e decodificar os resultados. Desta forma, o vetor de dados do *testbench* é gerado com ajuda do MatLab assim como o resultado também será entregue ao MatLab. Isto é feito, com a ideia de realizar uma comparação estatística do resultado obtido pela implementação do circuito com o *script* realizado em MatLab. Assim, a metodologia de verificação funcional abordada no desenvolvimento das arquiteturas de fluxo é como mostrada na figura 4.4.

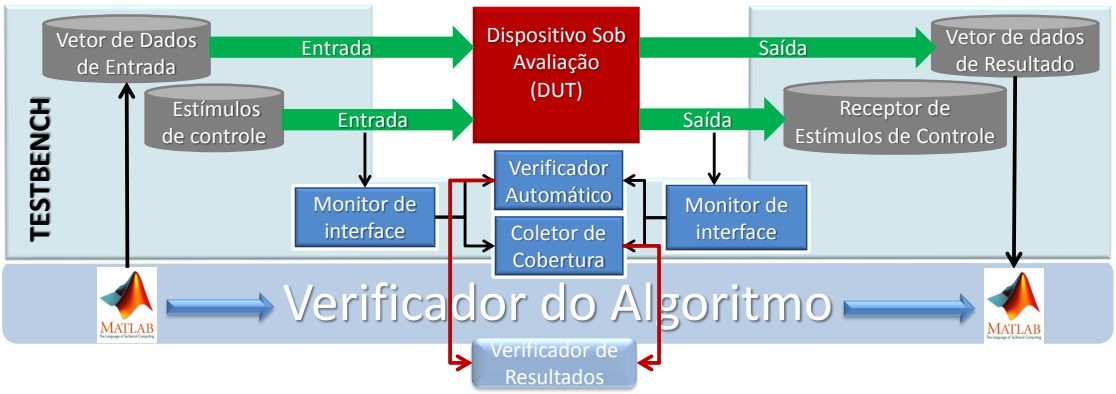


Figura 4.4. Metodologia de verificação funcional abordada no desenvolvimento das arquiteturas de fluxo de dados

A figura 4.4 mostra como é utilizado o *testbench* para capturar os valores, tanto do resultado das operações matemáticas como dos valores de controle internos do circuito. Por outro lado, é realizada uma verificação do seu funcionamento, usando para este fim simulações com a ferramenta para trabalhar no nível RTL *Questa Advanced Simulator* 10,2 da *Mentor Graphics*. As análises de síntese, desempenho e resultados de precisão foram registrados para cada caso.

Em termos da precisão das arquiteturas, foram divididas em dois cenários, ambos utilizando apenas uma precisão de 32 *bits*:

- O primeiro cenário é utilizado para operações aritméticas, sejam estas: soma/subtração, multiplicação, etc. Neste cenário são introduzidos valores aleatórios delimitados pelo conjunto de valores entre $(0, 1)$, com a condição que apenas são introduzidos números irracionais tal como comentado anteriormente. Com isto, uma análise quantitativa do erro associado foi realizada a partir das medidas do MSE (erro quadrático médio) e do MAE (erro absoluto médio), usando 1000 amostras para cada tamanho de palavra, sempre utilizando o MatLab como estimador estatístico configurado em 64 *bits*.

- O segundo cenário é utilizado para avaliar a precisão nas arquiteturas de operações matriciais, onde os dados de cada matriz (para várias dimensões partindo de 4×4 até 100×100) foram gerados usando o comando Matlab A = double (hilb (n)), que define as matrizes de Hilbert com números reais (em precisão dupla, ou seja, $a_{ij} = 1/(i+j-1)$) e direcionados para as arquiteturas a fim de avaliar sua precisão, no pior caso. Desta maneira, uma análise quantitativa destas arquiteturas também é realizada utilizando, neste caso, a norma de *Frobenius*, tal como mostrado na equação 4.1,

$$\|E\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|e_{ij}|^2}, \tag{4.1}$$

onde E representa o erro normalizado dado pela equação de Frobenius, m e n são as dimensões da matriz dada de $m \times n$, e e representa o erro associado ao resultado do resíduo entre os valores calculados pela arquitetura e os valores dados pelo MatLab.

4.1.2 Verificação em *Hardware* com Xilinx System Generator

O Xilinx System Generator é uma ferramenta de alto nível, que está associada ao Simulink do MatLab permitindo criar um ambiente de simulação multi-domínio e um projeto, baseado em modelos, o qual é direcionado para sistemas dinâmicos e embarcados de processamento digital de sinais de alto desempenho em FPGAs. A ideia é facilitar o uso de FPGAs evitando assim a descrição de circuitos, mediante o uso de HDLs, colocando blocos com *simulink*. A ferramenta XSG contém vários blocos com funções predefinidas e também possibilita o uso de funções próprias como caixas

pretas. O mesmo é simples de usar e permite a simulação e o uso de *test-benches*, assim como pode gerar o *bitstream*, a descrição em HDL e um *hardware co-simulation*.

Esta ferramenta é muito apropriada quando é desejado realizar rápidas verificações em *hardware*, evitando assim lidar com o problema das interfaces de comunicação, pois evita essas configurações pela implementação de uma comunicação JTAG associada com cada placa padrão fornecida pela Xilinx. A figura 4.5 apresenta como são realizadas as simulações usando esta ferramenta. Ambas as simulações são possíveis, tanto a simulação funcional quanto o *hardware co-simulation*.

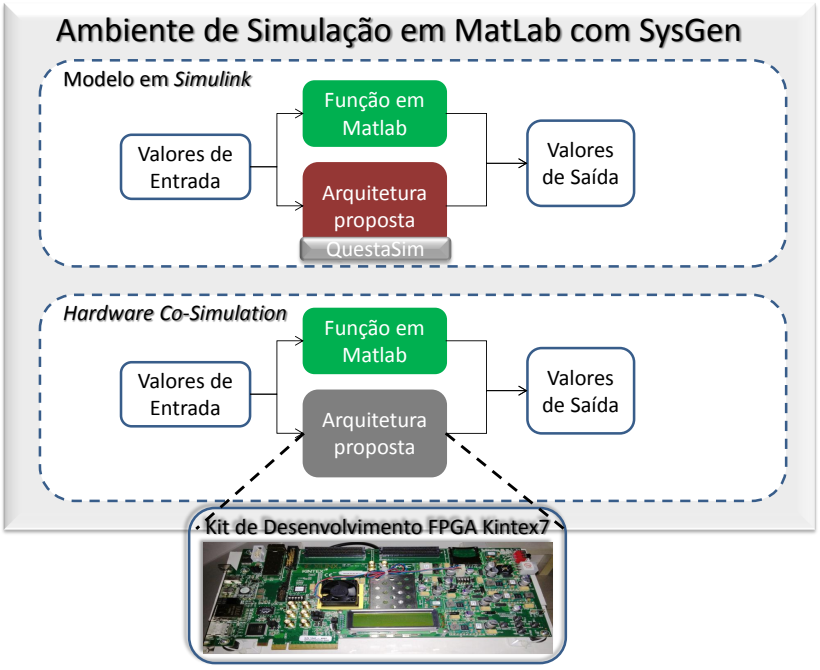


Figura 4.5. Metodologia de verificação em *Hardware* com XSG abordada no desenvolvimento das arquiteturas de fluxo de dados

Mediante o uso desta simulação, pode-se observar o real funcionamento das arquiteturas executado no FPGA e explorar as possibilidades de comparação que há com a ferramenta.

4.1.3 Validação das Arquiteturas

Em todos os casos de validação das arquiteturas aritméticas é analisado o impacto do formato utilizado pelo padrão IEEE-754 (tamanho de palavra) no erro associado, assim como a influência no consumo de recursos lógicos e os diferentes parâmetros de desenho, tais como consumo de LUT, Flip-Flops, DSP, Frequência, etc. No caso das arquiteturas matriciais, apenas foram consideradas arquiteturas com uma representação em ponto flutuante de 32 bits, sendo que o principal fator em análise é sua propagação de erro durante a execução dos cálculos. Adicionalmente, são apresentados valores quantitativos de consumo de recursos lógicos assim como valores alcançados dos parâmetros de projeto associados.

4.2 Fluxo dos Dados em Ponto Flutuante

Obter a solução de sistemas lineares (ou de funções matriciais) envolve não apenas inversões de matrizes, mas também algumas classes de decomposições. Adicional ao alto custo computacional requerido, essas técnicas podem apresentar problemas de estabilidade numérica, se não for usado um intervalo dinâmico de operação adequado. Por tanto, uma implementação eficiente (e com precisão de tais algoritmos) é apenas válida usando dispositivos com representação em ponto flutuante, permitindo assim obter os resultados com a precisão necessária.

Algumas operações sobre matrizes, como o caso das decomposições de matrizes supracitadas, são usadas em diferentes aplicações de radares militares avançados tal como o STAP (*Space-Time Adaptive Processing*) e em vários problemas de estimação em comunicações digitais, etc. A decomposição QR é comumente usada em casos de matrizes A de tamanho $m \times n$,

onde essa decomposição usa algoritmos computacionais custosos, os quais requerem uma alta precisão, fazendo da matemática de ponto flutuante uma necessidade real.

Tradicionalmente, os FPGAs não têm sido as plataformas escolhidas para tratar aplicações que demandam operações em ponto flutuante [111]. Embora, os fornecedores de FPGAs oferecem bibliotecas de operações em ponto flutuante, as mesmas apresentam um desempenho bem limitado quando utilizadas. A ineficiência dos projetos tradicionais em ponto flutuante no FPGA é parcialmente devida à profunda natureza do *pipeline* e às amplas estruturas aritméticas dos operadores em ponto flutuante, os quais criam *datapaths* com longas latências e congestionamento de roteamento. Para estes casos, o resultado final são projetos com uma frequência de operação baixa.

O fluxo de dados típico das operações em ponto flutuante é mostrado na figura 4.6, onde se tem que os dados passam por um processo de denormalização, antes mesmo de ser realizada a operação em questão. Após a operação ser feita, os dados são novamente normalizados. Observe que em uma operação normal o dado é antes arredondado e post-normalizado para, finalmente, ser enviado como resultado final. No intuito de atacar esse problema, a operação de redução foi tratada utilizando uniões entre as operações de multiplicação, soma e acumulação, em vez de realizar o processo por separado, reduzindo assim o caminho e a profundidade dos *pipelines*.

Isto é feito mediante uma análise do incremento em bits dos dados no *datapath*, escolhendo a melhor forma de normalização a fim de se introduzir uma precisão suficiente, eliminando ao máximo os passos de normalização e denormalização, tal como mostrado na figura 4.7. O formato IEEE 754 é apenas usado como limite no *datapath*, onde mantissas de tamanho

Figura 4.6. Fluxo de dados típico das operações em ponto flutuante

diferente são usadas para incrementar o intervalo dinâmico, e reduzir assim a necessidade dos passos de denormalização e normalização entre as operações sucessivas.

As funções de normalização e de denormalização usam deslocadores de bits (*barrel shifters*) de até 48 bits de comprimento para números em ponto flutuante de precisão simples. Quando o número em ponto flutuante tem o bit mais significativo da mantissa diferente de zero a representação é dita normalizada. Para evitar múltiplas representações para o mesmo valor, o número deverá ser denormalizado/normalizado mediante deslocações da mantissa. É comum, neste caso, analisar o consumo de multiplicadores, os quais estão disponíveis na FPGA, como neste caso. Por exemplo, em dispositivos Xilinx da família 7 têm-se multiplicadores de 25×18. Isto consome uma quantidade significativa de recursos lógicos e de roteamento, sendo a principal causa da ineficiência das implementações em ponto flutuante sobre FPGAs. Abordando uma metodologia de união de *datapath*,

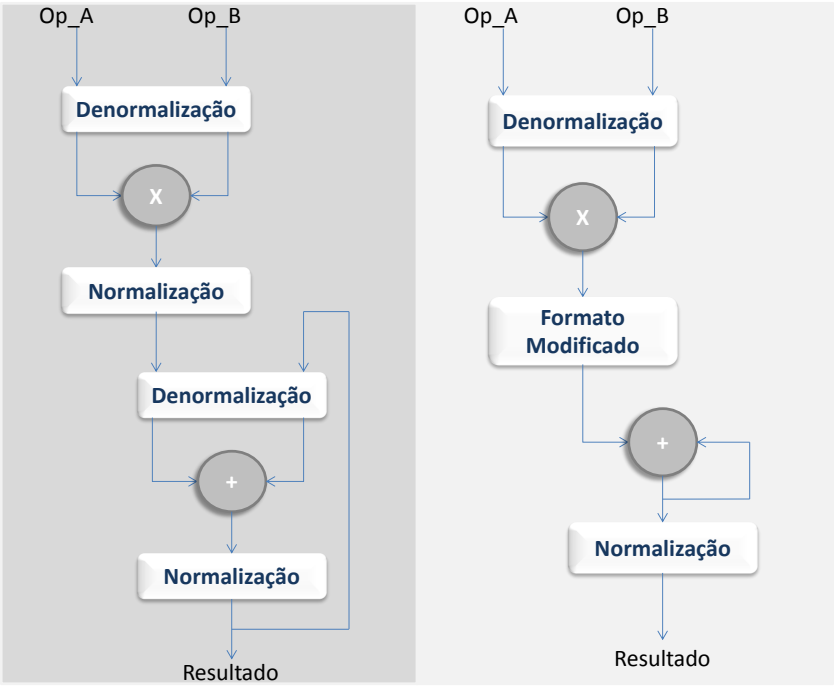


Figura 4.7. União de duas operações em ponto flutuante (neste caso a multiplicação e a soma); lado esquerdo: *datapath* típico, lado direito: *datapath* modificado evitando múltiplos processos de denormalizações/normalizações

pode-se eliminar de forma significativa a quantidade de *barrel shifters* e, assim, evitar o consumo de recursos lógicos e de roteamento utilizados na implementação dessas operações.

No lado direito da figura 4.7 pode ser observada a união do *datapath* para a operação conjunta de multiplicação soma acumulação, em um único bloco comparada com uma metodologia tradicional mostrada no lado esquerdo. A união do *datapath* do lado direito da figura 4.7 elimina a redundância do processo intermediario, removendo a normalização na saída do multiplicador e a denormalização na entrada do somador. Da mesma maneira, a estratégia elimina a necessidade de normalizar/denormalizar para realizar a acumulação. A eliminação de lógica e do roteamento extra, assim como do uso de multiplicadores em silício, faz que o tempo e a latência entre os *datapath* seja previsível.

Essa união do *datapath* é realizado modificando internamente o formato utilizado, seja baseado em uma nova representação ou, como no caso aqui utilizado, aumentando o tamanho da representação da mantissa para obter maior precisão.

4.3 Arquiteturas de Soma/Subtração e Multiplicação em Ponto Flutuante

Atualmente, ferramentas de desenvolvimento tais como Xilinx Vivado (ou ISE), Altera Quartus, Synopsys Synplify, etc., disponibilizam uma grande biblioteca de blocos de propriedade intelectual (ou *IPcores*), encontrando-se tanto unidades de cálculo matricial quanto unidades de cálculo aritmético em ponto flutuante. As ferramentas permitem uma grande flexibilidade enquanto à forma de uso desses *IPcores*, possibilitando alterações em precisão, consumo de DSP ou até latência, entre outras características. Porém, as librarias são geralmente fechadas sem permitir o acesso à descrição de *hardware*, o que torna impossível efetuar modificações na sua lógica para adequá-las nas especificações impostas.

Dada a necessidade pelo controle interno das diferentes etapas ou fases dentro das operações aritméticas e no intuito de melhorar a propagação do erro, assim como a latência dessas operações, foram propostas diferentes modificações nas bibliotecas desenvolvidas em [108, 112]. Neste caso, as bibliotecas foram modificadas para lidar com fluxos de dados em vez de tratar com um fluxo controlado por FSM. Várias especificações foram mantidas constantes em relação às bibliotecas propostas (ver [108, 112]), tais como (a) descrição de *hardware* genérica que possibilite a portabilidade de plataformas; e (b) arquiteturas parametrizáveis que permitam a mudança no tamanho de palavra viabilizando uma análise de erro associado.

Assim, baseado nas bibliotecas de ponto flutuante apresentadas em [108,

112], as estruturas principais foram projetadas e configuradas como arquiteturas de fluxo de dados, aceitando e operando dados em *pipeline*. Cada arquitetura foi estudada para permitir o seu posterior uso e fusão como um único bloco operacional em ponto flutuante. Assim, duas ou três operações em ponto flutuante sequenciais podem ser fusionadas em um único bloco, evitando assim normalizações/denormalizações extras e, intrinsecamente, a propagação do erro por duplo arredondamento.

Os resultados apresentados das arquiteturas desenvolvidas foram validados apenas para dois diferentes tamanhos de palavra 32 e 64 bits, obtendo dados de consumo de recursos, desempenho, frequência de operação e, erro associado. Sendo este último o parâmetro mais relevante no desenvolvimento deste trabalho, por causa da condição atribuída para as matrizes; de serem sistemas mal condicionados.

4.3.1 Arquitetura de Soma/Subtração

Entre os principais circuitos de operações em ponto flutuante se encontra a soma/subtração, sendo este muito difícil de implementar de forma ótima devido à sua complexidade arquitetural [113]. As etapas intrínsecas que envolvem o processo de soma em ponto flutuante apresentam uma grande complexidade onde nem sempre quebrar os passos em processos menores é simples, e por sua vez obter uma arquitetura baseada em fluxo de dados.

As etapas do processo de soma começam com a separação dos operandos de entrada em seus diferentes partes, (a) o sinal, (b) o expoente e (c) a mantissa. Seguido, são conferidas se as entradas dos valores são zero, infinito ou algum número com representação diferente e não válida do padrão IEEE−754. Caso os valores sejam válidos, um bit implícito '1' é inserido às mantissas. Desta forma, são comparados os dois operandos e deslocado

à direita o menor dos números. Seguidamente, é subtraído o número de bits deslocado do exponente, gerando um resultado preliminar do mesmo. Adicionalmente, as mantissas são somadas ou subtraídas dependendo da operação desejada [112].

Por fim, a normalização é feita caso seja necessário, analisando se o bit mais significativo do resultado da mantissa for '1'; caso contrário, o resultado da mantissa é deslocada à esquerda até encontrar o valor de '1' no bit mais significativo. Além disso, para cada bit deslocado é subtraído '1' do valor do expoente, e finalmente, são concatenados os resultados do sinal, expoente e mantissa, obtendo assim o valor desejado [112].

Na figura 4.8 é apresentado o diagrama de blocos da arquitetura do *FP-Add*. Duas partes compõem a arquitetura: (a) o bloco comparar e alinhar a mantissa (CAM) e (b) o bloco de normalização da mantissa (NM). Esses dois blocos estão projetados para trabalhar como uma arquitetura de fluxo de dados, permitindo a saída de um novo resultado a cada ciclo de *clock* após uma latência de dois ciclos necessários para preencher os processos da arquitetura.

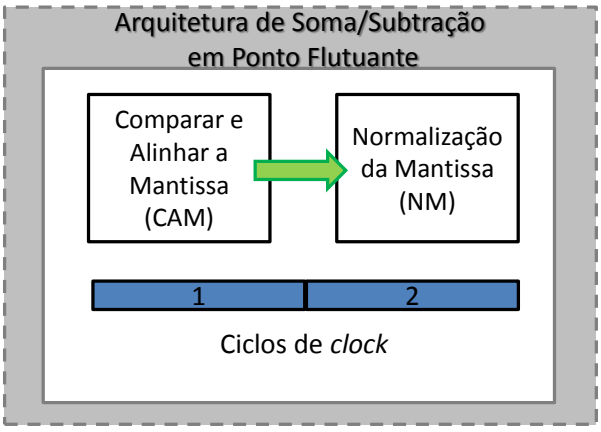


Figura 4.8. Diagrama de blocos da arquitetura FP-Add

O bloco CAM, permite comparar vários aspectos na entrada dos dados, tais como o dado com maior valor, maior expoente, o sinal dos dados e

a operação a ser realizada pela arquitetura (soma ou subtração). Além disso, este bloco realiza um alinhamento das mantissas, indispensável para realizar a operação. Na figura 4.9 pode ser observada uma descrição da arquitetura proposta para o bloco CAM envolvendo várias LUTs, a fim de evitar cálculos básicos de comparação e consumo de recursos lógicos importantes. Também são mostrados dois multiplexadores, os quais têm circuitos com lógicas especificas para lidar com somas binárias, deslocamentos binários para a direita ou esquerda e de concatenação (sempre que a lógica de seleção ou de controle precise). Neste caso, as saídas deste bloco são todas registradas para o seu posterior uso no bloco NM.

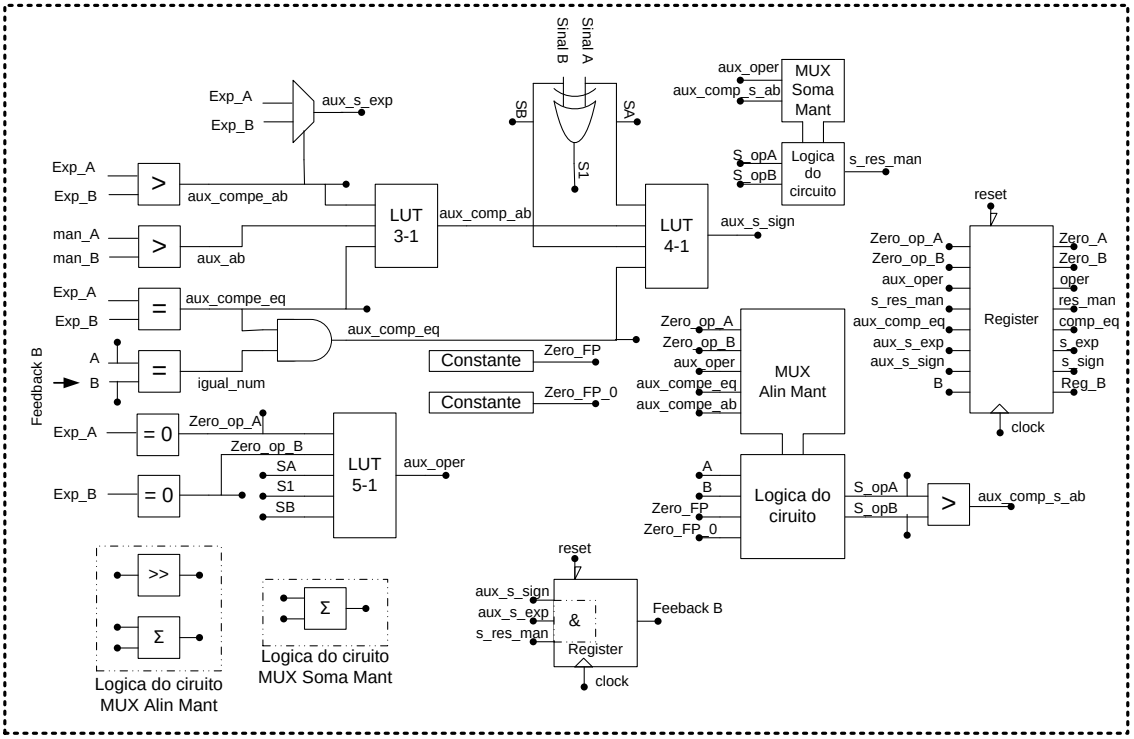


Figura 4.9. Arquitetura proposta para o bloco comparar e alinhar mantissa (CAM)

O bloco NM tem como tarefa devolver ao formato o valor da soma efetuada no bloco CAM, o que é chamado de normalização. Para fazer isso, um processo mostrado na figura 4.10 usa um multiplexador e um registrador. Neste processo são introduzidos vários sinais usados como condições para executar alguma operação, que no caso são deslocamentos binários e algu-

mas concatenações. No final, a saída é registrada e assim o valor obtido junto com a sua normalização é mantido em até pelo menos um ciclo de *clock*, ou até o sinal do *reset* ser ativado, apagando os valores acumulados assim como o valor final da saída.

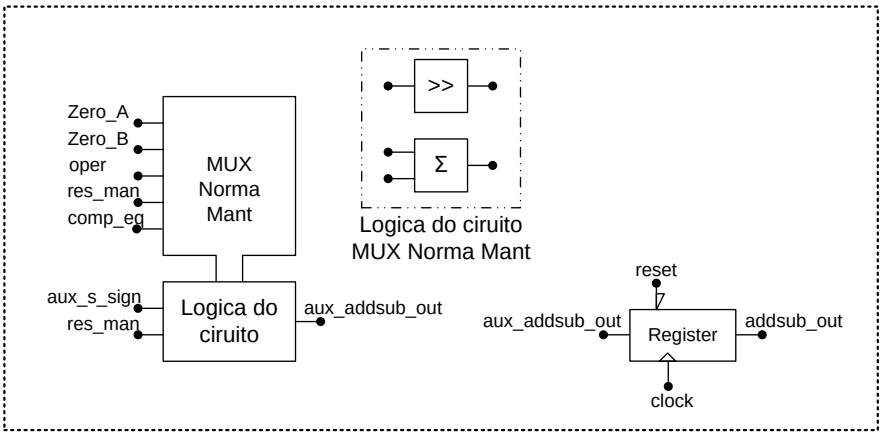


Figura 4.10. Arquitetura proposta para o bloco normalizar mantissa (NM)

4.3.2 Arquitetura de Multiplicação

As operações de multiplicação, assim com a de soma, fazem parte dos processos de redução envolvidos nas operações matriciais, sendo muito utilizadas e colocadas juntas, a fim de se obterem novas arquiteturas compartilhando suas características. Para realizar uma multiplicação em ponto flutuante uma sequência de passos deve ser seguida, a qual inicia com etapas de separação das diferentes partes do valor inserido, seguindo o formato IEEE−754.

Nessas separações são conferidas se as entradas dos valores são zero, infinito ou algum número com representação diferente e não válida do padrão IEEE−754. Caso os valores sejam válidos, um bit implícito '1' é inserido às mantissas. Seguidamente, as mantissas são multiplicadas, os expoentes são somados e é subtraído o valor de *bias* ao resultado, $(E_A + E_B - bias)$. Adicionalmente, o sinal é determinado na multiplicação $(S_A xor S_B)$ [112].

Por fim, uma normalização é feita caso seja necessário. Isto é feito se o bit mais significativo do resultado da mantissa for '1'; caso contrário, o resultado da mantissa é deslocada à esquerda até encontrar o valor de '1' no bit mais significativo. Além disso, para cada bit deslocado é subtraído '1' do valor do expoente, e finalmente, são concatenados os resultados do sinal, expoente e mantissa, obtendo assim o valor desejado [112].

A arquitetura do circuito de multiplicação (FP-Mul) é descrita na figura 4.11. Esta arquitetura está divida em duas etapas, sendo necessário um ciclo de *clock* para cada uma. A primeira etapa realiza todas as comparações (valor zero, infinito, etc), assim como calcula o valor do sinal do resultado, o resultado preliminar do expoente e a multiplicação das mantissas. Mediante uma operação lógica *xor* os sinais dos valores de entrada são comparados e o sinal do produto é determinando. Um bit extra é adicionado aos expoentes indicando *overflow*, o que leva necessariamente à subtração do resultado da soma dos expoentes com o *bias* devido ao duplo valor de bias obtido. Por outro lado, as mantissas de M_w bits junto com o bit implícito são multiplicadas resultando em uma palavra de $2(M_w + 1)$ bits que é truncada nos primeiros $M_w + 2$ bits [112].

Na etapa seguinte, é avaliado se no resultado da multiplicação das mantissas o bit MSB é '1'. Este bit permite controlar os M_w bits mais significativos do valor atual da mantissa, ou o valor da mantissa deslocado à esquerda em uma posição mediante um multiplexador. Caso o MSB do resultado da multiplicação das mantissas seja igual a '1', o resultado preliminar do expoente deve ser incrementado em '1'. Por fim, a existência de um *overflow* é avaliada no resultado do expoente, assim como é avaliado se os operandos de entrada são infinitos na representação IEEE−754. Caso afirmativo, o resultado final do produto é infinito, caso contrário concatenam-se os valores do sinal, expoente e mantissa [112].

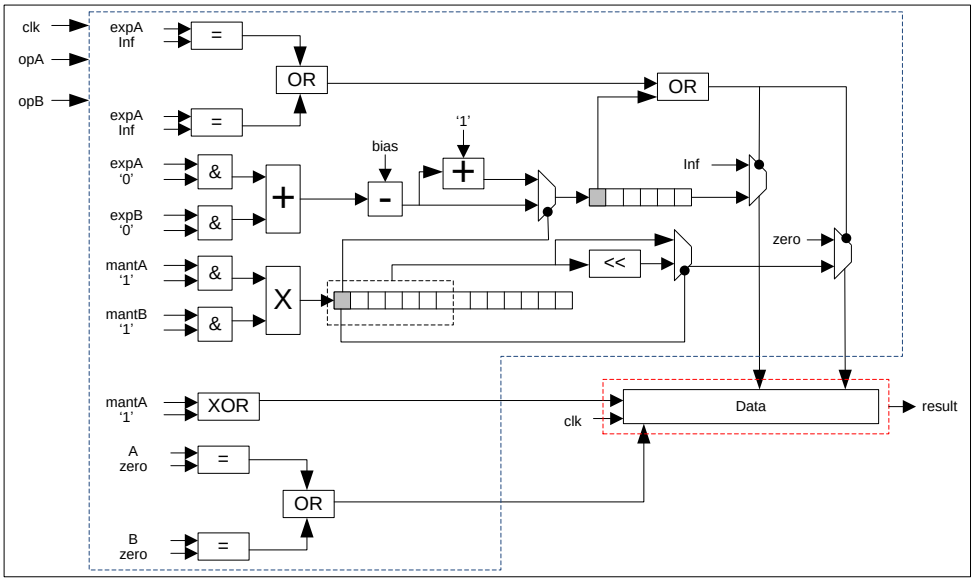


Figura 4.11. Arquitetura do Multiplicador em Ponto Flutuante (modificado de [112])

4.3.3 Resultados de Síntese e Análise da Precisão

Focando no desempenho dos operadores aritméticos e na redução do erro associado nas operações, os circuitos para *soma/subtração* e *multiplicação* em ponto flutuante foram desenvolvidos. Esses operadores foram configurados para lidar com palavras de 32 e 64 bits, baseados no formato IEEE−754, permitindo assim verificar a relação entre parâmetros de projeto tais como: custo em área lógica, tempo de execução e propagação do erro. Sendo este último parâmetro (a propagação do erro) um dos requisitos importantes no desenho de arquiteturas complexas, tal como as utilizadas no cálculo de operações matriciais.

Neste projeto foi usado como linguagem de descrição de *hardware* o VHDL mediante o uso da ferramenta desenvolvimento Vivado 2012.4 da Xilinx e configurado, para usar como dispositivo FPGA a família Kintex7. Assim, neste capítulo, são apresentados os resultados de síntese obtidos na implementação desses circuitos aritméticos na plataforma de desenvolvimento $KC705$, contendo um FPGA tipo $XC7K325T - 2FFG900$.

A Tabela 4.1 apresenta os valores do custo em área lógica relacionados às arquiteturas despois da síntese em termos flip-flops (FF), LookUp Tables (LUTs) e blocos dedicados de processamento digital (DSP48Es). O valor da frequência de operação do circuito também é incluída nesta tabela, expressada em Mega-Hertz (MHz). É possível observar, nesta tabela 4.1, como os operadores aritméticos quando configurados com uma representação de 64 bits consomem uma área lógica maior e, porém, um desempenho menor quando comparados com uma configuração com representação de 32 bits.

Tabela 4.1. Resultados de síntese para as arquiteturas FP-Add e FP-Mul em ponto flutuante

Modulo	Precisão (bits)	LUTs	DSP48(E)	Freq. [MHz]
FP-Add	32	559(0.27%)	0	364.904
	64	1310(0.64%)	0	338.415
FP-Mul	32	75(0.03%)	2(0.23%)	267.423
	64	459(0.22%)	9(1.07%)	233.875

Pode-se concluir que o circuito FP-Add possui um consumo alto de lógica combinacional se comparado com o consumo do circuito FP-Mul, onde este último requer menos recursos, porém faz uso de blocos DSPs embarcados. Embora, o circuito FP-Add não utiliza multiplicadores (recursos DSPs), o mesmo apresenta uma frequência de operação maior quando configurado tanto para uma representação de 32 bits quanto para uma representação de 64 bits .

Pode ser observado que os resultados de síntese destes operadores aritméticos mostram um consumo de recursos de 1% de LUTs e de menos do que 1% de DSPs disponíveis no dispositivo selecionado. Adicionalmente é importante ressaltar que o dispositivo FPGA utilizado ($XC7K325T-2FFG900$) é de capacidade media se comparado com os dispositivos de maior capacidade da família Kintex7.

Na tabela 4.2 são apresentados os resultados obtidos pela análise quantitativa da precisão das arquiteturas descritas da *soma/subtração* e *multiplicação*. É importante salientar que estas arquiteturas diferem das arquiteturas apresentadas [108, 112], no sentido de que a descrição *hardware* dos circuitos foi aprimorada visando uma economia de recursos, especificamente de blocos DSPs. As modificações citadas trouxeram, como efeito colateral, um aumento da frequência de operação. Finalmente, as arquiteturas descritas também foram modificadas para tratar dados em forma sequencial (na forma de um *pipeline*).

Tabela 4.2. Resultados da análise de precisão das arquiteturas FP-Add e FP-Mul em ponto flutuante

	Erro	32(8,23)	64(11,52)
FP-Add	MSE	7.63E-5	8.75E-8
	MAE	6.45E-3	9.27E-4
FP-Mul	MSE	2.12E-5	8.82E-11
	MAE	3.24E-2	2.16E-5

Como foi descrito na metodologia de verificação, os valores de avaliação introduzidos permitem obter os resultados do pior cenário de operação do circuito e, como esperado, o melhor comportamento é encontrado para um formato de 64 bits. O circuito FPmul apresenta o maior valor de MSE comparado à FP-Add. Isto é devido a erros incorporados no processo de truncamento, que facilmente podem ser resolvidos acrescentando etapas extras de arredondamento.

4.4 Arquitetura de FP-FMAC (*união multiplicação soma acumulação em ponto flutuante*)

Em 1990, a primeira instrução unida de FMA (*multiplicação soma*) foi proposta em [114], começando uma série de trabalhos visando melhorar

a primeira arquitetura FMA, a qual foi inicialmente implementada em ASICs, e em unidades integradas em *pipelines* de CPUs. Operações aritméticas em cadeias são comuns em diferentes operações matriciais, como as encontradas nas operações (ou circuitos) de redução, representando uma boa porcentagem do total dos desafios a resolver quando implementadas em *hardware*. Para desenvolver a arquitetura proposta, de *instrução unida de multiplicação soma acumulação* em ponto flutuante, foram analisadas algumas propostas interessantes implementadas em *hardware* , entre as quais podem-se citar : [115, 116, 117, 118, 119, 27], sendo assim, a principal proposta tomada do trabalho de bibliotecas de ponto flutuante apresentado em [108] e a implementação em FPGA da FMA apresentada em [120] para uma representação de 64 bits e do [113]. O uso das bibliotecas em ponto flutuante (ver [108, 112]) está justificado pela necessidade de se ter um controle sobre técnicas de arredondamento, assim como o controle total das diferentes etapas na cadeia de operações aritméticas. Essas técnicas serão aplicadas aos valores finais obtidos, reduzindo assim a propagação do erro associado em cada operação conjunta de multiplicação, soma e acumulação.

A fim de reduzir a propagação do erro de duplo arredondamento uma metodologia de fusão de *datapath* foi utilizada, tal como comentado em 4.2. Com esta metodologia foi possível melhorar tanto o desempenho, como reduzir a propagação do erro associado no processo de multiplicação, soma e acumulação. A figura 4.12, descreve o *datapath* da nova arquitetura desenvolvida, chamada de FP-FMAC (*união multiplicação soma acumulação em ponto flutuante*).

Os circuitos chamados de *circuitos de redução* são muito difíceis de implementar devido à alta complexidade das arquiteturas dos somadores em ponto flutuante. Essa complexidade é devida às etapas que envolvem o

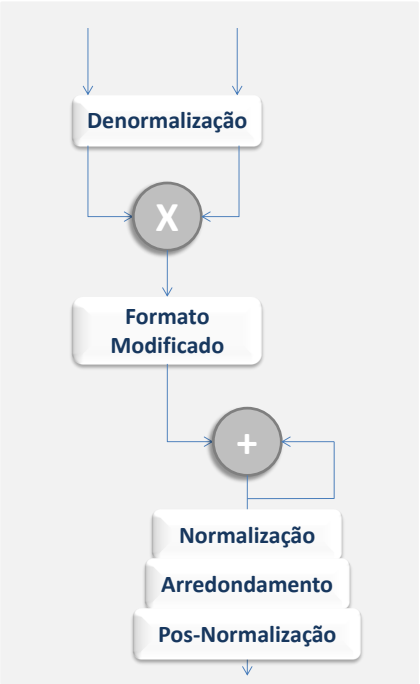


Figura 4.12. *Datapath* da arquitetura de FP-FMAC

processo de soma em ponto flutuante, sendo difícil realizar uma quebra que permita conseguir uma arquitetura em fluxo de dados. A FP-FMAC (*união multiplicação soma acumulação em ponto flutuante*) é uma arquitetura capaz de realizar a multiplicação em ponto flutuante ($a \times b$), seguida da soma em ponto flutuante (que é $(a \times b)_{resultado} + c_{acumulação}$) em um único bloco de *hardware*, realizando apenas uma operação de arredondamento depois de ambas as operações (multiplicação e soma) desenvolvidas [121, 32], tal como mostrado na figura 4.12.

A seguir são descritas as etapas da arquitetura FP-FMAC desenvolvidas neste trabalho:

1. Comparar e separar o sinal, expoente e mantissa dos operandos de entrada. Conferir se as entradas são zero, infinito ou uma representação inválida. Se o operando é válido adicionar o bit implícito '1' da mantissa.

2. Alinhar valores e multiplicar as mantissas, somar os exponentes e

determinar o sinal.

3. Separar o sinal, expoente e mantissa dos operandos de entrada. Conferir se as entradas são zero, infinito ou uma representação inválida.

4. Comparar os dois operandos. Deslocar à direita o menor dos números. O número de bits a deslocar correspondente ao resultado da subtração dos exponentes, sendo dita diferencia o resultado preliminar do expoente. Somar ou subtrair as mantissas segundo a operação desejada.

5. Deslocar à esquerda o resultado atual da mantissa até encontrar o valor de '1' no bit mais significativo (MSB). Para cada bit deslocado, subtrair '1' no valor atual do exponente. Finalmente, concatenar os resultados do sinal, exponente e mantissa.

Os passos 1 a 2 são executados em um único ciclo de *clock* assim como os passos 3 a 4. Por fim, o passo 5, de normalização, é desenvolvido tomando também um único ciclo de *clock*. Assim, depois de uma latência três ciclos de *clock* a arquitetura FP-FMAC gera um dado a cada ciclo de *clock*.

Na figura 4.13 é apresentado o diagrama de blocos da arquitetura do FP-FMAC. Três partes compõem a arquitetura: (a) um primeiro bloco que compara e alinha a mantissa para realizar a multiplicação, (b) um segundo bloco compara e alinha a mantissa para realizar a soma onde por sua vez é realizada a acumulação, e, por fim, (c) o bloco de normalização da mantissa (NM). Esses três blocos estão projetados para trabalhar como uma arquitetura baseada em fluxo de dados, permitindo assim a saída de um novo resultado a cada ciclo de *clock*, após uma latência de três ciclos necessários a fim de preencher os diferentes processos da arquitetura.

É importante ver que na saída do segundo bloco tem-se uma retroalimentação para a entrada desse bloco. Essa saída permite realizar a acumulação

dos dados introduzidos, empregando um sinal com um comprimento de palavra maior, chamado de HRCS (*high radix carry-save*). O valor somado é enviado de volta para o bloco sem passar por uma normalização, o que permite a realização desta operação em um único ciclo de *clock*. Apenas no final da operação, quando os valores totais introduzidos na operação são somados e acumulados, é quando realmente o valor final deve ser normalizado para o seu posterior uso.

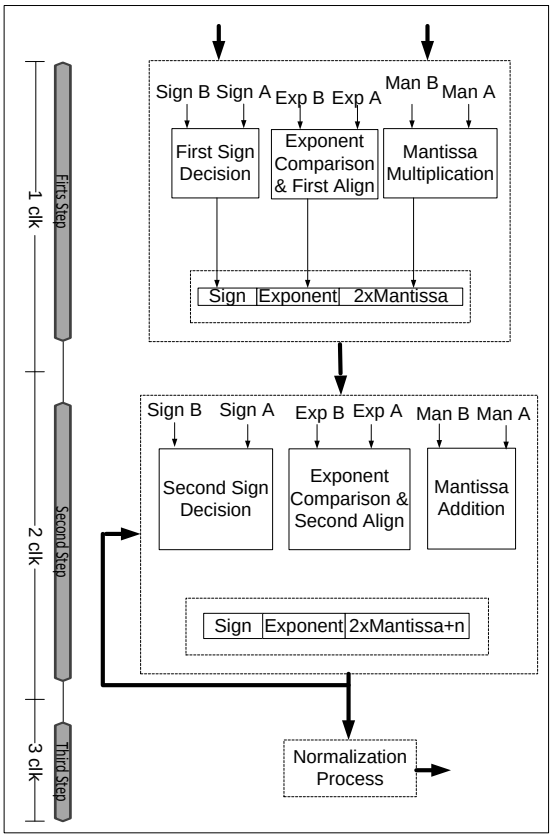


Figura 4.13. Diagrama de blocos da arquitetura FP-FMAC

O segundo bloco, permite comparar vários aspectos na entrada dos dados tais como o dado com maior valor, maior expoente, o sinal dos dados e a operação a ser realizada pela arquitetura (soma ou subtração). Além disso, este bloco realiza um alinhamento das mantissas, tarefa indispensável para realizar a operação. O bloco de normalização tem como tarefa devolver ao formato o valor da operação efetuada.

4.4.1 Resultados da implementação

A linguagem de descrição *hardware* usada foi o VHDL assistido por o *Xilinx Vivado* 2012.4, sendo que as arquiteturas foram implementadas na plataforma de desenvolvimento $KC705$, contendo um FPGA tipo $XC7K325T-2FFG900$. Como se pode observar os resultados da síntese do FP-FMAC, apresentados na tabela 4.3, mostram uma frequência acima de 100MHz. Este resultado obtido mostra, de forma clara, o ganho em desempenho alcançado quando a solução é implementada em uma arquitetura em fluxo de dados. Os valores de LUTs são considerados aceitáveis, dado que este FPGA tem recursos suficientes (203800 LUTS disponíveis).

Tabela 4.3. Resultados de síntese para a arquitetura de FP-FMAC

	Precisão (bits)	LUTs	DSP48(E)	Freq. [MHz]
FP-FMAC	32	305(0.14%)	2(0.23%)	204.145
	64	806(0.39%)	9(1.07%)	178.13

Os resultados do erro, sendo o parâmetro mais importante no desenvolvimento deste trabalho, pode ser analisado mediante a tabela 4.4. Estes resultados mostram como a propagação do erro aumenta enquanto o número de valores inseridos aumenta. Isto faz com que um bom controle da propagação do erro nas operações seja relevante, a fim de resolver sistemas que envolvem matrizes de grande porte (lembrar o fato de que as matrizes são considerados sistemas "mal condicionados").

Tabela 4.4. Resultados da análise de precisão da arquitetura de FP-FMAC

	Erro	32(8,23)	64(11,52)
FP-FMAC	MSE	5.68E-6	9.37E-9
	MAE	4.25E-3	8.16E-5

É importante ressaltar que o erro associado da arquitetura FP-FMAC, mostrado na tabela 4.4 é menor do que a soma do erro apresentado pelas

arquiteturas FP-Add e FP-Mul mostrado na tabela 4.2.

4.5 Arquitetura de decomposição QR baseada no método de ortonormalização modificado de Gram-Schmidt

Com o intuito de implementar arquiteturas para solução de sistemas matriciais, foram desenvolvidas diferentes arquiteturas para atingir o objetivo. Uma dessas arquiteturas é a decomposição e/ou fatoração QR, apresentada no capítulo 2.

Este método decompõe a matriz dada A em uma matriz ortogonal Q e uma matriz triangular superior R [122]. O algoritmo para a realização de tal decomposição é apresentado no algoritmo (2). Nesta parte do algoritmo é possível obter uma solução paralela devido às independências das operações. A solução aqui desenvolvida para a arquitetura QRDA (utilizando arranjos sistólicos completamente paralelizáveis e em *pipeline*) é baseada na proposta desenvolvida em [123] e modificada para tratar sistemas maiores que 4×4 assim como para usar uma representação em ponto flutuante. O algoritmo QR apresentado aqui está baseado no método Gram-Schmidt. O objetivo básico do método é produzir uma base orto-normal para um determinado espaço dado R^n (vide [124]).

Uma arquitetura dedicada a providenciar estes cálculos reduziria significativamente a latência do circuito, prestando atenção especial ao consumo de recursos, os quais seriam fortemente dimensionados em termos de multiplicadores e somadores. Para realizar essa arquitetura, os passos mostrados no algoritmo (2) foram seguidos, dividindo estes em dois processos principais. O primeiro, composto por duas unidades, chamadas de processo de normalização (NP) e o processo de ortogonalização (OP). O segundo, é a unidade de controle QR como mostrada na figura 4.14. Este controlador

permite o fluxo dos dados através dos processos NP e OP, ativando os diferentes multiplexadores associados a cada processo. A unidade QR é outra modificação apresentada neste trabalho referente a proposta desenvolvida em [123].

Algoritmo 2: Pseudo-código para a decomposição QR usando Gram-Schmidt Modificado

Input: A

1 Saídas Q, R;
2 $[m, n] = size(A)$;
3 $Q = A$;
4 $R = zeros(n)$;
5 **for** $j = 1...n$ **do**
6 **for** $i = 1...j - 1$ **do**
7 atribuir a $R(i, j)$ o valor de $Q(:, i)^T * Q(:, j)$;
8 atribuir a $Q(:, j)$ o valor de $Q(:, j) - Q(:, i) * R(i, j)$;
9 **end**
10 atribuir a $R(j, j)$ o valor da norma de $(Q(:, j))$;
11 atribuir a $Q(:, j)$ o valor da ortogonalização $Q(:, j)/R(j, j)$;
12 **end**

A unidade NP foi desenvolvida implementando os passos 11 e 12 do algoritmo (2), e a unidade OP por sua vez, foi desenvolvida implementando os passos 8 e 9. Pode-se observar que ambas as unidades NP e OP desenvolvem operações internas de redução matricial, o que leva necessariamente ao uso do bloco FP-FMAC previamente desenvolvido.

A tarefa de produzir o vetor de valores da matriz ortogonal Q, assim como os valores diagonais da matriz triangular R, está a cargo da unidade NP. Já a unidade OP é responsável por produzir os valores acima da diagonal da matriz R.

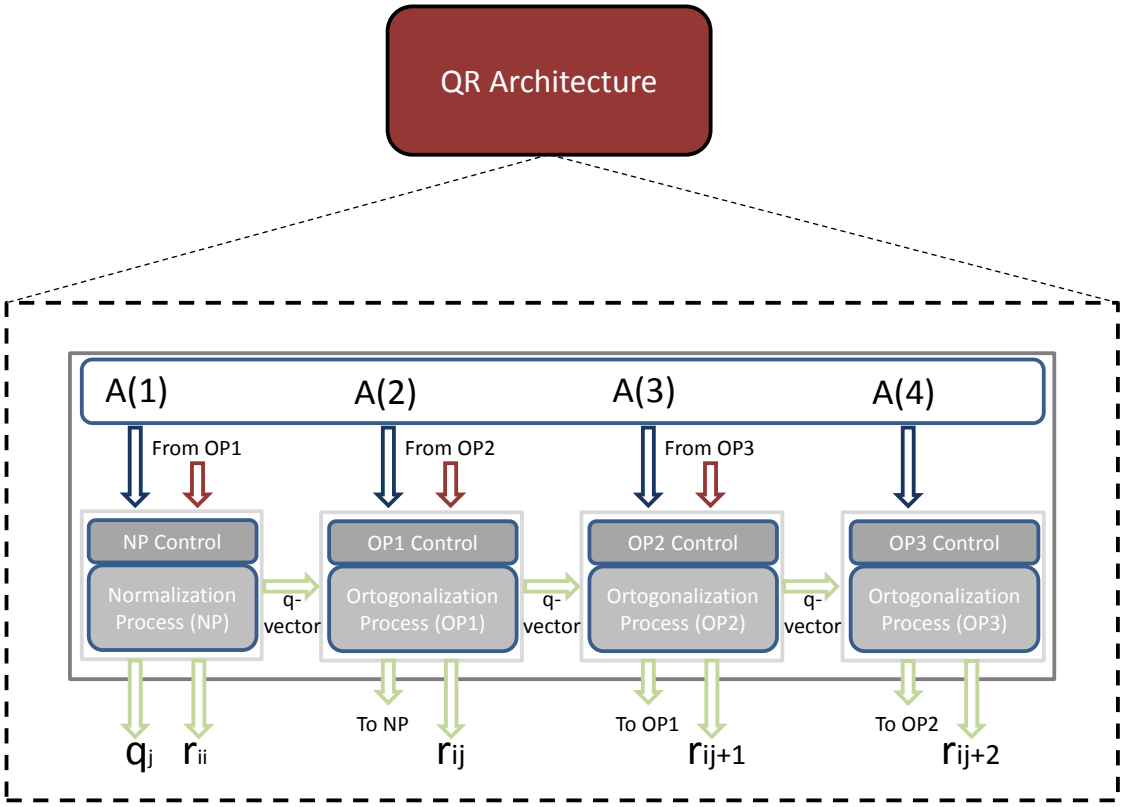


Figura 4.14. Estrutura em arranjos sistólicos da arquitetura de decomposição QR usando ortonormalização modificada de Gram-Schmidt

4.5.1 Estado da arte: Implementações em *hardware* reconfigurável do algoritmo de decomposição QR

Este trabalho apresenta uma comparação das diferentes implementações em *hardware* da decomposição QR (ver tabela 4.5).

Algumas propostas diferentes da decomposição QR podem ser encontradas tais como em [123], [125], [126], [127], [128], [129] e [130], mas devido a que são propostas e implementadas para lidar com uma representação em ponto fixo uma comparação entre elas foi evitada.

Observa-se que algumas implementações podem processar grandes tamanhos de matrizes, embora, isso dependa do dispositivo usado e se é ou não utilizada memória RAM externa. Também, solução de matrizes esparsas, as quais são capazes de tratar com matrizes de grandes tamanhos, não são

Tabela 4.5. Comparação das implementações da Decomposição QR em ponto flutuante

Referência	Ano	Tamanho da Matriz	Esparsa	Dispositivo	Requisitos da Matriz
[131]	2007	1280×640	Densa	Virtex II	linhas $\geq$ Colunas
[132]	2008	12×12	Densa	Virtex 5	Não-Singular
[133]	2009	$4k \times 4k$	Densa	Stratix II	Não-Singular
[111]	2012	400×400	Dense	Stratix V	linhas $\geq$ Colunas
EstaArquitetura	2014	100×100	Densa	Kintex 7	linhas $\geq$ Colunas

consideradas neste trabalho já que por definição carecem de generalidade.

Além disso, a referência [111] mostra que eles podem processar sistemas lineares com matrizes de tamanho 400×400 onde internamente lidam com o processo de decomposição QR alcançando um desempenho de até 162 GFLOPS. Mas as mesmas não apresentam informações referentes ao desempenho da arquitetura de decomposição QR assim como também não é apresentada informação referente ao desenho. Adicionalmente, neste trabalho, o dispositivo usado Kintex7 implementa multiplicadores DSP de 25×18 em vez dos multiplicadores DSP de 27×27 da Stratrix V usada em [111].

Também, é importante salientar que a proposta aqui desenvolvida, aumenta o seu desempenho enquanto o sistema incrementa seu tamanho, sendo limitado apenas pelo dispositivo *hardware* utilizado. Por último, o uso de multiplicadores DSP de tamanhos superiores aos usados pela kintex (25×18), permitiria reduzir as latências geradas pelas operações de multiplicação dentro da arquitetura.

4.5.2 Resultados da Implementação

A linguagem de descrição *hardware* usada foi o VHDL assistido por o *Xilinx Vivado* 2012.4 e as arquiteturas foram implementadas em uma plataforma

de desenvolvimento $KC705$, contendo um FPGA tipo $XC7K325T - 2FFG900$. Os resultados da síntese dos diferentes módulos: NP e OP, assim como os resultados do módulo de normalização vetorial (VN) são apresentados nas tabelas 4.6, 4.7 e 4.8, respectivamente.

Tabela 4.6. Resultados de síntese para o modulo NP

Modulo	LUTs	DSP48(E)	Freq. [MHz]
NP	1535(0.75%)	2(0.23%)	193.45

As implementações realizadas usam, além das arquiteturas aritméticas próprias previamente descritas, as bibliotecas de ponto flutuante disponíveis na Xilinx de divisão e raiz quadrada, as quais podem ser configuradas para trabalhar com diferentes tamanhos de palavra, tendo a precisão definida pelo usuário mediante o gerador de código fornecido pela Xilinx. Neste caso, as bibliotecas foram configuradas para manter uma precisão simples: 32 *bits*.

Devido ao fato de ser uma implementação usando uma arquitetura de fluxo (*pipeline*), as bibliotecas da Xilinx assim como todo o desenvolvimento dos diferentes módulos foram pensados para manter o fluxo dos dados enquanto novos valores são inseridos constantemente, produzindo ao final um valor a cada ciclo de *clock* logo após do tempo de latência ser alcançado.

Tabela 4.7. Resultados de síntese para o modulo OP

	LUTs	DSP48(E)	Freq. [MHz]
Modulo OP	1052(0.51%)	2(0.23%)	201.6

Com essa exigência, as bibliotecas de ponto flutuante foram configuradas permitindo projetar os diferentes módulos: NP, OP, VN. Porém, nesta proposta nem todas as operações têm um bom desempenho. Uma dessas arquiteturas é o acumulador somador multiplicador em ponto flutuante, razão pela qual foi necessária sua construção, permitindo assim alcançar

resultados satisfatórios de desempenho, tal como mostrado na tabela 4.8 (referente à implementação da norma vetorial).

Tabela 4.8. Resultados de síntese para o módulo norma vetorial (VN)

	LUTs	DSP48(E)	Freq. [MHz]
Modulo VN	694(0.34%)	2(0.23%)	198.7

A tabela 4.6 mostra uma frequência de operação no módulo NP de aproximandamente $193MHz$, o NP contem o modulo VN que tem a tarefa de realizar a norma vetorial. Esse processo é computacionalmente custoso e implica o uso da unidade de FP-FMAC. A arquitetura QRD-MGS pode operar aproximadamente a uma frequência de 170MHz com um desempenho continuo de 64.6 GFLOPS. Os resultados finais de toda a arquitetura, configurada para diferentes tamanhos de matrizes, são apresentados na tabela 4.9. Note-se que o consumo de recursos utilizado pela arquitetura indica que o dispositivo pode suportar tamanhos de matrizes maiores.

Tabela 4.9. Consumo de recursos *hardware* depois da síntese da arquitetura QRD-MGS para diferentes tamanhos de matrizes

Tamanho da Matriz Modulo		4	10	30	50	70	100
QRD-MGS	LUTs	1659(0.81%)	7237(3.55%)	26261(12.88%)	35241(17.29%)	53179(26.09%)	87852(43.10%)
	DSP	8(0.95%)	20(2.38%)	60(7.14%)	100(11.90%)	140(16.66%)	200(23.8%)

Como parte da validação da arquitetura QRD-MGS uma análise de propagação do erro é apresentada na tabela 4.10. Nessa tabela pode ser observado os valores obtidos a aplicar a norma de Frobenius aos resultados produzidos na comparação entre os valores entregues pela arquitetura e os valores entregues pelo MatLab. Vale a pena salientar que ainda os valores apresentados de erro, no caso de um sistema de 100×100, sejam de 14.28 e 4.53 para as matrizes Q e R, respectivamente, pode-se afirmar que o erro alcançado pela arquitetura é satisfatório devido a que os cálculos são baseados nas amostras com matrizes de Hilbert. Isto demostra que a

arquitetura desenvolvida pode tratar sistemas com valores aleatórios mantendo os valores de propagação do erro associados aos cálculos toleráveis, sem precisar usar uma precisão maior.

Esta análise também mostra como é difícil de controlar a propagação do erro enquanto o tamanho do sistema matricial aumenta. Um aumento no tamanho da palavra poderia ajudar significativamente a reduzir este erro associado, mas isto leva a latências maiores (devido a número de DSPs aninhados), assim como a um aumento considerável dos recursos lógicos necessários para implementar a arquitetura.

Tabela 4.10. Propagação do erro do circuito RTL sintetizado comparado com a referência em MatLab em representação em ponto flutuante dupla

	Matrix Size	Norma de Frobenius (Matriz Q/Matriz R)
QRD-MGS	4x4	3.46/3.01
	10x10	3.90/3.56
	30x30	8.12/4.10
	50x50	10.14/4.30
	70x70	11.93/4.42
	100x100	14.28/4.53

4.6 Arquitetura de Multiplicação Matriz Matriz e Arquitetura Multiplicação Diagonal Transposta Matriz

Como mostrado na figura 4.15, a arquitetura MMM (*multiplicação matriz matriz*) usa o circuito/arquitetura de redução desenvolvida, e comentada acima, chamada de FP-FMAC, como principal bloco dentro de sua arquitetura. A MMM é uma arquitetura totalmente *pipeline* e para sincronizar os dados são usados vários multiplexadores e uma série de *buffers*.

A arquitetura MDTM (*multiplicação diagonal transposta matriz*) toma a

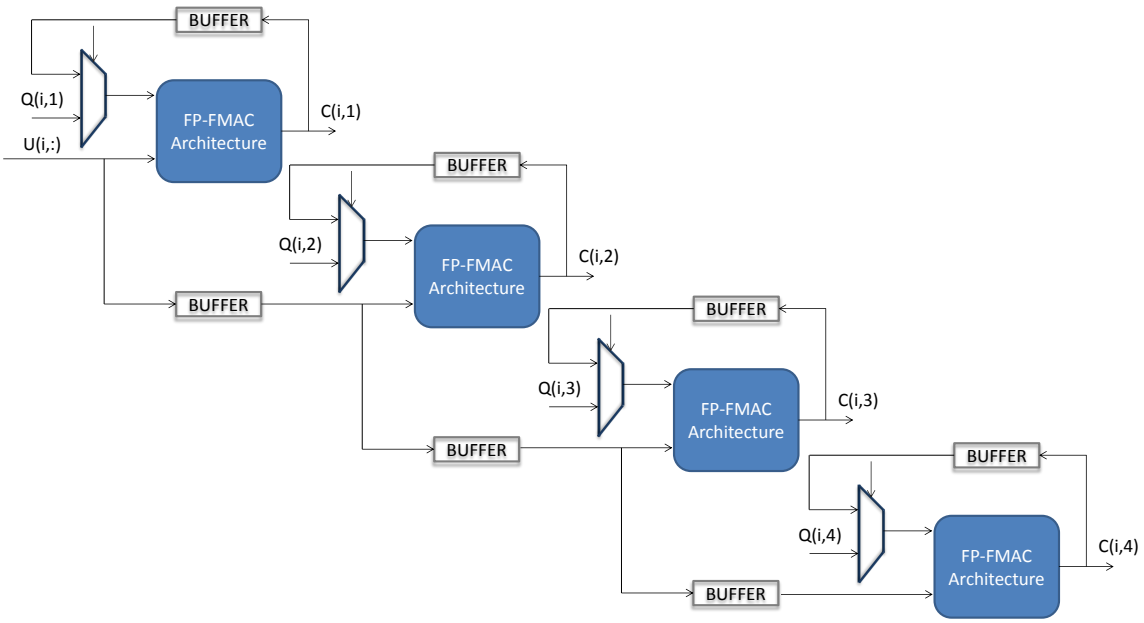


Figura 4.15. Estrutura de arranjo sistólico da arquitetura MMM

forma de Schur e encontra o resultado dado pela equação 4.2. Como descrito na figura (4.16), os valores dos elementos da matriz U e sua transposta U^T combinados com os valores diagonais da matriz S, são computados para obter o resultado desejado. Pode-se observar que novamente é utilizada a arquitetura FP-FMAC presente na arquitetura MMM para ajudar com os processos de redução.

$$Msqr = U \times S \times U^T \tag{4.2}$$

O principal gargalo da arquitetura MDTM é o cálculo da matriz transposta, causando um *pipeline hazard*. A transposta da matriz é uma arquitetura que recebe a matriz U e armazena os dados da mesma forma que os recebe sem utilizar mais recursos de memória (i.e., realizando a transposta da matriz no lugar). Assim, o circuito leva os valores correspondentes da transposta da matriz para a saída, decompondo assim as posições absolutas da transposta de uma matriz $n \times n$ em um registro predefinido, que é usado para tratar os valores correspondentes da transposição.

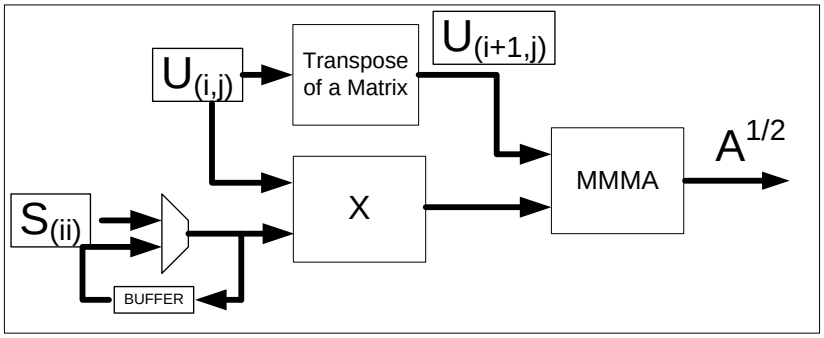


Figura 4.16. Desenho da estrutura da arquitetura MDTM

Uma memória interna para armazenar os valores da matriz U é utilizada na arquitetura MDTM devido à necessidade de se ter a transposta da matriz fora de estruturas *for loop*. O uso desta memória não interfere no fluxo de dados dentro da arquitetura.

4.6.1 Resultados da Implementação

O consumo de recursos da arquitetura MDTM é mostrado na Tabela 4.11, onde o número de DSPs utilizados na implementação da arquitetura está a apenas dois DSPs por cima do consumo de DSPs tanto do QRD-MGS quanto da arquitetura MMM. Este fato é devido à utilização do operador multiplicação de ponto flutuante, de modo a obter o resultado de $U \times S$. É importanta salientar que BRAMs são apenas utilizados na arquitetura MDTM, a qual foi configurada para receber os dados em cada ciclo de *clock*, para armazenar e realizar a transposta da matriz dada.

Tabela 4.11. Consumo de recursos *hardware* depois da síntese das arquiteturas MMM e MDTM para diferentes tamanhos de matrizes

Tamanho da Matriz / Modulos		4	10	30	50	70	100
MMM	LUTs	1439 (0.7%)	3289(1.61%)	9352(4.58%)	15476(7.59%)	22039(10.81%)	31953(15.67%)
	DSP	8	20	60	100	140	200
MDTM	LUTs	1829 (0.89%)	3574(1.75%)	9685(4.75%)	15947(7.82%)	22562(11.07%)	32821(16.1%)
	DSP	10	22	62	102	142	202

Vale a pena observar que o consumo de recursos da arquitetura MMM (ver

tabelas 4.11 e 4.9) é igual à apresentada pela arquitetura QRD-MGS em termos de DSPs, para as matrizes do mesmo tamanho.

Na tabela 4.12 são apresentados os resultados da validação da precisão mostrada pelas duas arquiteturas: (1) MMM e (2) MDTM. Nesta tabela pode-se observar como o aumento do sistema envolve um maior número de propagação do erro tal como mostrado pela norma de Frobenius. Os resultados apresentados na tabela 4.12 mostram como o sistema lida com dados difíceis de se representar em máquina, tal como os utilizados na matriz de Hilbert, comprovando que o erro alcançado pela arquitetura é satisfatório.

Tabela 4.12. Propagação do erro dos circuitos RTL sintetizados comparado com a referência em MatLab, em representação em ponto flutuante dupla

	Matrix Size	Norma de Frobenius
MMM	4x4	0.21
	10x10	1.87
	30x30	2.45
	50x50	3.71
	70x70	5.96
	100x100	7.37
MDTM	4x4	0.19
	10x10	3.62
	30x30	4.51
	50x50	5.84
	70x70	6.95
	100x100	8.48

4.7 Arquitetura da Raiz Quadrada de Uma Matriz

O algoritmo (3) mostra o método de Schur para obter uma solução numérica da função da raiz quadrada de uma matriz onde por transformações

de similaridade uma redução sistemática de matrizes em formas simples é alcançada (ver o a seção 2.3.2). Observe que o algoritmo usa a decomposição QR assim como algumas operações matriciais tais como multiplicação matriz-matriz e a transposta da matriz (usualmente denotada pelo sobrescrito T).

Algoritmo 3: Pseudo-código da raiz quadrada de uma matrix

Input: A

chamada da decomposição QR para decompor A e assinar o resultado a $[Q, R]$;

$U = Q$;

$B = R * Q$;

for $i = 1...Max\_iter$ **do**

    chamada da decomposição QR para decompor B e assinar o resultado a $[Q, R]$;

    $B = R * Q$;

    $U = U * Q$;

end

$S = sqrt(R)$;

$Msqrt = U * S * U^T$;

Para se obter uma simplicidade e eficiência maior na implementação, partes do algoritmo foram mapeadas em diferentes circuitos especializados para lidar com cada operação matricial particular (como comentado previamente). Adicionalmente as estratégias adotadas aqui permitem a reutilização do projeto; minimizando assim a fiação e o uso de multiplexadores entre os componentes de ponto flutuante.

A figura (4.17) mostra o fluxo de dados de toda a arquitetura, onde os principais componentes em ponto flutuante são apresentados. No fluxo de dados, o bloco com a letra maiúscula I corresponde a uma matriz identi-

dade, e os elementos diagonais da matriz R passam através de uma operação aritmética em ponto flutuante do tipo raiz quadrada. Esta arquitetura foi tomada das bibliotecas fornecidas pela *Xilinx IPCores*, e configurada para se encaixar apropriadamente aos requisitos deste trabalho.

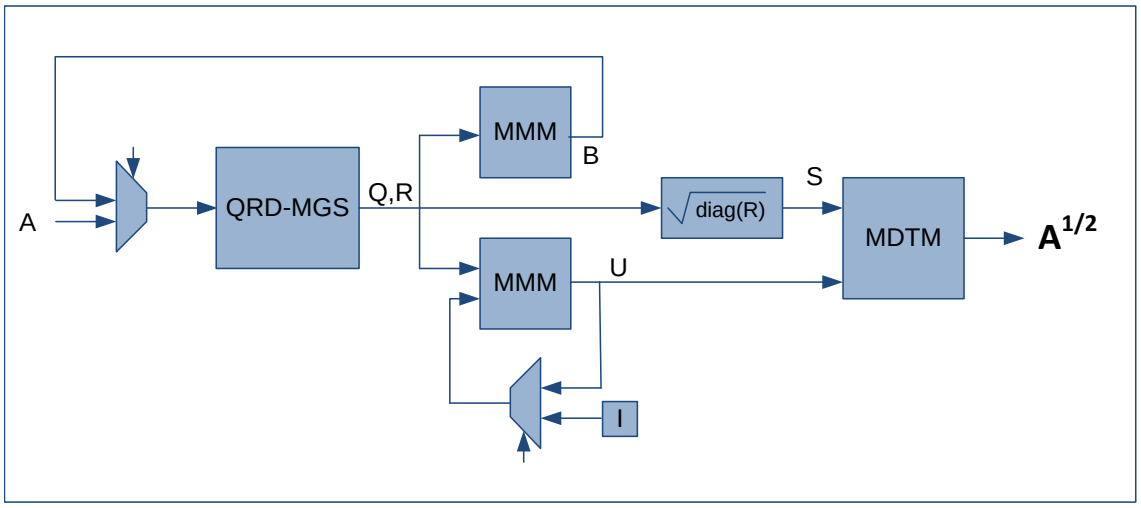


Figura 4.17. Fluxo dos dados na arquitetura.

4.7.1 Resultados da Implementação

A arquitetura da função de raiz quadrada de uma matriz proposta pode ser implementada para lidar com matrizes de tamanho até 100×100. Isto, devido à quantidade de recursos *hardware* , especialmente DSPs, disponíveis no dispositivo FPGA selecionado. A utilização de recursos do FPGA do projeto sob avaliação é coerente com as expectativas e é mostrado na tabela 4.13. Nessa tabela pode ser observado que para uma matriz de tamanho 100×100 o consumo de recursos de *hardware* LUTs é de 198794, correspondendo a aproximadamente o 97.54% do total de recursos disponíveis na FPGA selecionada. Adicionalmente, o número de DSPs consumidos é de aproximadamente 71.66% do total disponível.

Para comparar a precisão da arquitetura MatrixSqRt foram analisados quantitativamente dois exemplos numéricos: (1) $A - SS$ e (2) $A^{1/2} - S$.

Tabela 4.13. Consumo de recursos *hardware* depois da síntese da arquitetura MatrixSqRt para diferentes tamanhos de matrizes

Tamanho da Matriz		4	10	30	50	70	100
Modulo							
MatrixSqRt	LUTs	6827	17956	55163	84023	121603	198794
	DSP	26	62	182	302	422	602
	Freq (MHz)	120					

O desempenho desta análise para a arquitetura implementada executando sobre a plataforma de desenvolvimento selecionada usando uma representação em ponto flutuante de precisão simples é mostrada na Tabela 4.14. O erro é calculado comparando a saída da arquitetura MatrixSqRt simulada contra a referência previamente estabelecida em MatLab da matriz de Hilbert com representação em ponto flutuante de precisão dupla. As matrizes de Hilbert, onde as suas entradas são especificados como números de precisão de máquina, são difíceis de inverter usando técnicas numéricas e por causa disso são excelentes para avaliar a precisão da arquitetura implementada em *hardware* [134]. Para apresentar os dados em forma normalizada, a norma de Frobenius (como dado pela equação 4.1) é usada para obter uma medida da magnitude de erro geral na matriz resultante. Os valores gerados pela implementação de *hardware* chamada matriz S, são levados para o MatLab para calcular os dois exemplos numéricos.

É importante salientar que embora os valores apresentados de erro sejam de até 23 e 7 (ver tabela 4.14), para os diferentes casos numéricos sobre análise, é valido afirmar que o erro alcançado pela arquitetura é satisfatório devido a que os cálculos são baseados nas amostras com matrizes de Hilbert. Portanto, é mostrado que a arquitetura desenvolvida pode ser utilizada com uma precisão simples, a fim de tratar sistemas com valores aleatórios devido a que mantem os valores de propagação do erro associados toleráveis durante os cálculos.

Tabela 4.14. Propagação do erro dos circuitos RTL sintetizados comparado com a referência em MatLab em representação em ponto flutuante dupla

	Matrix Size	Norma de Frobenius (Caso 1/Caso 2)
MatrixSqRt	4x4	0.0213/ 0.0172
	10x10	15.3891/2.8281
	30x30	3.8523/ 1.7869
	50x50	5.6879/ 4.6738
	70x70	9.5213/ 4.4556
	100x100	23.0494/ 7.5706

4.7.2 Algumas Considerações Adicionais da Arquitetura

Conforme apresentado na secção 4.5, de modo a maximizar a eficiência da arquitetura, os componentes de *pipeline* devem permanecer completamente preenchidos quanto possível. Isto é conseguido nesta abordagem mediante multiplexação dos valores relacionados aos diferentes blocos do sistema.

Usando a arquitetura FP-FMAC descrita na seção 4.4, a profundidade do *pipeline* é dada pelo número de ciclos mostrado na figura 4.13 com uma latência de três. Adicionalmente, o circuito de MDTM é a parte crítica do projeto total, dado que internamente ele precisa realizar a transposta de uma matriz. Isto reduz o desempenho da arquitetura uma vez que o número de FLOPS diminui devido à latência no processo de armazenamento, de todos os elementos da matriz de dados. A arquitetura de função raiz quadrada da matriz é controlada por uma FSM (Finite State Machine), que tem a tarefa de endereçar (em uma ordem estrita), o fluxo de dados para os diferentes componentes de ponto flutuante.

4.8 Considerações Finais do Capítulo

As vantagens dos algoritmos propostos na álgebra linear numérica tendem a se limitar devido a problemas na propagação do erro induzidos pelas operações e não pelo método em se. Esses erros, conhecidos como erros de truncamento e em especial o erro de arredondamento duplo, devem ser tratados de tal forma que permitam resultados ótimos e com maior precisão.

Este trabalho demonstrou como o Método de Schur para lidar com a raiz quadrada de uma matriz, pode ser adequadamente implementado em *hardware* reconfigurável. A descrição detalhada dos circuitos desenhados foi apresentada mostrando as estruturas em arranjos sistólicos paralelos e de *pipeline* de cada um dos componentes em ponto flutuante. Os circuitos principais: QRD-MGS, MMM e MDTM, foram descritos, lidando eficientemente com a divisão interna dos processos da arquitetura proposta MatrixSqRt.

O circuito QRD-MGS pode processar até 64.6 GPFLOPs assim como pode ser configurado para lidar com diferentes tamanhos de matrizes, usando também, o FPGA da família Kintex7 deste projeto executado a 170MHz.

Usando como base o conhecimento adquirido no desenvolvimento das arquiteturas baseadas no método de Gauss-Jordan, foram estabelecidos parâmetros de precisão como requisitos do sistema. Contudo, uma metodologia de fusão de *datapath* das diferentes operações aritméticas foi implementada, permitindo reduzir a propagação do erro associado nos processos com multipla demanda de cadeias de operações em ponto flutuante. Desta metodologia surgiu a arquitetura FP-MAC. Uma arquitetura especial em ponto flutuante de multiplicação, soma e acumulação em um único bloco,

chamada de FP-FMAC, a qual foi descrita e lida com todos os processos de redução dentro dos circuitos principais. A arquitetura pode ser executada com um *clock* aproximado de 200 MHz.

Os resultados de precisão da arquitetura MatrixSqRt mostram que, ainda operando com matrizes de Hilbert, o sistema alcança erros não maiores que 23.0494 e 7.5706 para os diferentes casos numéricos de estudo quando implementado um sistema de 100×100 com representação em ponto flutuante de precisão simples. Finalmente, o circuito implementado exibe um desempenho de 10.41 GFLOPs e é capaz de computar soluções numéricas para a raiz quadrada de matrizes positivas definidas simétricas de diferentes tamanhos até 100×100 elementos no Kintex-7 FPGA KC 705 .

Para lidar com o processo de verificação e validação das arquiteturas propostas foi estabelecida uma metodologia de verificação funcional e de *hardware* que permitiu realizar diferentes análises. Entre os quais estão às análises quantitativas do erro associado das diferentes operações em ponto flutuante utilizadas e desenhadas neste trabalho. Esta metodologia facilitou padronizar o projeto de sistemas embarcados.

Capítulo 5 DISCUSSÃO DE RESULTADOS

Neste capítulo são discutidos os resultados das diferentes arquiteturas propostas para serem implementadas em *hardware* reconfigurável FPGA, baseadas no algoritmo de Eliminação de Gaussiana (e na variante do mesmo, a Eliminação Gauss-Jordan), tratando com a função de inversão de matriz assim como também a solução de um sistema linear de equações. Adicionalmente, são discutidos os resultados das arquiteturas realizadas para lidar com a raiz quadrada de uma matriz baseada no método de Schur.

5.1 Implementação das Arquiteturas para Solução de Sistemas de Equações Lineares e Inversão de Matrizes

Baseado no algoritmo de Eliminação Gaussiana, ou a sua variação Gauss-Jordan, foram introduzidas e apresentadas no capítulo 3 diferentes propostas em *hardware* reconfigurável FPGA para lidar com as soluções de sistemas de equações lineares e com a solução da inversão de matrizes. A arquitetura proposta permite a parametrização do sistema com relação ao tamanho da matriz e/ou sistema linear de equações.

Representações em ponto flutuante de 32, 40 e 64 bits foram utilizadas para implementar a arquitetura de inversão de matrizes. Por outro lado, para implementar a solução do sistema de equações lineares apenas foi feita uma implementação com representação em ponto flutuante de precisão simples (32 bits). O sistema descrito e implementado serve para tratar sistemas lineares com matrizes densas, as quais estão relacionadas a sistemas fortemente acoplados. Tanto a arquitetura de inversão quanto a arquitetura de

solução de sistema de equações lineares podem ser configuradas para lidar com sistemas com tamanhos a partir de 4×4 até 120×120. O circuito desenvolvido serve para entregar ambos os resultados simultaneamente (inversão e a solução de um sistema linear de equações) e a configuração é feita mediante um gerador de código VHDL desenvolvido em MatLab.

Um conjunto de circuitos com tarefas específicas, baseados em arquiteturas de controle, foram desenvolvidos para dividir a complexidade da implementação e permitir a execução do método de eliminação Gauss-Jordan. A complexidade para resolver o problema foi dividida em três fases principais: (a) Eliminação Gaussiana (pivô parcial e *forward elimination*), (b) diagonalização (*back substitution*) e (c) normalização, onde são executadas pelos quatro processadores internos: (1) Troca de linhas, (2) Encontrar o Pivô, (3) Eliminação da Matriz e (4) Circuito de Controle de Eliminação Gaussiana.

As comparações do desempenho *hardware/software* foram realizadas considerando tanto a arquitetura desenvolvida quanto a execução em *software* do algoritmo de eliminação Gauss-Jordan tendo em conta o aumento da dimensionalidade da matriz. A verificação da arquitetura para solução de sistemas de equações lineares foi realizada implementando uma arquitetura que suporta um sistema linear de equações de 6×6. Adicionalmente, um análise experimental do erro foi desenvolvido usando o MatLab como comparador estatístico com precisão dupla.

Os resultados de síntese mostrados nas diferentes tabelas descrevem uma tendência do crescimento dos recursos *hardware* enquanto o sistema aumenta de tamanho. A frequência máxima que a arquitetura proposta suporta depende do tamanho do sistema configurado, para um sistema de matricial de 4×4 é de 301.416MHz e para um de 36×36 é de 166.984MHz, executado em uma plataforma com um FPGA Virtex-$5XC5VLX110T$.

Os recursos utilizados (sejam LUT, DSP e RAM), mostram que o dispositivo sobre o qual foi realizada a implementação (neste caso Virtex5) pode suportar matrizes com tamanhos maiores, devido ao mínimo consumo de recursos usados pela arquitetura. Esse aumento do tamanho do sistema está condicionado ao número de blocos DSP, assim como ao número de blocos de memória interna disponíveis no dispositivo FPGA selecionado. O número total de memória RAM usado pela implementação em *hardware* da arquitetura depende da precisão. Assim para uma precisão simples, dupla e 40-bits, os respectivos valores dos blocos de memória RAM usados para armazenar uma matriz de 120×120 são: 60, 120 e 120, respectivamente.

A análise de validação do erro na arquitetura mostrou como este cresce enquanto o valor do sistema matricial tratado aumenta e assim, o controle da precisão é afetado. São poucos os trabalhos na literatura que mostraram preocupação com a propagação do erro, sendo este uma parâmetro de projeto crítico atribuído à condição das matrizes de "elementos matemáticos mal condicionados". Valores do erro associado para sistemas de tamanhos 120×120 estão por volta de $1E-3$ quando utilizada uma precisão dupla, sendo este o melhor dos resultados. Uma validação do tempo de execução da arquitetura foi feita, apresentando o tempo usado pela FPGA (em micro segundos (us)) para se obterem as soluções do sistema linear de equações (e a inversa da matriz), usando um relógio de 100Mhz. Pode-se observar que para um sistema de equações lineares de 90×90 a arquitetura apresentou um tempo de execução de aproximadamente 1.4ms, muito superior aos apresentados na literatura (ver [99, 101, 90])

Para acelerar a verificação funcional e por sua vez, evitar esforços de configuração de interfaces, etc., desenvolvendo uma simulação HIL (*Hardware-in-the-loop*), foi utilizado também o *Xilinx System Generator* anexo ao Matlab/Simulink. A arquitetura foi transformada em um bloco o que

permitiu ser usada como qualquer bloco em Matlab/Simulink com a ferramenta XSG em duas formas: como simulação em *software* ou como simulação HIL. Também foi demostrada a aplicação da arquitetura apoiando o co-processamento e a aceleração em *hardware* da implementação de uma rede neuronal tipo GMDH.

5.2 Implementações em *Hardware* das Arquiteturas Baseadas em Fluxo de Dados

No capítulo 4 foram apresentadas as arquiteturas em fluxo de dados em ponto flutuante propostas para lidar com a solução numérica em *hardware* reconfigurável da função raiz quadrada de uma matriz simétrica definida positiva. A descrição detalhada dos circuitos projetados foi apresentada mostrando as estruturas em arranjos sistólicos paralelos e de pipeline de cada um dos componentes em ponto flutuante.

Unidades aritméticas de soma/subtração, multiplicação foram implementadas na FPGA selecionada. As implementações foram realizadas para diferentes tamanhos de palavra, incluindo a representação de precisão simples (32 bits) e precisão dupla (64 bits). Uma metodologia de fusão de *datapath* das diferentes operações aritméticas foi implementada, permitindo reduzir a propagação do erro associado nos processos com múltipla demanda de cadeias de operações em ponto flutuante e assim construir a arquitetura FP-MAC. Uma arquitetura especial em ponto flutuante de multiplicação, soma e acumulação em um único bloco, chamada de FP-FMAC, a qual foi descrita e lida com todos os processos de redução dentro dos circuitos principais.

Também, no capítulo 4 foram implementadas em *hardware* o conjunto de circuitos que conformam a arquitetura principal MatrixSqRt de cálculo

da raiz quadrada de uma matriz, sendo estas QRD-MGS, MMM, MDTM. Adicionalmente, foram realizadas implementações com matrizes de diferentes tamanhos partindo de 4×4 até 100×100 e usando uma representação em ponto flutuante de precisão simples. Neste sentido, observa-se que algumas implementações, referentes a decomposição QR, podem processar grandes tamanhos de matrizes, embora, isso depende do dispositivo usado e se é ou não utilizada memória RAM externa (ver [131, 132, 133, 111]). O circuito QRD-MGS aumenta o seu desempenho enquanto o sistema incrementa seu tamanho, sendo limitado apenas pelo dispositivo *hardware* utilizado.

Uma metodologia de verificação/validação das arquiteturas propostas foi seguida facilitando a padronização dos mecanismos teste assim como agilizar esta etapa no processo de desenho de sistemas embarcados. Esta metodologia foi dividida segundo o seu foco de análise em verificação funcional, verificação em *hardware* ou teste no dispositivo e validação das arquiteturas, permitindo uma verificação funcional junto com uma simulação HIL sendo uma novidade na verificação de sistemas matriciais.

Os resultados de síntese lógica mostraram que os operadores aritméticos permitem uma implementação em *hardware* especificamente no dispositivo FPGA selecionado devido a que o seu consumo de recursos é baixo comparado com o *hardware* disponível no dispositivo. Estes resultados também mostraram que a unidade de multiplicação, FP-Mul, faz uso de blocos DSPs embarcados e requer menos recursos quando comparada com a unidade de soma/subtração, FP-Add. Embora, a unidade FP-Add utiliza mais recursos combinacionais é bom ressaltar que não faz uso de DSPs embarcados. Estes resultados demostraram uma redução significativa dos recursos *hardware* comparada com suas versões antecessoras apresentadas em [108, 112]

Referente aos resultados de síntese, do conjunto de circuitos principais que subdividem as tarefas no processo de cálculo da raiz quadrada da matriz, pode-se observar que o circuito QRD-MGS pode processar até 64.6 GPFLOPs. Adicionalmente, o mesmo podee ser configurardo para lidar com diferentes tamanhos de matrizes, usando também, o FPGA da família Kintex7 deste projeto executado a 170MHz. Os resultados, ainda, mostraram que os recursos disponíveis no FPGA são suficientes para realizar implementações da arquitetura com sistemas matriciais de maior tamanho.

Finalmente, o circuito implementado exibe um desempenho de 10.41 GFLOPs, sendo capaz de computar soluções numéricas para a raiz quadrada de matrizes positivas definidas simétricas de diferentes tamanhos até 100×100 elementos no Kintex-7 FPGA KC 705, limitados apenas pelos recursos *hardware* do dispositivo.

Os resultados de validação por sua vez mostraram que os operadores aritméticos de FP-Add e FP-Mul possuem um erro quadrático médio (MSE) de $7.63E-5$ e de $2.12E-5$, respectivamente, para uma representação de 32 bits. Enquanto que a arquitetura FP-FMAC implementando as operações aritméticas em um único bloco apresentou um erro associado de $5.68E-6$ para a mesma precisão. Contudo, no marco das aplicações em sistemas embarcados, representações com precisões menores são apreciadas pelo baixo consumo de recursos *hardware* . Assim, implementações com palavras de 64 bits (embora, apresentam uma maior precisão) são evitadas e substituídas por representações menores, desde que sejam atingidos os requerimentos de precisão desejados.

Os resultados de precisão da arquitetura MatrixSqRt mostram que, ainda operando com matrizes de Hilbert, o sistema atinge erros inferiores que 23.0494 e 7.5706 para os diferentes casos numéricos de estudo quando implementado um sistema de 100×100 com representação em ponto flutuante

de precisão simples e avaliado segundo a norma de Frobenius. Isto mostra que se faz necessário realizar adequações nas estruturas internas das operações que envolvem cálculos com representações em ponto flutuante e assim se manter baixos os erros associados nas diferentes aplicações. Nenhum outro trabalho na literatura disponível reporta este tipo de análise estatístico e ainda menos para uma arquitetura de solução da raiz quadrada de uma matriz simétrica.

Os resultados também mostram que é possível reduzir a propagação do erro associado nas operações matriciais sem aumentar internamente a precisão usada nas operações em ponto flutuante. Embora, quando maior seja o sistema o erro de propagação aumenta, o que torna difícil seu controle apenas com representações simples devido às propagações do erro de duplo arredondamento.

5.3 Contribuições Gerais do Trabalho

A maioria das soluções propostas, lidando com a complexidade computacional da solução numérica de sistemas matriciais na área da álgebra linear numérica, encontram-se apenas em *software*. Assim, há uma carência de processadores com *hardware* dedicado capaz de aproveitar eficientemente o *software*, e as mesmas acabem rodando em processadores de propósito geral (GPPs). Se bem, o desenvolvimento de processadores especializados com arquiteturas paralelas aproveitando técnicas de *acessos a memória*, *pipeline* e *arranjos sistólicos*, representam um desafio na implementação em *hardware*, as mesmas técnicas implementadas em circuitos dedicados podem permitir encontrar o balanço entre os principais parâmetros de desenho: *precisão*, *desempenho*, *consumo de recursos lógicos* e *potência*.

Essa busca pelos processadores especializados implementados em *hardware*

reconfigurável, como aquele suportado pelas *FPGAs*, permitiu fornecer, de forma inédita, uma série de contribuições ao estado da arte na área de soluções em *hardware* para problemas de álgebra linear numérica para sistemas embarcados e computação de alto desempenho. Entre as contribuições deste trabalho podem-se citar as seguintes:

- O desenvolvimento em *hardware* reconfigurável de uma arquitetura que permite obter a solução numérica tanto da função de inversão de matrizes (densas neste caso), quanto à solução de um sistema de equações lineares. Essas arquiteturas foram baseadas no método de eliminação Gaussiana ou Gauss-Jordan, dependendo do caso, sendo que as mesmas permitem modos de configurações (descrição de código parametrizável), para tratar sistemas de diferentes tamanhos assim como com diferentes precisões baseados em uma representação em ponto flutuante padrão IEEE−754.

  A portabilidade de estas arquiteturas está, principalmente, condicionada ao número de recursos DSPs e de memória interna RAM disponíveis no dispositivo selecionado, sem desconsiderar outros aspectos referentes aos recursos lógicos. Adicionalmente, foram estudadas e propostas modificações em *hardware* referentes às posições dos passos internos do algoritmo visando diminuir o erro associado para pequenos argumentos de entrada.

- A aplicação do bloco de solução de sistema de equações lineares apoiando os cálculos de uma rede neural tipo GMDH e seu uso para predizer de forma aproximada a estrutura tridimensional das proteínas. O bloco SSL foi usado especificamente para acelerar o método dos mínimos quadrados.

- Uma simulação da arquitetura de solução de sistemas de equações lineares baseado no uso da ferramenta XSG (*Xilinx System Generator*).

Um processo de verificação funcional foi desenvolvido tomando vantagem do XSG, permitindo ambas as simulações *software* e *hardware-in-the-loop* (HIL) usando os resultados respectivos do modelo de referencia em MatLab.

- O projeto de *hardware* parametrizáveis para os operadores de soma subtração e multiplicação usando uma representação aritmética de ponto flutuante com 32 e 64 bits de precisão. Estas arquiteturas foram desenvolvidas como arquiteturas de fluxo de dados. Além disso, foi desenvolvida uma arquitetura capaz de realizar em um único bloco a operação conjunta de multiplicação soma acumulação em ponto flutuante. Os resultados mostraram reduções em uso de recursos DSPs e do erro associado aos cálculos.

- A proposta de uma metodologia de fusão de *datapath* para conseguir descrever uma operação de multiplicação soma e acumulação em um único bloco *hardware* , evitando erros de duplo arredondamento e reduzindo sobre custos de roteamentos e *shift register*, dadas pelas etapas de denormalização/normalização. Esta metodologia pode ser usada no desenvolvimento de processos com grandes cadeias aninhadas de operações aritméticas em ponto flutuante, o que significa uma redução significativa não do erro associado nos cálculos.

- A verificação/validação das arquiteturas de solução de sistemas matriciais, sendo que esta tarefa foi desenvolvida mediante a implementação de uma metodologia de verificação/validação que permitiu padronizar cada um dos experimentos e realizar uma comparação dos resultados de forma normalizada. Foram planteadas dois diferentes caminhos de verificação: (1) verificação funcional e (2) verificação em *hardware* , assim como também foi estabelecida a validação das arquiteturas referentes a parâmetros de desenho previamente estabelecidos. A metodologia tratou a precisão como principal parâmetro de desenho de

projeto e foi possível realizar uma análise do erro associado, o custo em área lógica, frequência de operação e tempo de execução.

- Como um dos resultados principais deste trabalho, a nova (segundo o conhecimento, e única) implementação em *hardware* reconfigurável FPGA do método de Schur para calcular a função raiz quadrada de uma matriz simétrica e definida positiva. Para lidar com todos os processos de cálculo internos foram usadas estruturas em forma de arranjos sistólicos baseadas em três blocos matriciais principais: QRD-MGS, MMM e MDTM. Esta arquitetura usa uma representação em ponto flutuante de 32 bits de precisão baseada no padrão IEEE-754. Devido à paralelização e pipeline das tarefas, esta arquitetura alcanço resultados de desempenho e propagação do erro satisfatórios no dispositivo embarcado.

- Como resultado ao desenvolvimento das arquiteturas foi desenvolvida uma ferramenta de geração automática de código VHDL no Matlab, como parte e apoio à metodologia de verificação/validação, facilitando desta forma a implementação em FPGAs das diferentes arquiteturas propostas e aceleração do tempo de desenvolvimento em *hardware* das operações matriciais assim como das operações aritméticas. O gerador de código VHDL permite algumas parametrizações, tais como o tamanho do sistema (ou da matriz) e a precisão numérica do formato em ponto flutuante utilizado.

- Finalmente, neste trabalho foi provada a viabilidade do uso de arquiteturas baseadas em fluxo de dados, com topologias em formato de arranjos sistólicos e *pipelines*, para acelerar o desempenho e, por sua vez, obter a solução numérica de sistemas lineares baseados nos métodos de eliminação Gaussiana, decomposição QR e de Schur.

Capítulo 6 CONCLUSÕES E PERSPECTIVAS DE TRABALHOS FUTUROS

Neste trabalho foi apresentado um estudo da implementação em *hardware* reconfigurável de operadores matriciais para solução numérica de sistemas lineares e/ou funções matriciais em aplicações de pequeno e grande porte.

Durante o trabalho foram abordadas técnicas baseadas na álgebra linear numérica, a fim de obter a solução numérica de sistemas lineares de equações e calcular duas funções matriciais: (1) Inversão e (2) Raiz Quadrada. As conclusões deste trabalho estão divididas por seções referentes a cada capítulo tratando as implementações em *hardware* reconfigurável realizadas. Por fim, são apresentadas as perspectivas de trabalhos futuros.

6.1 Conclusões

- Foram implementadas em FPGA soluções de sistemas de equações lineares, assim como a solução da inversão de matrizes, onde as arquiteturas propostas permitem a parametrização do sistema com relação ao tamanho da matriz e/ou sistema lineares de equações.
- Foram projetadas e implementadas diferentes arquiteturas em fluxo de dados (QRD-MGS, MMM, MDTM) que permitiram lidar com a solução numérica em *hardware* (MatrixSqRt) da função raiz quadrada de uma matriz simétrica e definida positiva.
- Mediante o uso da metodologia de fusão de *datapath* foi possível reduzir a propagação do erro associado aos cálculos, sem a necessidade

de aumentar internamente a precisão usada nas operações em ponto flutuante dentro das arquiteturas propostas.

- Uma metodologia de verificação/validação das arquiteturas desenvolvidas foi proposta, visando facilitar a padronização dos mecanismos de teste assim como agilizar esta etapa no processo de desenho de sistemas embarcados.

- Foi desenvolvida uma ferramenta automatizada de geração de código VHDL parametrizável para apoiar nos processos de verificação/validação, permitindo variações de parâmetros de escalabilidade, tal como o tamanho do sistema linear.

- O desenvolvimento e implementação da arquitetura MatrixSqRt mostrou a viabilidade de implementar em *hardware* reconfigurável FPGA o método de Schur, para lidar com a raiz quadrada de uma matriz simétrica e definida positiva.

6.2 Trabalhos Futuros

O paralelismo intrínseco que permitem os *FPGAs* é uma caraterística que torna especiais o uso em diferentes campos onde há uma necessidade pela computação de alto desempenho. Essa característica é muito atraente na computação paralela onde são abordados sistemas com alta demanda de tarefas computacionais, como encontrar a solução numérica de sistemas matriciais. Embora, o desenvolvimento desde trabalho permitiu introduzir novas soluções numéricas de sistemas matriciais embarcadas, existe uma grande lacuna por ferramentas computacionais de baixo custo, alto desempenho e precisão, que permitam lidar e sobre tudo acelerar a solução numérica de sistemas matriciais baseados na paralelização dos algoritmos.

Assim, as perspectivas sobre os diversos estudos e implementações possíveis, que podem ser realizados visando complementar e melhorar os resultados (apresentados neste trabalho) são listados a continuação:

1. Neste trabalho foram usadas tanto arquiteturas próprias assim como algumas de propriedade da Xilinx. O projeto e desenvolvimento de todas as demais arquiteturas próprias faltantes (tais como divisão e raiz quadrada) permitirá a completa portabilidade das arquiteturas às diferentes plataformas de desenvolvimento, sejam elas da Altera, Microsemi, etc., assim como, será possível a modificação e manipulação interna da sua estrutura.

2. Para reduzir os custos dos processos de denormalização/normalização e os problemas associados a estas etapas nas operações aritméticas, sejam em precisão assim como em área, é necessário uma implementação da metodologia de fusão de *datapath* das grandes cadeias aninhadas de operações aritméticas em ponto flutuante dentro da arquitetura. Contudo, uma análise, em profundidade, das diferentes cadeias de operações aritméticas em ponto flutuante presentes dentro dos circuitos, permitiria um grande avanço na redução dos problemas associados aos erros de duplo arredondamento.

   Uma análise completa das cadeias aninhadas de operações em ponto flutuante, das diferentes arquiteturas para obter a solução numérica de sistemas matriciais, será de caráter inedito porque não há trabalhos reportados na literatura.

3. Durante qualquer tipo de cálculo utilizado em computação é sempre necessário fazer alguns processos de arredondamento intermediarios antes de se poderem utilizar os resultados nos processos subsequentes. A maioria destes cálculos apresenta um problema bem conhecido, denominado de duplo arredondamento. Quando muitos formatos em

ponto flutuante são suportados em um processo, torna-se difícil reconhecer o formato no qual as operações serão feitas. Isto faz que o resultado de uma sequência de operações aritméticas seja um procedimento difícil de predizer.

Uma classe de representação de números chamada de codificação 'RN" (*Round to Nearest*) foi introduzida em [135], permitindo que o arredondamento seja feito pelo simples truncamento, com uma propriedade adicional, que permite que os problemas de duplo arredondamento sejam evitados. Em cálculos matriciais, este tipo de erro (o duplo arredondamento) gera instabilidade nos resultados finais, tornado o sistema computacional mais lento onde são executados métodos com critérios de parada baseados no erro.

Neste contexto, uma implementação de novas bibliotecas aritméticas, baseadas na representação RN, seria um passo chave no projeto em *hardware* de novas arquiteturas para solução numérica de sistemas matriciais. Visando uma redução na propagação do erro associado ao duplo arredondamento assim como, melhorar o desempenho dos algoritmos onde será usada esta representação numérica.

Por fim, pode-se ressaltar que a propagação do erro tem um papel na limitação do tamanho do sistema com matrizes uma vez que o mesmo cresce enquanto o sistema aumenta de tamanho. Na revisão bibliográfica não foi encontrado nenhum resultado considerando a propagação do erro; isso devido à generalização e dificuldade de usar um formato de representação numérico diferente do IEEE Std. 754, impedindo assim uma comparação com outra representação numérica.

4. O circuito de MDTM é a parte crítica do projeto total, dado que internamente ele precisa realizar a transposta de uma matriz. Isto reduz o desempenho da arquitetura uma vez que o número de FLOPS decresse devido à latência no processo de armazenamento, de todos os

elementos da matriz de dados. Uma melhoria imediata do desempenho do circuito estaria baseada na implementação de uma arquitetura MDTM capaz de lidar de forma ótima ou de uma melhor forma com a operação da transposta da matriz.

5. Com o incremento das características tecnológicas das FPGAs, é fácil encontrar várias vantagens arquitetônicas dentro de cada empresa fornecedora de *hardware* . Entre essas vantagens podem-se citar os multiplicadores DSP internos assim como o tamanho máximo de palavra que pode chegar a operar na arquitetura, como no caso da Xilinx que usa multiplicadores de 25×18 vs. Altera (algumas famílias) 27×27. Essas vantagens podem representar grande diferença em desempenho e uma comparação das arquiteturas aqui implementadas com a Xilinx vs Altera poderia demonstrá-lo.

Referências Bibliográficas

[1] BOYER, C. B.; MERZBACH, U. C. *A History of Mathematics*. [S.l.]: John Wiley & Sons, Inc., 2011.

[2] CAYLEY, A. A memoir on the theory of matrices. *Philosophical Transactions of the Royal Society of London*, The Royal Society, v. 148, p. pp. 17–37, 1858. ISSN 02610523. Disponível em: <http://www.jstor.org/stable/108649>.

[3] KANGSHEN JOHN N. CROSSLEY, A. W. C. L. S. *The Nine Chapters on the Mathematical Art: Companion and Commentary*. [S.l.]: Oxford University Press, 2000.

[4] THOMAS, M. *The Theory of Determinants in the Historical Order of Development*. [S.l.]: New York Dover Publications, 1960.

[5] HIGHAM, N. J. *Functions of Matrices: Theory and Computation*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2008. xx+425 p. ISBN 978-0-898716-46-7.

[6] STRIKWERDA, J. *Finite Difference Schemes and Partial Differential Equations*. [S.l.]: SIAM: Society for Industrial and Applied Mathematics, 2004.

[7] BURDEN, R. L.; FAIRES, J. D. *Numerical Analysis*. [S.l.]: Cengage Learning, 2010.

[8] CHARNEY, J.; FJöRTOFT, R.; NEUMANN, J. Numerical integration of the barotropic vorticity equation. *Tellus A*, v. 2, n. 4, 1950. ISSN 1600-0870. Disponível em: <http://www.tellusa.net/index.php/tellusa/article/view/8607>.

[9] ANDERSON, E. et al. *LAPACK Users' Guide*. Third. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999. ISBN 0-89871-447-8 (paperback).

[10] ALLAIRE, G.; KABER, S. M.; TRABELSI, K. *Numerical Linear Algebra*. [S.l.]: Springer, 2008.

[11] GATHEN, J. von zur; GERHARD, J. *Modern Computer Algebra*. Cambridge University Press, 2003.

[12] MATULA, D.; KORNERUP, P. Finite precision rational arithmetic: Slash number systems. *Computers, IEEE Transactions on*, C-34, n. 1, p. 3–18, 1985. ISSN 0018-9340.

[13] KORNERUP, P.; MATULA, D. Finite precision lexicographic continued fraction number systems. In: *Computer Arithmetic (ARITH), 1985 IEEE 7th Symposium on*. [S.l.: s.n.], 1985. p. 207–213.

[14] OLVER, F. W. J.; TURNER, P. R. Implementation of level-index arithmetic using partial table look-up. In: *Computer Arithmetic (ARITH), 1987 IEEE 8th Symposium on*. [S.l.: s.n.], 1987. p. 144–147.

[15] VUILLEMIN, J. Exact real computer arithmetic with continued fractions. *Computers, IEEE Transactions on*, v. 39, n. 8, p. 1087–1105, 1990. ISSN 0018-9340.

[16] VUILLEMIN, J. E. On circuits and numbers. *IEEE Transactions on Computers*, v. 43, p. 868–879, 1994.

[17] DAVIS, T. A. *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms)*. [S.l.]: Society for Industrial and Applied Mathematic, 2006.

[18] BERTERO, P. B. M. *Introduction to Inverse Problems in Imaging*. [S.l.]: CRC Press, 2010.

[19] KAIPIO, J.; SOMERSALO, E. *Statistical and Computational Inverse Problems (Applied Mathematical Sciences)*. [S.l.]: Springer, 2005.

[20] VELHO, H. F. d. C.; RAMOS, F. M. Numerical inversion of two-dimensional geoelectric conductivity distributions from magnetotelluric data. *Revista Brasileira de GeofÃsica*, scielo, v. 15, p. 133 – 144, 07 1997. ISSN 0102-261X.

[21] BECK, J. V.; BLACKWELL, B.; JR., C. R. S. C. *Inverse Heat Conduction: Ill-Posed Problems*. [S.l.]: Wiley-Interscience, 1985.

[22] GORDON, C. M. *The Square Root Function of a Matrix*. DissertaÃ§Ã£o (Mestrado) — Deparment of Mathematics and Statistics - Georgia State University, 2007.

[23] BJÖRCK, A.; HAMMARLING, S. A Schur method for the square root of a matrix. *Linear Algebra and its Applications*, v. 52-53, n. 0, p. 127 – 140, 1983. ISSN 0024-3795. Disponível em: <http://www.sciencedirect.com/science/article/pii/002437958380010X>.

[24] GOLUB, G. H.; LOAN, C. F. van V. *Matrix Computations*. 3rd. ed. The Johns Hopkins University Press, 1996.

[25] TREFETHEN, L. N.; BAU, I. D. *Numerical Linear Algebra*. [S.l.]: SIAM: Society for Industrial and Applied Mathematics, 1997.

[26] KULISCH, U.; ARCHIVES, T. P. S. U. C. The fifth floating-point operation for top-performance computers or accumulation of floating-point. *unknown*, unknown, 1998. Disponível em: <http://citeseer.ist.psu.edu/465324.html>.

[27] SUN, S.; ZAMBRENO, J. A floating-point accumulator for fpga-based high performance computing applications. In: *Field-Programmable Technology, 2009. FPT 2009. International Conference on*. [S.l.: s.n.], 2009. p. 493–499.

[28] CONSTANTINIDES, G.; KINSMAN, A.; NICOLICI, N. Numerical data representations for fpga-based scientific computing. *Design Test of Computers, IEEE*, v. 28, n. 4, p. 8–17, 2011. ISSN 0740-7475.

[29] KILTS, S. *Advanced FPGA Design: Architecture, Implementation, and Optimization*. [S.l.]: Wiley-IEEE Press, 2007.

[30] MEYER-BAESE, U. *Digital Signal Processing with Field Programmable Gate Arrays (Signals and Communication Technology)*. [S.l.]: Springer, 2007.

[31] WANG, X. *Variable Precision Floating-point Divide and Square Root for Eficient FPGA Implementation of Image and Signal Processing Algorithms*. Tese (Doutorado) — The Department of Electrical and Computer Engineering, Northeastern University, Boston, Massachusetts, USA, 2007.

[32] UNKNOWN. Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, p. 1–58, 2008.

[33] MARTIN-DOREL, É.; MELQUIOND, G.; MULLER, J.-M. *Some issues related to double roundings*. [S.l.], nov. 2011. 42 p. Disponível em: <http://hal-ens-lyon.archives-ouvertes.fr/ensl-00644408>.

[34] MULLER, J.-M. et al. *Handbook of Floating-Point Arithmetic*. [S.l.]: Birkhäuser Boston, 2010. 572 p. ACM G.1.0; G.1.2; G.4; B.2.0; B.2.4; F.2.1., ISBN 978-0-8176-4704-9.

[35] FIGUEROA, S. A. When is double rounding innocuous? *SIGNUM Newsl.*, ACM, New York, NY, USA, v. 30, n. 3, p. 21–26, jul. 1995. ISSN 0163-5778. Disponível em: <http://doi.acm.org/10.1145/221332.221334>.

[36] CID, S. A. F. del. *A Rigorous Framework for Fully Supporting the IEEE Standard for Floating-Point Arithmetic in High-Level Programming Languages*. Tese (Doutorado) — Department of Computer Science, New York University, 2000.

[37] ECHMAN, F.; ÖWALL, V. A scalable pipelined complex valued matrix inversion architecture. In: *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*. [S.l.: s.n.], 2005. p. 4489–4492 Vol. 5.

[38] RAFIQUE, A.; KAPRE, N.; CONSTANTINIDES, G. Enhancing performance of tall-skinny qr factorization using fpgas. In: *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*. [S.l.: s.n.], 2012. p. 443–450.

[39] LIN, K.-H. et al. Iterative qr decomposition architecture using the modified gram-schmidt algorithm. In: *Circuits and Systems, 2009. ISCAS 2009. IEEE International Symposium on*. [S.l.: s.n.], 2009. p. 1409–1412.

[40] BOLAND, D.; CONSTANTINIDES, G. An fpga-based implementation of the minres algorithm. In: *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*. [S.l.: s.n.], 2008. p. 379 –384.

[41] BURKS, A. W.; GOLDSTINE, H. H.; NEUMANN, J. von. *Perspectives on the computer revolution (Preliminary discussion of the logical design of an electronic computing instrument - 1946)*. Norwood, NJ, USA: Ablex Publishing Corp., 1989. 39–48 p. ISBN 0-89391-369-3.

[42] PATTERSON, D. A.; HENNESSY, J. L. *Computer Organization and Design: The Hardware/Software Interface*. [S.l.]: The Morgan Kaufmann Series in Computer Architecture and Design, 2013.

[43] WULF, W. A.; MCKEE, S. A. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, ACM, New York, NY, USA, v. 23, n. 1, p. 20–24, mar. 1995. ISSN 0163-5964. Disponível em: <http://doi.acm.org/10.1145/216585.216588>.

[44] HARTENSTEIN, R. Are we really ready for the breakthrough? [morphware]. In: *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*. [S.l.: s.n.], 2003. p. 7 pp.–. ISSN 1530-2075.

[45] XILINX. *7 Series CLB Architecture*. Disponível em: <http://www.xilinx.com/training/fpga/7_series_CLB_architecture_video.htm>.

[46] HUTTON, M. et al. A methodology for fpga to structured-asic synthesis and verification. In: *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*. [S.l.: s.n.], 2006. v. 2, p. 1–6.

[47] HAMBLEN, J. O.; FURMAN, M. D. *Rapid Prototyping of Digital Systems*. [S.l.]: Springer, 1999.

[48] HO, S. et al. Structured asic: Methodology and comparison. In: *Field-Programmable Technology (FPT), 2010 International Conference on*. [S.l.: s.n.], 2010. p. 377–380.

[49] TSAI, Y.-W. et al. Using structured asic to improve design productivity. In: *Integrated Circuits, ISIC '09. Proceedings of the 2009 12th International Symposium on*. [S.l.: s.n.], 2009. p. 25–28.

[50] HARTENSTEIN, R. Why we need reconfigurable computing education. *the 1st International Workshop on Reconfigurable Computing Education*, 2006.

[51] HARTENSTEIN, R. Coarse grain reconfigurable architecture (embedded tutorial). In: *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*. New York, NY, USA: ACM, 2001. (ASP-DAC '01), p. 564–570. ISBN 0-7803-6634-4. Disponível em: <http://doi.acm.org/10.1145/370155.370535>.

[52] FERDOUS, T. Design and fpga-based implementation of a high performance 32-bit dsp processor. In: *Computer and Information Techno-*

logy (ICCIT), 2012 15th International Conference on. [S.l.: s.n.], 2012. p. 484–489.

[53] DAGA, M.; SCOGLAND, T.; FENG, W. chun. Architecture-aware mapping and optimization on a 1600-core gpu. In: *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*. [S.l.: s.n.], 2011. p. 316–323. ISSN 1521-9097.

[54] KUNGT, H. T.; LEISERSONT, C. E. Systolic arrays (for vlsi). *Society for Industrial\ & Applied*, p. 256, 1979.

[55] KUNG, S.-Y. Vlsi array processors. *ASSP Magazine, IEEE*, v. 2, n. 3, p. 4–22, Jul 1985. ISSN 0740-7467.

[56] YOKOYAMA, Y.; KIM, M.; ARAI, H. Implementation of systolic rls adaptive array using fpga and its performance evaluation. In: *Vehicular Technology Conference, 2006. VTC-2006 Fall. 2006 IEEE 64th.* [S.l.: s.n.], 2006. p. 1–5.

[57] STOJANOVIC, N. et al. Matrix-vector multiplication on a fixed size unidirectional systolic array. In: *Telecommunications in Modern Satellite, Cable and Broadcasting Services, 2007. TELSIKS 2007. 8th International Conference on*. [S.l.: s.n.], 2007. p. 457–460.

[58] SNOPCE, H.; ELMAZI, L. The importance of using the linear transformation matrix in determining the number of processing elements in 2-dimensional systolic array for the algorithm of matrix-matrix multiplication. In: *Information Technology Interfaces, 2008. ITI 2008. 30th International Conference on*. [S.l.: s.n.], 2008. p. 885–892. ISSN 1330-1012.

[59] LU, L. et al. Qca systolic array design. *Computers, IEEE Transactions on*, v. 62, n. 3, p. 548–560, March 2013. ISSN 0018-9340.

[60] HARTENSTEIN, R. *The von Neumann Syndrome*. [S.l.], 2007. Disponível em: <http://www.academia.edu/4816058/The_von_Neumann_Syndrome>.

[61] KULKARNI, A.; YEN, D. Systolic processing and an implementation for signal and image processing. *Computers, IEEE Transactions on*, C-31, n. 10, p. 1000–1009, Oct 1982. ISSN 0018-9340.

[62] ATOCHE, A. C.; AGUILAR, J.; CASTILLO, J. Systolic array implementations for real time enhancement of remote sensing imaging.

In: *Programmable Logic, 2009. SPL. 5th Southern Conference on*. [S.l.: s.n.], 2009. p. 59–64.

[63] SUDHA, N.; MOHAN, A.; MEHER, P. A self-configurable systolic architecture for face recognition system based on principal component neural network. *Circuits and Systems for Video Technology, IEEE Transactions on*, v. 21, n. 8, p. 1071–1084, Aug 2011. ISSN 1051-8215.

[64] MAMATHA, I. et al. Reduced complexity architecture for convolution based discrete cosine transform. In: *Electronic System Design (ISED), 2013 International Symposium on*. [S.l.: s.n.], 2013. p. 67–71.

[65] WAKABA, Y. et al. An efficient hardware matching engine for regular expression with nested kleene operators. In: *Field Programmable Logic and Applications (FPL), 2011 International Conference on*. [S.l.: s.n.], 2011. p. 157–161.

[66] MOSLEH, M. et al. Fpga implementation of a linear systolic array for speech recognition based on hmm. In: *Computer and Automation Engineering (ICCAE), 2010 The 2nd International Conference on*. [S.l.: s.n.], 2010. v. 3, p. 75–78.

[67] SINGH, S.; HAN, J.-Y. Systolic arrays-warp speed ahead for compute-bound problems using systolic arrays. *Potentials, IEEE*, v. 10, n. 1, p. 7–11, Feb 1991. ISSN 0278-6648.

[68] JAIN, S.; POTTATHUPARAMBIL, R.; SASS, R. Implications of memory-efficiency on sparse matrix-vector multiplication. In: *Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on*. [S.l.: s.n.], 2011. p. 80–83.

[69] CARDOSO, J. M. P.; HüBNER, e. a. M. *Reconfigurable Computing: From FPGAs to Hardware/Software Codesign*. [S.l.]: Springer New York, 2011.

[70] ANTON, H.; RORRES, C. *Álgebra Linear Com Aplicações*. BOOKMAN COMPANHIA ED, 2001.

[71] KOLMAN, B.; HILL, D. R. *Introdução à Álgebra Linear Com Aplicações*. LTC ED, 2001.

[72] LEVINGER, B. W. The square root of a 2×2 matrix. *Mathematics Magazine*, Mathematical Association of America, v. 53,

n. 4, p. pp. 222–224, 1980. ISSN 0025570X. Disponível em: <http://www.jstor.org/stable/2689616>.

[73] HIGHAM, N. Stable iterations for the matrix square root. *Numerical Algorithms*, Kluwer Academic Publishers, v. 15, n. 2, p. 227–242, 1997. ISSN 1017-1398.

[74] HIGHAM, N. J. *Functions of Matrices: Theory and Computation.* [S.l.]: Society for Industrial & Applied Mathematics, 2008. 445 p.

[75] MATHIAS, R. Condition estimation for matrix functions via the schur decomposition. *SIAM J. Matrix Anal. Appl*, v. 16, p. 565–578, 1997.

[76] MATOS, G. de; NETO, H. On reconfigurable architectures for efficient matrix inversion. In: *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on.* [S.l.: s.n.], 2006. p. 1 –6.

[77] MATOS, G. de; NETO, H. Memory optimized architecture for efficient gauss-jordan matrix inversion. In: *Programmable Logic, 2007. SPL '07. 2007 3rd Southern Conference on.* [S.l.: s.n.], 2007. p. 33 –38.

[78] IRTURK, A. et al. Automatic generation of decomposition based matrix inversion architectures. In: *ICECE Technology, 2008. FPT 2008. International Conference on.* [S.l.: s.n.], 2008. p. 373–376.

[79] BURIAN, A.; TAKALA, J.; YLINEN, M. A fixed-point implementation of matrix inversion using cholesky decomposition. In: *Micro-NanoMechatronics and Human Science, 2003 IEEE International Symposium on.* [S.l.: s.n.], 2003. v. 3, p. 1431 – 1434 Vol. 3. ISSN 1548-3746.

[80] HAPPONEN, A.; PIIRAINEN, O.; BURIAN, A. Gsm channel estimator using a fixed-point matrix inversion algorithm. In: *Signals, Circuits and Systems, 2005. ISSCS 2005. International Symposium on.* [S.l.: s.n.], 2005. v. 1, p. 119 – 122 Vol. 1.

[81] SALMELA, P. et al. Several approaches to fixed-point implementation of matrix inversion. In: *Signals, Circuits and Systems, 2005. ISSCS 2005. International Symposium on.* [S.l.: s.n.], 2005. v. 2, p. 497 – 500 Vol. 2.

[82] DUARTE, R.; NETO, H.; VESTIAS, M. Double-precision gauss-jordan algorithm with partial pivoting on fpgas. In: *Digital System Design, Architectures, Methods and Tools, 2009. DSD '09. 12th Euromicro Conference on.* [S.l.: s.n.], 2009. p. 273 –280.

[83] EL-AMAWY, A.; DHARMARAJAN, K. Parallel vlsi algorithm for stable inversion of dense matrices. *Computers and Digital Techniques, IEE Proceedings E*, v. 136, n. 6, p. 575 – 580, nov 1989. ISSN 0143-7062.

[84] SINGH, C. K.; PRASAD, S. H.; BALSARA, P. T. Vlsi architecture for matrix inversion using modified gram-schmidt based qr decomposition. In: *VLSI Design, 2007. Held jointly with 6th International Conference on Embedded Systems., 20th International Conference on.* [S.l.: s.n.], 2007. p. 836 –841. ISSN 1063-9667.

[85] EILERT, J.; WU, D.; LIU, D. Efficient complex matrix inversion for mimo software defined radio. In: *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on.* [S.l.: s.n.], 2007. p. 2610 – 2613.

[86] WU, D. et al. Fast complex valued matrix inversion for multi-user stbc-mimo decoding. In: *VLSI, 2007. ISVLSI '07. IEEE Computer Society Annual Symposium on.* [S.l.: s.n.], 2007. p. 325 –330.

[87] KARKOOTI, M.; CAVALLARO, J.; DICK, C. Fpga implementation of matrix inversion using qrd-rls algorithm. In: *Signals, Systems and Computers, 2005. Conference Record of the Thirty-Ninth Asilomar Conference on.* [S.l.: s.n.], 2005. p. 1625 – 1629. ISSN 1058-6393.

[88] ALONSO, R.; LUCIO, D. Parallel architecture for the solution of linear equation systems implemented in fpga. In: *Electronics, Robotics and Automotive Mechanics Conference, 2009. CERMA '09.* [S.l.: s.n.], 2009. p. 275 –280.

[89] MARTINEZ-ALONSO, R.; MINO, K.; TORRES-LUCIO, D. Array processors designed with vhdl for solution of linear equation systems implemented in a fpga. In: *Electronics, Robotics and Automotive Mechanics Conference (CERMA), 2010.* [S.l.: s.n.], 2010. p. 731–736.

[90] ARIAS-GARCIA, J. et al. A fast and low cost architecture developed in fpgas for solving systems of linear equations. In: *Circuits and Systems (LASCAS), 2012 IEEE Third Latin American Symposium on.* [S.l.: s.n.], 2012. p. 1–4.

[91] LEDESMA-CARRILLO, L. M. et al. Reconfigurable fpga-based unit for singular value decomposition of large m x n matrices. In: *Proceedings of the 2011 International Conference on Reconfigurable Computing and FPGAs.* Washington, DC, USA: IEEE Computer Society, 2011.

(RECONFIG '11), p. 345–350. ISBN 978-0-7695-4551-6. Disponível em: <http://dx.doi.org/10.1109/ReConFig.2011.77>.

[92] MORRIS, G.; PRASANNA, V.; ANDERSON, R. A hybrid approach for mapping conjugate gradient onto an fpga-augmented reconfigurable supercomputer. In: *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*. [S.l.: s.n.], 2006. p. 3 –12.

[93] MORRIS, G. R.; PRASANNA, V. K. Sparse matrix computations on reconfigurable hardware. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 40, n. 3, p. 58–64, mar. 2007. ISSN 0018-9162. Disponível em: <http://dx.doi.org/10.1109/MC.2007.103>.

[94] DUBOIS, D. et al. Non-preconditioned conjugate gradient on cell and fpga based hybrid supercomputer nodes. In: *Field Programmable Custom Computing Machines, 2009. FCCM '09. 17th IEEE Symposium on*. [S.l.: s.n.], 2009. p. 201 –208.

[95] ROLDAO, A.; CONSTANTINIDES, G. A. A high throughput fpga-based floating point conjugate gradient implementation for dense matrices. *ACM Trans. Reconfigurable Technol. Syst.*, ACM, New York, NY, USA, v. 3, n. 1, p. 1:1–1:19, jan. 2010. ISSN 1936-7406. Disponível em: <http://doi.acm.org/10.1145/1661438.1661439>.

[96] RAFIQUE, A.; KAPRE, N.; CONSTANTINIDES, G. A high throughput fpga-based implementation of the lanczos method for the symmetric extremal eigenvalue problem. In: CHOY, O. et al. (Ed.). *Reconfigurable Computing: Architectures, Tools and Applications*. [S.l.]: Springer Berlin Heidelberg, 2012, (Lecture Notes in Computer Science, v. 7199). p. 239–250.

[97] GREISEN, P. et al. Evaluation and fpga implementation of sparse linear solvers for video processing applications. *Circuits and Systems for Video Technology, IEEE Transactions on*, v. 23, n. 8, p. 1402–1407, 2013. ISSN 1051-8215.

[98] GONZALEZ, J.; NúñEZ, R. C. Lapackrc: Fast linear algebra kernels/solvers for fpga accelerators. *Journal of Physics: Conference Series*, v. 180, n. 1, p. 012042, 2009. Disponível em: <http://stacks.iop.org/1742-6596/180/i=1/a=012042>.

[99] ARIAS-GARCIA, J. et al. A suitable fpga implementation of floating-point matrix inversion based on gauss-jordan elimination. In: *Programmable Logic (SPL), 2011 VII Southern Conference on.* [S.l.: s.n.], 2011. p. 263 –268.

[100] SÁNCHEZ, D. et al. Parameterizable floating-point library for arithmetic operations in fpgas. In: *in ACM International Symposium on Integrated Circuits and System Design.* [S.l.: s.n.], 2009.

[101] ARIAS-GARCIA, J. et al. Fpga implementation of large-scale matrix inversion using single, double and custom floating-point precision. In: *Programmable Logic (SPL), 2012 VIII Southern Conference on.* [S.l.: s.n.], 2012. p. 1–6.

[102] PETKOV, N. *Systolic Parallel Processing.* [S.l.]: ELSEVIER SCIENCE PUBLISHER B.V, 1993.

[103] ARIAS-GARCIA, J. et al. Fpga hil simulation of a linear system block for strongly coupled system applications. In: *Industrial Technology (ICIT), 2013 IEEE International Conference on.* [S.l.: s.n.], 2013. p. 1017–1022.

[104] CHE, S. et al. Accelerating compute-intensive applications with gpus and fpgas. In: *Application Specific Processors, 2008. SASP 2008. Symposium on.* [S.l.: s.n.], 2008. p. 101–107.

[105] XILINX. *Xilinx System Generator for DSP.* Disponível em: <www.xilinx.com/ise/optional prod/system generator.htm>.

[106] BRAGA, A. et al. Hardware implementation of gmdh-type artificial neural networks and its use to predict approximate three-dimensional structures of proteins. In: *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on.* [S.l.: s.n.], 2012. p. 1–8.

[107] BRAGA, A. L. S. *Redes Neurais Baseadas no Método de Grupo de Manipulação de Dados: Treinamento, Implementações e aplicações.* Tese (Doutorado) — Universidade de Brasília, 2013.

[108] M, M. D. et al. Tradeoff of fpga design of a floating-point library for arithmetic operators. *International Journal of Integrated Circuits and Systems*, v. 5, p. 42–52, 2010.

[109] GAJSKI, D. D. et al. *Embedded System Design: Modeling, Synthesis and Verification.* [S.l.]: Springer, 2009.

[110] TSAI, W.-T. et al. Rapid embedded system testing using verification patterns. *Software, IEEE*, v. 22, n. 4, p. 68–75, July 2005. ISSN 0740-7459.

[111] STAFF, T. *An Independent Analysis of Floating-point DSP Design Flow and Performance on Altera 28-nm FPGAs.* [S.l.], 2012.

[112] MUñOZ, D. M. *Otimização por Inteligência de Enxames Usando Arquiteturas Paralelas para Aplicações Embarcadas.* Tese (Doutorado) — Universidade de Brasília, 2012. Disponível em: <http://hdl.handle.net/10482/13055>.

[113] BACHIR, T.; DAVID, J.-P. Performing floating-point accumulation on a modern fpga in single and double precision. In: *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on.* [S.l.: s.n.], 2010. p. 105–108.

[114] HOKENEK, E.; MONTOYE, R.; COOK, P. Second-generation risc floating point with multiply-add fused. *Solid-State Circuits, IEEE Journal of*, v. 25, n. 5, p. 1207–1213, Oct 1990. ISSN 0018-9200.

[115] LUO, Z.; MARTONOSI, M. Accelerating pipelined integer and floating-point accumulations in configurable hardware with delayed addition techniques. *Computers, IEEE Transactions on*, v. 49, n. 3, p. 208–218, 2000. ISSN 0018-9340.

[116] VANGAL, S. et al. A 5 ghz floating point multiply-accumulator in 90 nm dual vt cmos. In: *Solid-State Circuits Conference, 2003. Digest of Technical Papers. ISSCC. 2003 IEEE International.* [S.l.: s.n.], 2003. p. 334–497 vol.1. ISSN 0193-6530.

[117] BODNAR, M. et al. Floating-point accumulation circuit for matrix applications. In: *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on.* [S.l.: s.n.], 2006. p. 303–304.

[118] ZHUO, L.; MORRIS, G.; PRASANNA, V. High-performance reduction circuits using deeply pipelined operators on fpgas. *Parallel and Distributed Systems, IEEE Transactions on*, v. 18, n. 10, p. 1377–1392, 2007. ISSN 1045-9219.

[119] DINECHIN, F. de et al. An fpga-specific approach to floating-point accumulation and sum-of-products. In: *ICECE Technology, 2008. FPT 2008. International Conference on.* [S.l.: s.n.], 2008. p. 33–40.

[120] LIEBIG, B.; HUTHMANN, J.; KOCH, A. Architecture exploration of high-performance floating-point fused multiply-add units and their automatic use in high-level synthesis. In: *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International.* [S.l.: s.n.], 2013. p. 134–143.

[121] QUINNELL, E. C. *Floating-Point Fused Multiply-Add Architectures.* Tese (Doutorado) — The University of Texas at Austin, 2007.

[122] SAAD, Y. *Iterative Methods for Sparse Linear Systems.* [S.l.]: The Society for Industrial and Applied Mathematics, 2003.

[123] CHANG, R.-H. et al. Iterative QR Decomposition Architecture Using the Modified Gram-Schmidt Algorithm for MIMO Systems. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, v. 57, n. 5, p. 1095–1102, May 2010. ISSN 1549-8328.

[124] MEYER, C. D. *Matrix Analysis and Applied Linear Algebra with solutions.* [S.l.]: SIAM: Society for Industrial and Applied Mathematics, 2000.

[125] ABELS, M.; WIEGAND, T.; PAUL, S. Efficient FPGA implementation of a High throughput systolic array QR-decomposition algorithm. In: *Signals, Systems and Computers (ASILOMAR), 2011 Conference Record of the Forty Fifth Asilomar Conference on.* [S.l.: s.n.], 2011. p. 904–908. ISSN 1058-6393.

[126] ASLAN, S.; NIU, S.; SANIIE, J. FPGA implementation of fast QR decomposition based on givens rotation. In: *Circuits and Systems (MWSCAS), 2012 IEEE 55th International Midwest Symposium on.* [S.l.: s.n.], 2012. p. 470–473. ISSN 1548-3746.

[127] CERVANTES-LOZANO, P.; GONZALEZ-PEREZ, L.; GARCIA-GARCIA, A. A VLSI architecture for the QR decomposition based on the MCGR algorithm. In: *Reconfigurable Computing and FPGAs (ReConFig), 2013 International Conference on.* [S.l.: s.n.], 2013. p. 1–6.

[128] HAN, B.; YANG, Z.; ZHENG, Y. FPGA implementation of QR decomposition for MIMO-OFDM using four CORDIC cores. In: *Com-*

munications (ICC), 2013 IEEE International Conference on. [S.l.: s.n.], 2013. p. 4556–4560. ISSN 1550-3607.

[129] NIU, S. et al. Hardware and software design for QR Decomposition Recursive Least Square algorithm. In: *Circuits and Systems (MWSCAS), 2013 IEEE 56th International Midwest Symposium on.* [S.l.: s.n.], 2013. p. 117–120. ISSN 1548-3746.

[130] CHAUHAN, A.; MEHRA, R. Analysis of QR Decomposition for MIMO Systems. In: *Electronic Systems, Signal Processing and Computing Technologies (ICESC), 2014 International Conference on.* [S.l.: s.n.], 2014. p. 69–73.

[131] TAI, Y.-G.; LO, C.-T. D.; PSARRIS, K. Applying Out-of-Core QR Decomposition Algorithms on FPGA-Based Systems. In: *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on.* [S.l.: s.n.], 2007. p. 86–91.

[132] WANG, X.; LEESER, M. Efficient FPGA Implementation of QR Decomposition Using a Systolic Array Architecture. In: *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays.* New York, NY, USA: ACM, 2008. (FPGA '08), p. 260–260. ISBN 978-1-59593-934-0. Disponível em: <http://doi.acm.org/10.1145/1344671.1344718>.

[133] DOU, Y. et al. FPGA accelerating three QR decomposition algorithms in the unified pipelined framework. In: *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on.* [S.l.: s.n.], 2009. p. 410–416. ISSN 1946-1488.

[134] FORSYTHE, G. E.; MOLER, C. *Computer Solution of Linear Algebraic Systems.* [S.l.]: Prentice-Hall, 1967.

[135] KORNERUP, P. Rn-coding of numbers: definition and some properties. In: *in "Proceedings of the 17th IMACS World Congress on Scientific Computation, Applied Mathematics and Simulation.* [S.l.: s.n.], 2004.

APÊNDICES

Apêndice A PUBLICAÇÔES REALIZADAS

A.1 TRABALHOS PUBLICADOS

A.1.1 Arquiteturas para inversão de matrizes

- Arias-Garcia, J. and Jacobi, R.P. and Llanos, C.H. and Ayala-Rincon, M., A suitable FPGA implementation of floating-point matrix inversion based on Gauss-Jordan elimination. In: Proc. International Southern Conference on Programmable Logic (SPL), Córdoba, Argentina, 2011, pp.263-268.

  Abstract:This work presents an architecture to compute matrix inversions in a hardware reconfigurable FPGA with single-precision floating-point representation, whose main unit is the processing component for Gauss-Jordan elimination. This component consists of other smaller arithmetic units, organized to maintain the accuracy of the results without the need to internally normalize and de-normalize the floating-point data. The implementation of the operations and the whole unit take advantage of the resources available in the Virtex-5 FPGA. The performance and resource consumption of the implementation are improvements in comparison with different more elaborated architectures whose implementations are more complex for low cost applications. Benchmarks are done with solutions implemented previously in FPGA and software, such as Matlab

- Arias-Garcia, J.; Llanos, C.H.; Ayala-Rincon, M.; Jacobi, R.P., FPGA implementation of large-scale matrix inversion using single, double and custom floating-point precision. In: Proc. International Southern Conference on Programmable Logic (SPL), Bento Gonçalves, Brazil, 2012, pp.1,6.

  Abstract: This work presents an architecture to compute matrix inversions in a hardware reconfigurable FPGA using different floating-point representation precision: single, double and 40-bits. The architectural approach is divided into five principal parts, four modules

and one unit, namely Change Row Module, Pivo Module, Matrix Elimination Module, Normalization Module and finally the Gauss-Jordan Control-Circuit Unit. This division allows the work with other smaller arithmetic units that are organized in order to maintain the accuracy of the results without the need to internally normalize and de-normalize the floatingpoint data. The implementation of the operations and the whole units take advantage of the resources available in the Virtex-5 FPGA. The error propagation and resource consumption of the implementation, specially the internal RAM memory blocks that are used, constitute improvements when compared with previous work of the authors and other more elaborated architectures whose implementations are significantly more complex than the current one and thus unsuitable for its application. The approach is validated by implementing benchmarks based on solutions in FPGA and software (e.g. Matlab) implemented previously.

A.1.2 Arquiteturas para solução de sistemas de equações lineares

- Arias-Garcia, J.; Llanos, C.H.; Ayala-Rincon, M.; Jacobi, R.P., A fast and low cost architecture developed in FPGAs for solving systems of linear equations. In: Proc. International IEEE Third Latin American Symposium on Circuits and Systems (LASCAS), Playa del Carmen, Mexico, 2012, pp.1,4.

  Abstract: This paper presents a low cost architecture for the solution of linear equations based on the Gaussian Elimination Method using a reconfigurable system based on FPGA. This architecture can handle single data precision that follows the IEEE 754 floating point standard. The implementation takes advantage of both the internal memory and the DSP blocks (available in the Virtex-5 FPGA). The architectural approach is composed of four modules and one specific unit (namely, Change Row Module, Pivo Module, FB Module, Normalization Module and finally the Gaussian Elimination Controller Unit). This structure can be combined with other smaller arithmetic units in order to maintain the accuracy of the results without the need to internally normalize and de-normalize the floatingpoint data. Also, a special Memory Access Unit was implemented in order to deal with the writing/reading operations to/from the internal RAM. The

resource consumption of the implementation (specially the internal RAM memory blocks that are used) points out several improvements when compared to previous work of the authors and other more elaborated architectures whose implementations are significantly more complex and, thus, unsuitable for low cost applications.

A.1.3 Simulação baseda em *Hardware in the loop*

- Arias-Garcia, J.; Braga, A; Llanos, C.H.; Ayala-Rincon, M.; Pezzuol Jacobi, R.; Foltran, A, FPGA HIL simulation of a linear system block for strongly coupled system application. In: Proc. IEEE International Conference on Industrial Technology (ICIT), Cape Town, South Africa, 2013, pp.1017,1022.

  Abstract: This paper introduces a hardware simulation flow that is based on the Xilinx System Generator Tool (XSG), of an architecture for solving dense linear systems, presented as a strongly coupled system, which is in turn based on Gaussian Elimination using an FPGA. A functional verification process is achieved by taking advantage of the XSG, allowing both software and hardware-in-the-loop (HIL) simulations and using the respective results achieved in Matlab as a reference model. The linear system block embedded in the XSG can handle single, double and custom data precision, following the IEEE 754 floating point standard. The current architecture improves the use of internal RAM of the selected FPGA device (Virtex-5) through a special Memory Access Unit, reducing the data access among this RAM and the different modules in the architecture. Examples of systems of six equations, which are suitable for some robotics applications, have been used for comparing the performance of the Linear System block and Matlab, the latter used as a statistical estimator, in order to validate the data results.

A.1.4 Aplicações da arquitetura de solução de um sistema de equações lineares

- Braga, AL.S.; Arias-Garcia, J.; Llanos, C.; Dorn, M.; Foltran, A; Coelho, L.S., Hardware implementation of GMDH-type artificial neu-

ral networks and its use to predict approximate three-dimensional structures of proteins. In: Proc International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC),York, United Kingdom, 2012, pp.1,8.

Abstract: Implementation of artificial neural networks in software on general purpose computer platforms are brought to an advanced level both in terms of performance and accuracy. Nonetheless, neural networks are not so easily applied in embedded systems, specially when the fully retraining of the network is required. This paper shows the results of the implementation of artificial neural networks based on the Group Method of Data Handling (GMDH) in reconfigurable hardware, both in the steps of training and running. A hardware architecture has been developed to be applied as a co-processing unit and an example application has been used to test its functionality. The application has been developed for the prediction of approximate 3-D structures of proteins. A set of experiments have been performed on a PC using the FPGA as a co-processor accessed through sockets over the TCP/IP protocol. The design flow employed demonstrated that it is possible to implement the network in hardware to be easily applied as an accelerator in embedded systems. The experiments show that the proposed implementation is effective in finding good quality solutions for the example problem. This work represents the early results of the novel technique of applying the GMDH algorithms in hardware for solving the problem of protein structures prediction.

A.2 PUBLICAÇÕES RELACIONADAS

- Yudi, J.; Arias-Garcia, J.; Sánchez-Ferreira, C.; Muñoz, D.M.; Llanos, C.; Motta J.M.S.T. An FPGA-based omnidirectional vision sensor for motion detection on mobile robots. In: Proc. International Journal of Reconfigurable Computing. Hindawi, v. 2012, 2012.

- Yudi, J.; Muñoz, D.M.; Arias-Garcia, J.; Llanos, C.; Motta J.M.S.T. FPGA based image processing for omnidirecctional vision on mobile robots. In: Proc. International Symposium on Integrated Circuits and System Design. João Pessoa, Brazil: ACM, 2011, p.113-118.

- Alves-Almeida, Ariane.; Arias-Garcia, Janier.; Llanos, C.; Ayala-

Rincon, M. Verification of Hardware Implementations through Correctness of their Recursive Definitions in PVS. In: Proc. International Symposium on Integrated Circuits and System Design, Aracaju, Brazil, 2014.

Apêndice B FUNDAMENTOS DE ÁLGEBRA LINEAR

Neste apêndice são introduzidos os conceitos, definições e alguns teoremas gerais considerados relevantes para uma boa compreensão do trabalho desenvolvido. Os conceitos e definições apresentadas nesta seção foram tomadas das seguintes referencias bibliográficas [70, 71].

B.1 Matrizes e Vetores

Neste trabalho serão adotadas convenções para manter uma organização simples e clara das definições discutidas nesta seção. Serão usadas letras maiúsculas em itálico para matrizes, A, letras maiúsculas cursivas em negrito para espaços vetoriais, $\boldsymbol{V}$, e letras minúsculas em itálico para vetores, x. Algumas definições importantes serão apresentadas a seguir,

***Definição* B.1** *Uma matriz $A^{m\times n}$ é um arranjo retangular de $m \times n$ números reais (ou complexos) distribuídos em m linhas horizontais e n colunas verticais (ver equação B.1),*

$$A_{ij} = \begin{pmatrix} a_{11} & a_{12} & \cdots & \cdots & a_{1j} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & \cdots & a_{2j} & \cdots & a_{2n} \\ \vdots & \vdots & \cdots & \cdots & \vdots & \cdots & \vdots \\ a_{i1} & a_{i2} & \cdots & \cdots & a_{ij} & \cdots & a_{in} \\ \vdots & \vdots & & & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & \cdots & a_{mj} & \cdots & a_{mn} \end{pmatrix} \tag{B.1}$$

em que a i-ésima linha de A e a j-ésima coluna de A são representadas segundo as expressões B.2 e B.3 respectivamente.

$$\begin{pmatrix} a_{i1} & a_{i2} & \cdots & a_{in} \end{pmatrix} (1 \leq i \leq m) \tag{B.2}$$

$$\begin{pmatrix} a_{1j} \\ a_{2j} \\ \vdots \\ a_{mj} \end{pmatrix} (1 \leq j \leq n) \tag{B.3}$$

Assim, A é dita uma matriz m por n (representado por $m \times n$). No caso de $m = n$, A é denominada de matriz quadrada de ordem n, onde os números $a_{11}, a_{22}, ..., a_{nn}$ formam a diagonal principal de A. Adicionalmente, o número a_{ij}, que está na i-ésima linha e na j-ésima coluna de A, é denominado de i,j-ésimo de A ou o elemento (i,j) de A. Frequentemente a equação B.1 é representada como $A = \left(a_{ij}\right)$.

***Definição* B.2** *Uma matriz quadrada $A = \left(a_{ij}\right)$ em que todo elemento fora da diagonal principal é zero, isto é, $a_{ij} = 0$ para $i \neq j$, é denominada de matriz diagonal.*

***Definição* B.3** *Uma matriz diagonal $A = \left(a_{ij}\right)$ em que todos os termos da diagonal principal são iguais, $a_{ij} = c$ para $i \equiv j$ e $a_{ij} = 0$ para $i \neq j$, é denominada de matriz escalar.*

***Definição* B.4** *Se $A = \left(a_{ij}\right)$ e $B = \left(b_{ij}\right)$ são matrizes $m \times n$, a soma de A e B é uma matriz $C = \left(c_{ij}\right)$, $m \times n$, definida pela equação B.4, em que $(1 \leq i \leq m, 1 \leq j \leq n)$. Portanto, C é obtida pela adição dos elementos correspondentes de A e B.*

$$c_{ij} = a_{ij} + b_{ij} \tag{B.4}$$

***Definição* B.5** *Se $A = \left(a_{ij}\right)$ é uma matriz $m \times n$ e r é um número real, a multiplicação por um escalar de A por r, rA, é a matriz $B = \left(b_{ij}\right)$, $m \times n$, em que $b_{ij} = ra_{ij}$ para $(1 \leq i \leq m, 1 \leq j \leq n)$. Assim, B é obtida pela multiplicação de cada elemento de A por r.*

***Definição* B.6** *Se* $A = \left(a_{ij}\right)$ *é uma matriz* $m \times n$*, então a matriz* $n \times m$*,* $A^T = \left(a_{ij}^T\right)$*, em que* $a_{ij}^T = a_{ji}$ *para* $(1 \leq i \leq n, 1 \leq j \leq m)$ *é chamada transposta de* A*. Dessa maneira, os elementos em cada linha de* A^T *são os elementos na coluna correspondente de* A*.*

***Definição* B.7** *O produto escalar ou produto interno dos vetores de dimensão* n $\boldsymbol{a}$ *e* $\boldsymbol{b}$ *é a soma dos produtos dos elementos correspondentes. Dessa forma, se* $a = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix}$ *e* $b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$ *então o produto escalar* $\boldsymbol{a} \times \boldsymbol{b}$ *está definido segundo a equação B.5.*

$$a \cdot b = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n = \sum_{i=1}^{n} a_i b_i \tag{B.5}$$

***Definição* B.8** *Se* $A = \left(a_{ij}\right)$ *é uma matriz* $m \times p$ *e* $B = \left(b_{ij}\right)$ *é uma matriz* $p \times n$*, o produto de* A *e* B *representado por* AB*, é a matriz* $C = \left(c_{ij}\right)$*,* $m \times n$*, definida pela equação B.6.*

$$C_{ij} = a_{i1} b_{1j} + a_{i2} b_{2j} + \cdots + a_{ip} b_{pj} = \sum_{k=1}^{p} a_{ik} b_{kj} \qquad (1 \leq i \leq m, 1 \leq j \leq n) \tag{B.6}$$

***Definição* B.9** *A matriz escalar* $n \times n$*, cujos elementos da diagonal principal são todos iguais a* 1 *(ver expressão B.7), é denominada de matriz identidade de ordem* n*.*

$$\begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \\ 0 & 0 & \cdots & 1 \end{pmatrix} \tag{B.7}$$

***Definição* B.10** *Uma matriz A $m \times n$ está na forma escalonada reduzida por linhas se a mesma satisfaz as seguintes propriedades:*

1. *Todas as linhas nulas, se existirem, ocorrem abaixo de todas as linhas não-nulas.*
2. *O primeiro elemento diferente de zero a partir da esquerda de uma linha não nula é um 1. Este elemento é chamado de um inicial desta linha.*
3. *Para cada linha diferente de zero, o um inicial aparece à direita e abaixo dos uns iniciais das linhas precedentes.*
4. *Se uma coluna contém o um inicial, então todos os outros elementos naquela coluna são iguais a zero.*

Em uma matriz na forma escalonada reduzida por linhas, os primeiros coeficientes das linhas não nulas formam uma escada. Uma matriz $m \times n$ que satisfaz as propriedades (1), (2) e (3) está na *forma escalonada por linhas.*

***Definição* B.11** *Uma operação elementar nas linhas de uma matriz $A = \left(a_{ij}\right)$ $m \times n$ é uma das seguintes operações*

1. *Permuta das linhas r e s de A. Ou seja, substituir $a_{r1}, a_{r2}, \ldots, a_{rn}$ por*

   $a_{s1}, a_{s2}, \ldots, a_{sn}$ *e* $a_{s1}, a_{s2}, \ldots, a_{sn}$ *por* $a_{r1}, a_{r2}, \ldots, a_{rn}$.

2. *Multiplicação da r-ésima linha de A por* $c \neq 0$. *Ou seja,* $a_{r1}, a_{r2}, \ldots a_{rn}$ *por* $ca_{r1}, ca_{r2}, \ldots, ca_{rn}$

3. *Adição de d vezes a r-ésima linha de A à s de A,*$r \neq s$. *Ou seja, substituir* $a_{s1}, a_{s2}, \ldots, a_{sn}$ *por* $a_{s1} + da_{r1}, a_{s2} + da_{r2}, \ldots, a_{sn} + da_{rn}$

***Definição* B.12** *Uma matriz An* $\times$ *n é chamada triangular superior se todos os elementos por baixo da diagonal principal são zero. Assim, a matriz ilustrada na equação B.8 é uma matriz triangular superior.*

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22} & a_{23} \\ 0 & 0 & a_{33} \end{pmatrix} \tag{B.8}$$

***Definição* B.13** *Uma matriz A é estritamente triangular superior se* $i \geq j \Rightarrow A\left(i, j\right) = 0$, *ou seja, a matriz é necessariamente nula abaixo da primeira sobrediagonal.*

***Definição* B.14** *Uma matriz An* $\times$ *n é chamada triangular inferior se todos os elementos acima da diagonal principal são zero. Assim, a matriz mostrada na equação B.9 é uma matriz triangular inferior.*

$$\begin{pmatrix} a_{11} & 0 & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \tag{B.9}$$

***Definição* B.15** *Uma matriz A é estritamente triangular inferior se* $i \leq j \Rightarrow A\left(i, j\right) = 0$, *ou seja, a matriz é necessariamente nula acima da primeira subdiagonal.*

***Definição* B.16** *Uma matriz An*$\times$*n é chamada invertível (ou não-singular ou regular) se existir uma matriz Bn* $\times$ *n tal que* $AB = BA = I_n$. *A matriz B é chamada inversa de A. Se essa matriz B não existe, então A é chamada de singular (ou não-invertível).*

***Definição* B.17** *Seja* $A = \left(a_{ij}\right)$ *uma matriz* $n \times n$*. Definimos o determinante de A (escreve-se* $det(A)$ *ou* $|A|$*) pela expressão B.10,*

$$det(A) = |A| = \sum_{1}^{n} (\pm) a_{1,j1} a_{1,j2} \cdots a_{n,jn} \tag{B.10}$$

onde o somatório varia por todas as permutações $j1, j2, \cdots jn$ *do conjunto* $S = \{1, 2, \cdots, n\}$*. O sinal é positivo* $(+)$ *ou negativo* $(-)$ *conforme a permutação* $j1, j2, \cdots jn$ *seja par ou ímpar.*

***Definição* B.18** *Seja* $A = \left(a_{ij}\right)$ *uma matriz* $n \times n$*. Seja* M_{ij} *a submatriz* $(n-1) \times (n-1)$ *de A obtida pela eliminação da i-ésima linha e da j-ésima coluna de A. O determinante* $det(M_{ij})$ *é chamado determinante menor de* a_{ij}*. O co-fator* A_{ij} *de* a_{ij} *é definido tal como mostrado na equação B.11.*

$$A_{ij} = (-1)^{i+j} det(M_{ij}) \tag{B.11}$$

***Definição* B.19** *Seja* $A = \left(a_{ij}\right)$ *uma matriz* $n \times n$*. A matriz adjunta de* $An \times n$*, representada por adj A, é a matriz cujo i,j-ésimo elemento é o co-fator de* A_{ij} *de* a_{ij}*. Assim, a matriz ilustrada na equação B.12, é a matriz adjunta de A.*

$$adj\ A = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{n1} \\ A_{21} & A_{22} & \cdots & A_{n2} \\ \vdots & \vdots & & \vdots \\ A_{1n} & A_{2n} & \cdots & A_{nn} \end{pmatrix} \tag{B.12}$$

***Definição* B.20** *Seja* $A \in C^{m \times p}$*, a transposta de A denotada por* A^T *tal que* $A^T \in C^{p \times m}$ *e* $A^T\left(i, j\right) = A\left(j, i\right)$ *para* $1 \leq i \leq p$ *e* $1 \leq j \leq m$*, ou seja, trocamos as linhas pelas colunas de A.*

***Definição* B.21** *Seja $A \in C^{m\times p}$, a Hermitiana de A denotada por A^H tal que $A^H \in C^{p\times m}$ e $A^H\left(i,j\right) = A\left(\bar{j},i\right)$ para $1 \leq i \leq p$ e $1 \leq j \leq m$, ou seja, trocamos as linhas pelas colunas de A e substituímos cada entrada complexa de A pelo seu conjugado. Usa-se também a notação A^*. Se A é real então $A^T = A^H$ (ver equação B.13).*

$$A = \begin{pmatrix} 3 & 2+i & -i \\ 2-i & 5 & 5-i \\ i & 5+i & -1 \end{pmatrix} \tag{B.13}$$

***Definição* B.22** *Se $A^H = A$, então A é Hermitiana.*

***Definição* B.23** *Se $A^T = A$, então A é simétrica.*

***Definição* B.24** *Se $x \neq 0 \Rightarrow x^H Ax > 0$, então A é positiva e definida.*

***Definição* B.25** *Se $x^H Ax \geq 0$ para todo $x \in C^m$, então A é positiva e semidefinida.*

***Definição* B.26** *Se $A^H A = AA^H = I$ onde I é a matriz identidade de ordem m, então A é unitária.*

***Definição* B.27** *Se $A^T A = AA^T = I$, então A é ortogonal.*

***Definição* B.28** *Se $A^H A = AA^H$, então A é normal.*

***Definição* B.29** *A matriz A é Hessenberg superior se $i > j+1 \Rightarrow A\left(i,j\right) = 0$, ou seja, todas as entradas abaixo da primeira subdiagonal são nulas (ver equação B.14).*

$$\begin{pmatrix} A(1,1) & A(1,2) & \cdots & \cdots & \cdots & A(1,m) \\ A(2,1) & A(2,2) & \cdots & \cdots & \cdots & A(2,m) \\ 0 & A(3,2) & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & \ddots & \ddots & \vdots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & A(m,m-1) & A(m,m) \end{pmatrix} \tag{B.14}$$

***Definição* B.30** *A matriz A é Hessenberg inferior se* $i < j+1 \Rightarrow A\left(i,j\right) = 0$*, ou seja, todas as entradas acima da primeira sobrediagonal são nulas (ver equação B.15).*

$$\begin{pmatrix} A(1,1) & A(1,2) & 0 & \cdots & \cdots & 0 \\ A(2,1) & A(2,2) & A(2,3) & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \ddots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & A(m-1,m) \\ A(m,1) & A(m,2) & \cdots & \cdots & \cdots & A(m,m) \end{pmatrix} \tag{B.15}$$

***Definição* B.31** *A matriz A é tridiagonal se A é Hessenberg superior e A é Hessenberg inferior, concomitantemente (ver equação B.16).*

$$\begin{pmatrix} A(1,1) & A(1,2) & 0 & \cdots & \cdots & 0 \\ A(2,1) & A(2,2) & A(2,3) & 0 & \cdots & 0 \\ 0 & A(3,2) & \ddots & \ddots & \ddots & \vdots \\ \vdots & 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & A(m-1,m) \\ 0 & 0 & \cdots & 0 & A(m,m-1) & A(m,m) \end{pmatrix} \tag{B.16}$$

***Definição* B.32** *Um escalar* λ *é autovalor de um operador linear* $A : V \Longrightarrow V$ *se existir um vector x diferente de zero tal que* $Ax = \lambda x$*. O vector x é chamado autovetor.*

B.2 Definições de Espaços Relevantes

Seja $A \in C^{n \times p}$, então:

***Definição* B.33** *O núcleo, ou espaço nulo de A, será denotado como* $nuc\left(A\right)$ *para todo* $\{x \in C^p; Ax = 0\}$.

Definição* B.34** *A imagem de A será denotada como* $im\left(A\right)$ *para todo* $\{y \in C^n;$ *existe* $x \in C^p; Ax = y\}$. *Ou seja, é o espaço que contém todos os vetores gerados pela aplicação de A nos vetores do domínio. Como esse espaço é formado por combinações lineares de colunas de A, utilizaremos, também, o nome* ***espaço coluna*.*

Definição* B.35** *A imagem de* A^T *será denominada* ***espaço linha *de A, pois é formada por combinações lineares das linhas de A. A imagem de* A^H *será denominada* ***espaço conjugado linha*** *de A, pois é formada por combinações lineares dos conjugados das linhas de A.*

Definição* B.36** *O* ***posto coluna *é a dimensão do espaço coluna.*

Definição* B.37** *O* ***posto linha *é a dimensão do espaço linha ou do espaço conjugado linha.*

Definição* B.38** *O* ***complemento ortogonal *de* $\mathbf{W}$, *subespaço de* C^n, *é formado por todos os vetores ortogonais a* $\mathbf{W}$, *e é denotado por* $W^{\perp}$.

B.3 O conceito de Base

Uma base, de um espaço vetorial C^n, é um conjunto linearmente independente de n vetores $\{v_1, v_2, ..., v_n\}$ que geram esse espaço C^n. Seja $V \in C^{n \times n}$ uma matriz cujas colunas são os vetores v_i, $i = 1 : n$, então essa base será ortonormal se e somente se $V^H V = I_n$. A representação de um vetor $v \in C^n$ dessa base será dada pela equação B.17.

$$v = \sum_{i=1}^{n} v^H v_i v_i \tag{B.17}$$

Caso o conjunto $\{v_1, v_2, ..., v_n\}$ seja uma base para C^n, mas não seja um conjunto de vetores ortonormais, então um vetor qualquer $v \in C^n$ pode ter a forma mostrada pela equação B.18 com α_i tomando valores dado pela equação B.19.

$$v = \sum_{i=1}^{n} \alpha_i v_i \tag{B.18}$$

$$\alpha_i = v^H u_i,\ i = 1 : n, \tag{B.19}$$

Assim, $\{u_1, u_2, ..., u_n\}$ é uma outra base para C^n tal que $U^H V = I_n$, cujas colunas de U são os vetores u_i, $i = 1 : n$. A base $\{u_1, u_2, ..., u_n\}$, definida acima, é denominada de **base adjunta** de $\{v_1, v_2, ..., v_n\}$. Os $2n$ vetores u_i e v_i, $i = 1 : n$, formam uma **coleção biortogonal** de elementos de C^n.

B.4 Demostração do Teorema de Schur

***Demostração* B.1** *(Teorema de Schur). Por indução assuma-se que o resultado é verdadeiro para matrizes de ordem $n-1$ sendo A de ordem n. Seja v um autovetor com valores λ. Normalizar v, ou seja, substituir v por $v/\|v\|$ se necessario. Estender v em uma base ortonormal v, w_1, ..., w_{n-1} usando Gram-Schmidt. Seja Q a matriz de transição dessa base. Assim Q é unitaria e $Q^{-1}AQ$ é da forma*

$$\begin{pmatrix} \lambda & * & \cdots & * \\ 0 & & & \\ \vdots & & C & \\ 0 & & & \end{pmatrix}.$$

Por indução existe um V unitario com $V^{-1}CV$ sendo triangular superior. Considere

$$U = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & & & \\ \vdots & & V & \\ 0 & & & \end{pmatrix}$$

a qual é unitaria também. Então QU é unitaria e $(QU)^{-1} A (QU) = U^{-1} \left(Q^{-1}AQ\right) U =$

$$= \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & & & \\ \vdots & & V^{-1} & \\ 0 & & & \end{pmatrix} \begin{pmatrix} \lambda & * & \cdots & * \\ 0 & & & \\ \vdots & & C & \\ 0 & & & \end{pmatrix} \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & & & \\ \vdots & & V & \\ 0 & & & \end{pmatrix} = \begin{pmatrix} \lambda & * & \cdots & * \\ 0 & & & \\ \vdots & & V^{-1}CV & \\ 0 & & & \end{pmatrix}$$

a qual é triangular superior.