IMPLEMENTAÇÃO EM FPGA DE UMA BIBLIOTECA PARAMETRIZÁVEL PARA INVERSÃO DE MATRIZES BASEADA NO ALGORITMO GAUSS-JORDAN, USANDO REPRESENTAÇÃO EM PONTO FLUTUANTE

JANIER ARIAS GARCÍA

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA MECÂNICA

IMPLEMENTAÇÃO EM FPGA DE UMA BIBLIOTECA PARAMETRIZÁVEL PARA INVERSÃO DE MATRIZES BASEADA NO ALGORITMO GAUSS-JORDAN, USANDO REPRESENTAÇÃO EM PONTO FLUTUANTE

JANIER ARIAS GARCÍA

Orientador: Prof. Dr. Ricardo Pezzuol Jacobi

DISSERTAÇÃO DE MESTRADO

Publicação: ENM.DM-36A/10

Brasília, 24 de Setembro de 2010

UNIVERSIDADE DE BRASÍLIA
Faculdade de Tecnologia

DISSERTAÇÃO DE MESTRADO

IMPLEMENTAÇÃO EM FPGA DE UMA BIBLIOTECA PARAMETRIZÁVEL PARA INVERSÃO DE MATRIZES BASEADA NO ALGORITMO GAUSS-JORDAN, USANDO REPRESENTAÇÃO EM PONTO FLUTUANTE

JANIER ARIAS GARCÍA

Dissertação de Mestrado submetida ao Departamento de Engenharia Mecânica da faculdade de Tecnologia da Universidade de Brasília como requisito parcial para a obtenção do grau de Mestre em Sistemas Mecatrônicos

Banca Examinadora

Prof. Dr. Ricardo Pezzuol Jacobi, CIC/UnB ______________________________
Orientador

Prof. Dr. Carlos H. Llanos Quintero, ENM/UnB ______________________________
Co-orientador

Prof. Dr. Carla Maria Chagas e Cavalcante Koike, ENM-CIC/UnB ______________________________
Examinador interno

Prof. Dr. Pedro de Azevedo Berger, CIC/UnB ______________________________
Examinador externo

Brasília, 24 de Setembro de 2010

FICHA CATALOGRÁFICA

ARIAS GARCIA., JANIER
IMPLEMENTAÇÃO EM FPGA DE UMA BIBLIOTECA PARAMETRIZÁVEL PARA INVERSÃO DE MATRIZES BASEADA NO ALGORITMO GAUSS-JORDAN, USANDO REPRESENTAÇÃO EM PONTO FLUTUANTE [Distrito Federal] 2010.
xiv, 71p. 210 × 297 mm (ENM/FT/UnB, Mestre, Sistemas Mecatrônicos, 2010). Dissertação de Mestrado – Universidade de Brasília. Faculdade de Tecnologia.
Departamento de Engenharia Mecânica.

1. Aritmética de Ponto flutuante
2. FPGAs
3. Eliminação de Gauss
4. Redução de Gauss-Jordan

I. ENM/FT/UnB
II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

ARIAS GARCIA, JANIER. (2010). Implementação em FPGA de uma biblioteca parametrizável para inversão de matrizes baseada no algoritmo Gauss-Jordan, usando representação em ponto flutuante. Dissertação de Mestrado em Sistemas Mecatrônicos, Publicação ENM.DM-36A/10, Departamento de Engenharia Mecânica, Universidade de Brasília, Brasília, DF, 71p.

CESSÃO DE DIREITOS

AUTOR: Janier Arias Garcia.

TÍTULO: Implementação em FPGA de uma biblioteca parametrizável para inversão de matrizes baseada no algoritmo Gauss-Jordan, usando representação em ponto flutuante

GRAU: Mestre **ANO**: 2010

É concedida à Universidade de Brasília permissão para reproduzir cópias desta dissertação de mestrado e para emprestar ou vender tais cópias somente para propósitos acadêmicos e científicos. O autor reserva outros direitos de publicação e nenhuma parte dessa dissertação de mestrado pode ser reproduzida sem autorização por escrito do autor.

Janier Arias Garcia

Agradecimentos

Agradeço primeiramente a minha família pelo apoio constante durante esta etapa da minha vida.
Ao professor Ricardo Pezzuol Jacobi, meu orientador, por ter-me brindado a possibilidade de trabalhar no seu lado.
Ao professor Carlos Humberto Llanos pela dedicação, pelos conhecimentos transmitidos, e mais ainda, pela confiança, paciência e amizade brindadas no desenvolvimento deste trabalho.
Aos meus colegas e amigos do GRACO, especialmente ao Daniel Muñoz, Jones Yudi, Diego Sanchez, Ronald Hurtado, André Braga, Alvaro Patiño, Jesus Pinto e Gloria Lopéz pela ajuda, conselhos, também pelas alegrias compartidas e pela companhia nos momentos difíceis.
À CNPQ (Conselho Nacional de Desenvolvimento Científico e Tecnológico) pelo apoio financeiro deste trabalho.
Ao Grupo de automação e Controle (GRACO) e todos meus professores pelo suporte e formação acadêmica.
Agradeço especialmente a Diana Paola Goméz Mendoza pelo desvelo, compreensão e amor oferecidos durante este tempo.

JANIER ARIAS GARCÍA

RESUMO

As operações computacionais em que se desenvolvem cálculos matriciais são à base, ou melhor, o coração de muitos algoritmos computacionais científicos, por exemplo: processamento de sinais, visão computacional, robótica, entre outros. Esse tipo de algoritmos em que desenvolvem-se cálculos matriciais terminam sendo tarefas computacionalmente custosas, e suas implementações em hardware exigem grandes esforços e tempo. Existe então uma crescente demanda por arquiteturas que permitam cálculos matriciais, proporcionando soluções rápidas e eficientes para este tipo de problema.

Este trabalho apresenta diferentes arquiteturas computacionais para inverter matrizes em hardware reconfigurável, FPGA: (a) *sequencial*, (b) *pipeline* e (c) *Paralelo*. Estas arquiteturas usam uma representação de ponto flutuante tanto em precisão simples (32 bits) quanto precisão dupla (64 bits), visando o uso em implementações de baixo consumo de recursos lógicos, na qual a unidade principal é o componente de processamento para redução Gauss-Jordan. Esse componente consiste de outras pequenas unidades organizadas de tal forma que mantêm a precisão dos resultados sem a necessidade de internamente normalizar e de-normalizar os dados em ponto flutuante. No intuito de gerar arquiteturas de baixo custo, este trabalho propõe o estudo de diferentes formas de abordar o problema, descrevendo em código VHDL estas arquiteturas em que os tamanhos de matrizes são definidos pelos usuários. Os resultados de erro e de tempo de execução das arquiteturas desenvolvidas foram comparados contra o MatLab, que faz uma simulação comportamental do código VHDL gerado através do ambiente de simulação *ModelSim*.

A implementação das operações e da própria unidade procura explorar os recursos disponíveis na FPGA Virtex-5. O desempenho e o consumo de recursos são apresentados, comparando as diferentes arquiteturas desenvolvidas entre si e entre outras arquiteturas propostas encontradas em publicações anteriores. Além disso, é mostrado o decremento no desempenho a medida que o tamanho da matriz aumenta.

ABSTRACT

Computer operations demanding matrix calculations are at the heart of many scientific computing algorithms such as: signal processing, computer vision, robotics, among others. Because these algorithms perform matrix calculations, they are often computationally expensive, and their hardware implementations require much effort and time. So there is a growing demand for architectures that perform matrix calculations, fast and efficiently.

This work presents different computer architectures for matrix inversion in FPGA reconfigurable hardware: (a) sequential, (b) pipeline and (c) Parallel. These architectures use a floating point representation in both single-precision (32 bit) and double precision (64 bits), suitable for use in low cost implementations, and where main component is Gauss-Jordan reduction. This component consists of other small units arranged in such a way that maintains the accuracy of results without the need of internally normalizing and de-normalizing the floating point data. In order to generate low-cost architectures, this work proposes to study different ways of approaching the problem in VHDL code, and allowing that sizes of matrices be defined by users. All architectures were simulated using MatLab, with a behavioral simulation of VHDL code generated by ModelSim simulation environment. As a result of comparing the error obtained by the architecture, with the inversion performed using MatLab as static estimator.

The implementation of operations and the unit seeks to explore the resources available in Virtex-5 FPGA. The performance and resource consumption are presented, comparing the different architectures developed between themselves and with others proposed in previous publications. In addition, it is shown the influence of the array size in the performance.

Sumário

Lista de Figuras .... **xii**

Lista de Tabelas .... **xiii**

1 Introdução **1**

1.1 Contextualização 1

1.2 Definição do problema e motivações 4

1.3 Objetivos 5

1.3.1 Objetivo Geral 5

1.3.2 Objetivos Específicos 5

1.4 Estrutura do Texto 5

2 Plataformas Paralelas para Algoritmos de Cálculo Matricial **7**

2.1 Arquiteturas von Neumann 8

2.2 Arranjos Massivos de Processadores em Paralelo (MPPAs) 8

2.3 Unidades de Processamento Gráfico (GPUs) 9

2.4 *Field Programmable Gate Arrays (FPGAs)* 10

2.4.1 Descrição geral da família Virtex-5 13

2.4.2 Arquitetura da Virtex-5 13

2.4.3 Conclusão do capítulo 14

3 Decomposição, Redução e Inversão de Matrizes **16**

3.1 Trabalhos correlatos à implementação de inversões de matrizes em *hardware* 16

3.2 Definições Gerais 17

3.3 Decomposição de Matrizes 22

3.3.1 Decomposição LU 22

3.3.2 Decomposição Cholesky 22

3.3.3 Decomposição QR 23

3.4 Redução de Gauss-Jordan ... 23
3.5 Os Métodos de Inversão de Matrizes ... 24
3.5.1 Inversão de Matriz baseada na decomposição LU ... 24
3.5.2 Inversão de Matriz baseada na decomposição Cholesky ... 25
3.5.3 Inversão de Matriz baseada na decomposição QR ... 25
3.5.4 Inversão de Matriz baseada na Redução de Gauss-Jordan ... 26
3.5.5 Inversão de Matriz usando método analítico ... 27
3.5.6 Complexidade dos algoritmos de inversão de matrizes ... 27
3.6 Conclusão do Capítulo ... 27

4 Implementação ... 30
4.1 Bibliotecas de ponto flutuante ... 30
4.1.1 Unidade de Soma/Subtração ... 30
4.1.2 Unidade de Multiplicação ... 32
4.1.3 Unidade de Divisão ... 33
4.2 Implementação do Algoritmo de Gauss-Jordan ... 35
4.2.1 O Algoritmo Gauss-Jordan ... 35
4.2.2 Arquitetura reconfigurável da inversão da matriz ... 37
4.2.3 O fluxo de dados desde/para o bloco Inver_GJ ... 40
4.2.4 Unidade Trocador de linhas ... 43
4.2.5 Sobre a memoria RAM ... 45
4.2.6 O *pipeline* ... 50
4.2.7 O paralelismo ... 50
4.3 Etapas no desenvolvimento do Projeto ... 52
4.4 Conclusão do Capítulo ... 52

5 Resultados ... 53
5.1 Algoritmos GJ-Pivô_A e GJ-Pivô_B ... 53
5.2 A unidade de Change_Row ... 58
5.3 Arquiteturas de inversão de matrizes ... 60
5.4 Conclusão do Capítulo ... 63

6 Conclusões ... 66
6.1 Comentários finais e conclusões ... 66
6.2 Propostas de futuros trabalhos ... 67
6.2.1 GECO ... 67
6.2.2 Explorar as RAM internas ... 67
6.2.3 Paralelismo em um pipeline de operações ... 67

6.2.4 Métodos e correção do erro 68

REFERÊNCIAS BIBLIOGRÁFICAS 69

Lista de Figuras

2.1 Aquitetura CPU Multi-nucleo ... 9

2.2 Arquitetura da FPGA e seus recursos: I/O de células, blocos lógicos (CLBs) e interconexões ... 11

2.3 Etapas de um projeto com FPGAs (modificado de [Aragão 1998]) ... 12

3.1 Decomposição LU (obtido de [Irturk 2009]) ... 22

3.2 Decomposição Cholesky (obtido de [Irturk 2009]) ... 23

3.3 Decomposição QR (obtido de [Irturk 2009]) ... 23

3.4 As etapas da solução da inversão de matrizes utilizando decomposição LU (modificado de [Irturk 2009]) ... 25

3.5 As etapas da solução da inversão de matrizes utilizando decomposição Cholesky (modificado de [Irturk 2009]) ... 25

3.6 As etapas da solução da inversão de matrizes utilizando decomposição QR (modificado de [Irturk 2009]) ... 26

3.7 Número total de operações no domínio logarítmico para a inversão de matrizes baseada nos métodos de decomposição (modificado de [Irturk 2009]) ... 28

3.8 Número total de operações no domínio logarítmico para a inversão de matrizes baseada na redução de Gauss-Jordan ... 28

4.1 Algoritmo seguido na implementação da unidade de Soma/Subtração (Obtido de [Sanchez 2009]) ... 31

4.2 entradas/saidas para o bloco soma/subtração ... 31

4.3 Algoritmo seguido na implementação da unidade de Multiplicação (Obtido de [Sanchez 2009]) ... 32

4.4 Entradas/Saidas para o bloco Multiplicação ... 33

4.5 Código da Unidade de Multiplicação em ponto flutuante (Obtido de [Sanchez 2009]) 33

4.6 Algoritmo seguido na implementação da unidade de Divisão (Modificado de [Sanchez 2009]) ... 34

4.7 Implementação do algoritmo Newton - Raphson para divisão (Obtido de [Sanchez 2009]) ... 35

4.8 Estrutura das arquiteturas propostas. A seção dentro do quadro na arquitetura Inv_GJ-B foi modificado de [Duarte Horácio Neto 2009] 38
4.9 Estrutura sequencial da unidade de eliminação matricial (modificado de [Duarte Horácio Neto 2009]) 40
4.10 Estrutura sequencial da unidade de normalização (modificado de [Duarte Horácio Neto 2009]) 41
4.11 Diagrama de bloco do módulo de inversão GJ para a matriz A 42
4.12 Diagrama de bloco do módulo de inversão GJ para a matriz I 42
4.13 Controle interno no bloco Inver_GJ 42
4.14 Registro pos_memoria 43
4.15 Código em VHDL da Unidade Trocador de linhas 45
4.16 FSM da unidade Trocador de linhas 46
4.17 Memória True Dual-Port RAM (Obtido de [LogiCORE IP Block Memory Generator v4.2 2010]) 46
4.18 Memória Simple Dual-Port RAM (Obtido de [LogiCORE IP Block Memory Generator v4.2 2010]) 47
4.19 Transformação dos endereços matriciais em endereços vetoriais 47
4.20 Transformação dos endereços matriciais em endereços vetoriais concatenados 48
4.21 Código em VHDL do Controlador da memória RAM_sequencial 48
4.22 Código em VHDL do Controlador da memória RAM_combinacional 49
4.23 FSM do controlador da memória RAM 50
4.24 *Pipeline* implementado em *hardware* 51
4.25 Implementação em *hardware* dos multiplicadores e somadores em paralelo 51
4.26 Caminho seguido para obter a melhor solução em *hardware* para inverter matrizes 52

5.1 Erro médio para o algorítimo GJ-Pivô_A usando precisão simples 54
5.2 Erro quadrático médio para o algorítimo GJ-Pivô_A usando precisão simples 55
5.3 Erro médio para o algorítimo GJ-Pivô_B usando precisão simples 55
5.4 Erro quadrático médio para o algorítimo GJ-Pivô_B usando precisão simples 56
5.5 Erro médio para o algorítimo GJ-Pivô_A usando precisão dupla 56
5.6 Erro quadrático médio para o algorítimo GJ-Pivô_A usando precisão dupla 57
5.7 Erro médio para o algorítimo GJ-Pivô_B usando precisão dupla 57
5.8 Erro quadrático médio para o algorítimo GJ-Pivô_B usando precisão dupla 58
5.9 Número de comparações mínimo para encontrar o Pivô 59
5.10 Desempenho da unidade Trocador de linhas-01 59
5.11 Desempenho da unidade Trocador de linhas-02 60
5.12 Desempenho da unidade Trocador de linhas-03 60

5.13 Desempenho da Arquitetura Sequencial 61
5.14 Desempenho da Arquitetura Pipeline 62
5.15 Desempenho da arquitetura Paralelo 63
5.16 Comportamento da frequência á medida que o tamanho da matriz aumenta 64

Lista de Tabelas

2.1 Comparação entre GPU, MPPA e FPGA 15

4.1 MSE da unidade Soma/Subtração para diferentes larguras de bits (Obtido de [Sanchez 2009]) 32
4.2 MSE da unidade multiplicação para diferentes larguras de bits (Obtido de [Sanchez 2009]) 33

5.1 Resultados da Sínteses para N=4 na arquitetura sequencial 62
5.2 Resultados da Sínteses para N=36 na arquitetura sequencial 63
5.3 Resultados da Sínteses para N=4 na arquitetura Paralelo 64
5.4 Resultados da Sínteses para N=36 na arquitetura Paralelo 64

LISTA DE SÍMBOLOS

ABNT	Associação Brasileira de Normas Técnicas
ALU	Unidade Aritmética e Lógica
ASIC	Application Specific Integrated Circuits
CAD	Computer Aided Design
CLB	Configuration Logic Block
CMOS	Complementary Metal Oxide Semiconductor
CMP	Multi-core Processors
CPU	Unidade Central de Processamento
CU	Unidade de Controle
DSP	Digital Signal Processor
GJ	Gauss-Jordan
GPGPU	General Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
ISE	Integrated Software Environment
LTU	LookUp Table
N	Número Elementos
ME	Mean Error
MSE	Mean Square Error
MIMD	Multiple Instruction Streams, Multiple Data
MPPA	Arranjos Massivos de Processadores em Paralelo
QR-GR	QR Givens rotations
QR-MGS	QR Modified Gram-Schmidt
RTL	Register Transfer Level
VHDL	VHSIC-HDL Very-High Scale of Integration Circuit - Hardware Description Language
VLSI	Very Large Scale of Integration

1 Introdução

1.1 Contextualização

Na resolução de um problema em engenharia, ou em outras ciências, é geralmente possível distinguir três fases: (1) a formulação matemática do problema, (2) a escolha de um método ou combinações de métodos analíticos e numéricos para resolver o problema formulado, e (3) o cálculo numérico da solução. Basicamente um método numérico é um conjunto ordenado de operações aritméticas e lógicas, fundamentado em teoremas da análise matemática, que conduz à solução de um problema matemático. A um método numérico está pois associado um algoritmo. Um método numérico deve ser acompanhado de um estudo sobre majorações de erros, convergência, escolha do passo, estabilidade, etc.; e para cada problema deve-se saber algo acerca do seu bom ou mau condicionamento [CARPENTIER 1993].

Para escolher dentre vários métodos numéricos o mais indicado à resolução de um determinado problema devemos saber como esses se deduzem e por conseguinte os seus domínios de aplicação e suas limitações; como já referimos devemos fazer as respectivas análises de erros e devemos, em certos casos, estimar o número de operações a efetuar em cada método. Uma vez escolhido um método numérico devemos construir um algoritmo associado a este. Este algoritmo deverá conter os principais passos do método e permitir ao programador traduzi-lo em um programa inteligível para o computador. Na hipótese de vários métodos numéricos poderem conduzir à resolução do mesmo problema com a precisão exigida pode não ser indiferente a escolha de um deles. Critérios computacionais tais como maior rapidez de execução, menor ocupação da memória e menor complexidade computacional podem ser decisivos na escolha [CARPENTIER 1993].

Um problema diz-se bem condicionado se pequenos erros nos dados produzem pequenos erros no resultado ou de forma equivalente, se uma pequena mudança nos dados produz uma pequena mudança no resultado. Caso contrário o problema é mal condicionado [Trefethen 1997]. Assim, as matrizes são elementos matemáticos mal condicionados, devido a que pequenas perturbações (mudanças) nos parâmetros da matriz influenciam nas operações com a mesma, como no caso da inversão de matrizes. O significado de pequenas ou de grandes mudanças depende das aplicações. Isto gera uma lista de parâmetros e demandas para fazer na hora da escolha do método

matemático para inverter matrizes, onde tem-se, entre as demandas, a escolha da plataforma adequada para a implementação em *hardware*.

Os FPGAs são dispositivos de *hardware* reconfigurável usados como uma plataforma comum para resolver diferentes problemas computacionais que envolvem cálculos algébricos intensivos, por exemplo: comunicações sem fio, reconhecimento de voz, processamento de imagens, robótica e sistemas oscilatórios, já que pode ser configurado especificamente para um algoritmo, aproveitando o paralelismo intrínseco do mesmo. Frequentemente, a maioria dessas aplicações envolve algoritmos que devem tratar com estruturas matriciais e as suas respectivas operações, tornando esse tipo de aplicações computacionalmente custosas. Uma das operações mais importantes e de alto custo computacional é a inversão de matrizes, na qual muitas aproximações numéricas são aplicadas, na maioria em software. Embora, a complexidade computacional da inversão de matrizes seja uma pergunta aberta, soluções populares sub-cúbicas em software tais como de $O(n^{2.807})$ Strassen e de $O(n^{2.376})$ Coppersmith-Winograd baseadas em aproximações, aplicada a inversões de matrizes $n \times n$, são de grande interesse teórico, mas, não são usadas na prática porque só são viáveis para matrizes de tamanhos grandes, o que faz que a sua implementação em *hardware* seja inadequada (e.g. Capitulo 12 de [Gathen e Gerhard 2003]). Algoritmicamente, métodos simples como Redução de Gauss-Jordan (GJ), embora tenham uma alta complexidade ($O(n^3)$), são muito importantes para desenvolver implementações com arquiteturas práticas, visto que permitem um alto grau de paralelismo.

A implementação em *hardware* desses algoritmos é muito interessante devido ao fato de eliminar os problemas que surgem em uma arquitetura von Neumann, principalmente com relação às instruções de escrita/leitura desde/para a memória RAM. No caso de operações matriciais que são implementadas em *hardware*, somente operações de escrita/leitura precisam ser executadas sem a complexidade dos passos de decodificação/execução relacionados com as execuções de instruções, já que são fortemente relacionadas ao modelo de von Neumann. Os FPGAs são plataformas adequadas para estes cálculos aritméticos computacionalmente intensos pois provêem poderosas características de arquiteturas computacionais: (a) grande quantidade de elementos lógicos programáveis (LUTs), (b) blocos de memória RAM (BRAMs), (c) multiplicadores embutidos, (d) *Shift register* (SRLs), Blocos DSP e (e) administradores de relógio digital (DCMs) [Irturkt Bridget Bensont e Kastnert 2008].

Métodos de decomposição ou redução precisam ser introduzidos para inversões de matrizes de tamanho grande devido a que aproximações analíticas resultam em arquiteturas difíceis de extender, devido ao uso de determinantes, impedindo o reaproveitamento dessas arquiteturas para a implementação de inversões de matrizes de diferentes tamanhos.

Alguns métodos como Redução de GJ [Duarte Horácio Neto 2009] e decomposição QR (QRD) [C.K. PrasadS.H. 2007, M.Karkooti J.R.Cavallaro 2005] são os métodos de decomposição tradi-

cionalmente usados, além dos métodos LU e Cholesky. O método de Cholesky é usado geralmente para matrizes quadradas definidas positivas (matriz Hermitiana) e não singulares, enquanto que a Redução de GJ e o QRD são usados para qualquer tipo de matrizes [A.Irturk e R.Kastner 2008].

Por causa do interesse em usar as vantagens do *hardware* reconfigurável para aplicações de baixo consumo de recursos lógicos, optou-se pelo algoritmo de redução de GJ. Dentre as vantagens da Redução de GJ está o fato do mesmo ser um método direto, alem de exigir apenas três operações matemáticas (especificamente, soma/subtração, multiplicação e divisão). Por outro lado, o QRD usa pelo menos uma operação a mais. Especificamente, o método QRD-Gran-Schmidt-Orthogonormalization [C.K. PrasadS.H. 2007] usa também raiz quadrada, enquanto que o QRD-Givens-Rotations usa operações de seno e cosseno.

Algumas arquiteturas propostas na literatura usam soluções computacionais para acelerar a inversão, por exemplo, estruturas com arranjos sistólicos, que gastam uma grande quantidade de área para alcançar alguma vantagem em desempenho (velocidade). Porém, em aplicações de baixo custo a área tem uma função importante, contanto que isso não penalize a precisão [Edman e Öwall 2005, Bigdeli Morteza Biglari-Abhari e Lai 2006].

A utilização de uma representação aritmética em ponto flutuante (FP) tende a piorar a situação, uma vez que as bibliotecas de FP normalmente usam uma grande quantidade dos recursos do FPGA [Sánchez D. Muñoz e Ayala-Rincón 2009]. A implementação otimizada de operações aritméticas em ponto flutuante em FPGAs têm uma importância relevante em uma variedade de aplicações científicas, devido ao grande alcance dinâmico necessário para a representação de números reais, o que permite um melhor desempenho dos algoritmos envolvidos, quando comparados com a sua representação de dados em ponto fixo.

Implementações em ponto flutuante são baseadas no padrão IEEE-754 para representação de números em formato de sequências de bits, e são caracterizadas por 3 componentes: (a) o sinal S, (b) o bias do expoente E com Ew sendo o bit de largura e (c) a mantissa M com Mw sendo o bit de largura. Um valor zero para S denota um número positivo, e um valor um denota um número negativo. A constante ($Bias$) é adicionada ao expoente para fazer o intervalo do expoente não-negativo, enquanto que a mantissa representa a magnitude do número. Este padrão permite ao usuário trabalhar com uma precisão-simples de 32-bits assim como uma precisão dupla de 64-bits. Adicionalmente, o mesmo padrão prevê trabalhar uma precisão apropriada (escolhida a priori pelo usuário) de acordo com a aplicação.

A utilização do FPGA oferece a flexibilidade necessária ao desenvolvimento do sistema proposto, sendo uma plataforma já bastante estudada para aplicações de alto desempenho. Um ponto importante é que os FPGAs podem implementar de forma natural o paralelismo inerente aos algoritmos utilizados para inversões de matrizes, sendo possível a implementação de arquite-

turas com estruturas *pipeline*. Um ASIC poderia ser utilizado, porém o longo tempo de desenvolvimento e o elevado custo só viabilizariam projetos de larga escala de produção. Adicionalmente, os FPGAs oferecem ainda outras vantagens em relação aos microprocessadores genéricos e aos DSPs (Digital Signal Processors), tendo em conta a sua flexibilidade, em aplicações de alto desempenho e pequeno volume, e ainda mais particularmente em aplicações que possam explorar larguras de bits específicas e com alto grau de paralelismo de instruções [Silva 2010].

1.2 Definição do problema e motivações

O processamento de matrizes é de alta relevância para uma série de tecnologias. Por exemplo, a *Orthogonal Frequency Division Multiplexing (OFDM)* é uma das tecnologias mais promissoras pela alta taxa de dados na comunicação sem fio. Sistemas *Multiple Input Multiple Output (MIMO)*, que melhoram o desempenho e capacidade da comunicação sem fio usando varias antenas para receber e transmitir, são freqüentemente usados em conjunto com OFDM para melhorar a capacidade do canal e reduzir a interferência intersimbólica(ISI). Sistemas MIMO-OFDM exigem compensação no lado do receptor para remover o efeito do canal sobre o sinal. A inversão da matriz é essencial para um cálculo de compensação, em que o tamanho da matriz depende do número de antenas no transmissor e no receptor. Melhores taxas de dados podem ser obtidas usando mais antenas em ambos os lados, o que gera inversão de matrizes de tamanhos maiores, incrementando assim o custo computacional [A.Irturk B.Benson 2007].

Além disso, diferentes mecanismos de reconhecimento de sinais, como no caso de reconhecimento de voz, implicam na necessidade de operações matriciais complexas. Neste caso, são necessárias diferentes operações tanto algébricas quanto matriciais (entre elas inversões de matrizes) para o uso de métodos poderosos de análise de voz como *Linear Predictive Coding(LPC)*, também conhecido como *analise LPC* ou *auto-regressivo(AR)*. Este método é amplamente usado devido a sua rapidez e simplicidade, sendo ainda uma maneira eficaz de se estimarem os principais parâmetros de produção da voz [Huang e Hon 2001].

Por isso, o desenvolvimento de uma arquitetura capaz de inverter matrizes, reduzindo assim o custo computacional, empregando uma mínima quantidade de recursos em *hardware* sem perder precisão é relevante em aplicações de engenharia e ciência da computação.

1.3 Objetivos

1.3.1 Objetivo Geral

O objetivo geral deste trabalho é desenvolver uma arquitetura de baixo custo em área para inversões de matrizes, que utilizem representação em ponto flutuante, para implementações em arquiteturas reconfiguráveis do tipo FPGA, para aplicações de sistemas embarcados.

1.3.2 Objetivos Específicos

Os objetivos específicos deste trabalho são os seguintes:

1. Estudar e desenvolver diferentes algoritmos existentes na literatura para inversões de matrizes em *hardware*, usando arquiteturas reconfiguráveis.
2. Simular e implementar a inversão de matrizes em FPGAs.
3. Desenvolver uma arquitetura para inversões de matrizes, visando o mínimo consumo de recursos, maximizando as características de desempenho e acurácia.
4. Definir uma metodologia que permita comparações dos resultados obtidos.

1.4 Estrutura do Texto

Esta dissertação está organizada da seguinte forma:

No capítulo 2 é feita uma revisão bibliográfica sobre as diferentes plataformas de *hardware* paralelo que são usadas para implementações de diferentes algoritmos de cálculos matriciais: *Graphic Processing Units (GPUs)*, *Massively Parallel Processor Arrays (MPPAs)* e *FPGAs*, mostrando as vantagens e desvantagens de cada.

Em seguida, o capítulo 3 descreve os diferentes métodos de decomposição matricial (QR, LU e Cholesky) assim como a Redução de Gauss-Jordan. Descreve ainda os algoritmos para inversões de matrizes baseados nos diferentes métodos de decomposição e Redução de matrizes, e apresenta algumas definições importantes para a compreensão de conceitos relacionados aos cálculos matriciais.

No capitulo 4 apresentam-se as descrições das arquiteturas desenvolvidas, mostrando as unidades encarregadas de diferentes tarefas dentro da arquitetura, assim como as unidades aritméti-

cas de ponto flutuante adotadas. Resultados experimentais são discutidos no capítulo 5, seguido das conclusões no capítulo 6.

2 Plataformas Paralelas para Algoritmos de Cálculo Matricial

O cálculo matricial é um dos grandes tópicos de interesse na álgebra linear. Muitos algoritmos para cálculos matriciais são computacionalmente custosos e de alta demanda de recursos de memória. Sua resolução de forma rápida e eficiente tem sido objeto de pesquisa na área de computação paralela. Houve uma extensa pesquisa de novas arquiteturas que provêem vários tipos de núcleos de computação em um único dispositivo. Exemplos destas plataformas são *Chip Multiprocessadores (CMPs)*, *Graphic Processing Units (GPUs)*, *Massively Parallel Processor Arrays (MPPAs)*, entre outras, que implementam diferentes tipos de organizações arquiteturais, baseadas em processadores homogêneos ou heterogêneos. Embora cada uma dessas arquiteturas tenha suas vantagens inerentes assim como suas desvantagens, todas dependem fortemente da capacidade de realizar uma grande quantidade de paralelismo.

Como o mercado de aplicativos que usa cálculos matriciais está rapidamente se expandindo, há uma crescente demanda por ferramentas de projeto que possam acelerar o processo de otimizar os circuitos gerados, minimizando a utilização de recursos, por meio de redução das equações lógicas, assim como a otimização destes algoritmos para essas plataformas altamente paralelas [Kovar J. Kloub 2008]. Exemplos destas ferramentas incluem *NVIDIA Compute Unified Device Architecture (CUDA)* para *GPUs*, *aDesigner* para *MPPA*.

Se alguém quiser projetar e implementar uma plataforma que fornece grande quantidade de paralelismo, a escolha de um ambiente de desenvolvimento desempenha um papel importante. Os projetistas precisam decidir entre uma implementação de *hardware* ou software para explorar o paralelismo inerente dos algoritmos. A implementação em *hardware* dedicado consiste em projetar um circuito integrado de propósito específico (*ASIC*), que ofereça desempenho alto. Entretanto, esta abordagem leva um alto custo de desenvolvimento e produção, sendo viável apenas para volumes de produção muito grande. Por outro lado, a implementação em software dos algoritmos normalmente utiliza processadores digitais de sinais (*DSPs*), devido à facilidade de desenvolvimento e o tempo rápido de comercialização. *Field Programmable Gate Arrays (FPGAs)* são predominantes para aplicações de computação intensiva, pelo fato de fornecer uma grande quantidade de paralelismo. Os *FPGAs* desempenham um papel de intermediário entre

ASICs e *DSPs*, em relação a que eles têm a programação da sua estrutura (configuração) via software. Sistemas reconfiguráveis (baseados em FPGAs) são comprovadamente eficientes para permitir a distribuição ideal dos cálculos dentro de um *hardware* definido. Neste caso, existe a possibilidade de se ter um casamento ideal entre o paralelismo intrínseco dos algoritmos a serem implementados e o paralelismo fornecido pelas FPGAs[Pavel F. Otto 2001].

Neste contexto, pode se afirmar que plataformas de projeto baseadas em FPGAs (sistemas reconfiguráveis) são importantes para o projeto e teste de futuras arquiteturas multi-core extremamente paralelas (prototipação de sistemas), devido à sua arquitetura flexível, a qual pode ser reconfigurada ainda em tempo real [Hartenstein 2010].

2.1 Arquiteturas von Neumann

A Arquitetura de von Neumann (de John von Neumann), é uma arquitetura de computador que se caracteriza por armazenar seus programas no mesmo espaço de memória que os dados, podendo assim manipular tais programas. A arquitetura proposta por von Neumann reúne os seguintes componentes: (a) uma memória, (b) uma unidade aritmética e lógica (ALU), (c) uma unidade central de processamento (CPU), composta por diversos registradores, e (d) uma Unidade de Controle (CU), cuja função é buscar um programa na memória, instrução por instrução, e executá-lo sobre os dados de entrada.

O fato de se ter uma separação entre a memória e a CPU, assim como usar a mesma memória tanto para armazenar as instruções quanto para armazenar os dados, diminui o rendimento de processamento comparado com a quantidade de memória. Na maioria dos computadores modernos as CPU conseguem atingir tempos de processamento altos quando comparadas como o fluxo de informação desde/para a memória RAM, obrigando a mesma CPU fazer paradas enquanto aguarda pelos dados. Esta diferença em desempenho é conhecida como "*gargalo de von Neumann*", sendo este um dos maiores inconvenientes das arquiteturas convencionais [Hartenstein 2003].

2.2 Arranjos Massivos de Processadores em Paralelo (MPPAs)

Arranjos massivos de processadores em paralelo (MPPA) empregam até centenas de CPUs com arquitetura RISC assim como arranjos simples de memórias RAM, para introduzir um paralelismo massivo. As *MPPA* representam uma arquitetura *MIMD (Multiple Instruction streams, Multiple Data)*, onde as CPU empregam pequenas memórias locais as quais são instanciadas

em uma grade regular 2D, utilizando canais de comunicação entre eles. *MPPAs* são diferentes aos processadores multi-núcleo/*multi-core* (*CMPs*) ilustrada na figura 2.1. Um problema destas arquiteturas é que representam um caso especial do modelo de von Neumann, em que os processadores operam segundo o modelo clássico de von Neumann, usando uma memória compartilhada [Thomas L. Howes 2009] e grandes unidades de controle dedicadas à gestão e programação de tarefas. Existem diferentes arquiteturas *MPPA*: *Ambric* da *Ambric Inc.*, *picoArray* da *picoChip* e *SEAforth* da *IntellaSys*.

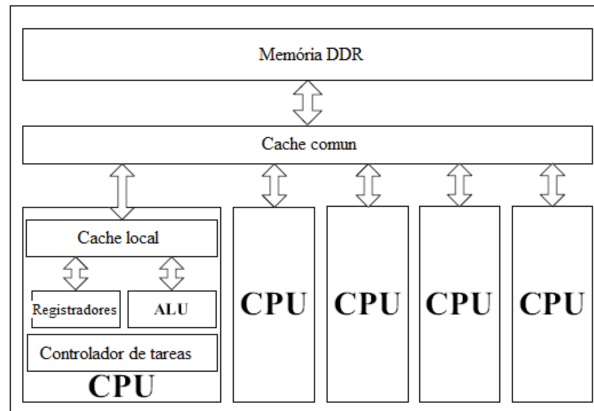

Figura 2.1. Aquitetura CPU Multi-nucleo

2.3 Unidades de Processamento Gráfico (GPUs)

Por volta do ano 2000 foi introduzido um novo tipo de processador, a *unidade de processamento gráfico (GPU)* ou também chamada *unidade de processamento visual (VPU)*, o qual era dedicado a atender às demandas de renderização de gráficos 3D em tempo real dos vídeo-jogos daquela época. Atualmente, um processador desse tipo ultrapassa a quantidade de 120 GFLOPs (Giga Floating-Point Operations per Second - Bilhões de operações de ponto-flutuante por segundo) de poder de processamento, enquanto um processador Intel Pentium IV de 3.0GHz executa apenas 12 GFLOPs [Kehtarnavaz N; Gamadia 2006]. Grande parte do poder de processamento das GPUs está em sua arquitetura que explora paralelismo e pipelining de modo avançado. Devido a que o conjunto de operações que esses processadores executam é restrito, este pôde ser extremamente otimizado. Uma GPU tipicamente possui diversos processadores dedicados trabalhando em paralelo, cada um com até 128 bits de profundidade de cores (4 vezes mais que os usuais 32 bits) por pixel [Kelly F.; Kokaram 1999]. Isso permite um grau de precisão na renderização muito alto, alcançando um realismo impressionante.

Essas unidades de processamento gráfico (GPUs) eram como ASICs, com poucos recursos

de configuração. Porém com o tempo foi sendo incorporada uma maior flexibilidade e possibilidade de programação, atraindo a atenção da comunidade de computação de alto desempenho. Atualmente diversas empresas e centros de pesquisa de todo o mundo já possuem trabalhos em desenvolvimento com o intuito de utilizar as GPUs para outras aplicações que não a renderização 3D, um conceito conhecido como GPGPU (General-Purpose processing on a Graphics Processing Unit). As GPUs já são utilizadas em problemas de processamento de imagens e vídeos em tempo real, em aplicações complexas como a reconstrução de imagens médicas de ressonância magnética e ultra-som. Um dos problemas enfrentados pelas primeiras gerações de GPUs eram os barramentos utilizados para conexão aos PCs, já que o barramento utilizado, PCI (Peripheral Component Interconect), não era suficientemente rápido para atender às aplicações. O advento do barramento PCI Express está, ao menos temporariamente, suprindo às necessidades de conexão desses dispositivos. Outra questão importante é que as GPUs atuais foram desenvolvidas para apresentarem um alto desempenho, a despeito do consumo de energia, consumindo ainda mais que um GPP comum. Essa característica deixaria as GPUs de fora do mercado de aplicações embarcadas, porém já há um esforço de desenvolvimento de GPUs para sistemas embarcados [Silva 2010].

2.4 *Field Programmable Gate Arrays (FPGAs)*

Esta seção discute alguns conceitos relevantes sobre *FPGAs* e que serão importantes neste trabalho. Adicionalmente, são apresentadas algumas características sobre um dispositivo específico, *Virtex-5* da Xilinx, o qual foi usado na implementação em *hardware*.

FPGAs foram inventadas pela Xilinx em 1984 e representam estruturas em formas de arranjos de portas/memórias (ou elementos lógicos). Uma arquitetura de um FPGA consiste em um conjunto de arranjos de blocos lógicos configuráveis, blocos I/O configuráveis assim como interconexões programáveis. Adicionalmente, os recursos lógicos podem incluir *ALUs*, elementos de memória e decodificadores. Os três diferentes tipos de elementos de programação para uma *FPGA* são *RAM estática*, *anti-fusível*, e *flash EPROM*.

Uma arquitetura genérica da *FPGA* é mostrada na figura 2.2, onde observa-se que segmentos interconectados de metal podem ser ligados de forma arbitrária por chaves programáveis para formar as redes de conexões entre as células. *FPGAs* podem ser usadas em praticamente qualquer sistema de lógica digital e proporcionam os benefícios de níveis elevados de integração, sem os riscos do custo do desenvolvimento personalizado dos ASICs.

A metodologia de projeto para implementar uma arquitetura em um *FPGA* é mostrada na figura 2.3. A mesma figura inclui as etapas necessárias para esta tarefa

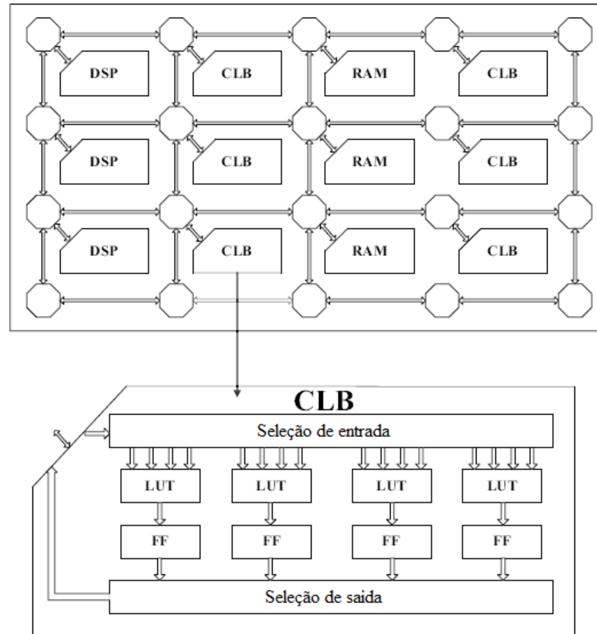

Figura 2.2. Arquitetura da FPGA e seus recursos: I/O de células, blocos lógicos (CLBs) e interconexões

[FPGA Design Flow Overview]: (a) desenho do circuito, (b) a síntese do 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 e o mapeamento tecnológico. Esta etapa garante que o projeto citado é otimizado para a arquitetura projetada. As conexões são criadas como uma *netlist* e são armazenadas como um arquivo *NGC* ou como um arquivo *EDIF*, para serem posteriormente usados com uma das seguintes ferramentas tecnológicas de síntese:

- Xilinx Synthesis Technology (XST).
- LeonardoSpectrum from Mentor Graphics Inc.
- Precision from Mentor Graphics Inc.
- Synplify and Synplify Pro from Synopsys.

Implementação do projeto segue da síntese de projeto e inclui os seguintes passos: (a) tradução, (b) mapeamento, (c) posicionamento e roteamento e (d) geração do arquivo de programação. A fase de *tradução* aplica as restrições de projeto ao *netlist* e cria o arquivo do projeto Xilinx. A fase de *mapeamento* 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.

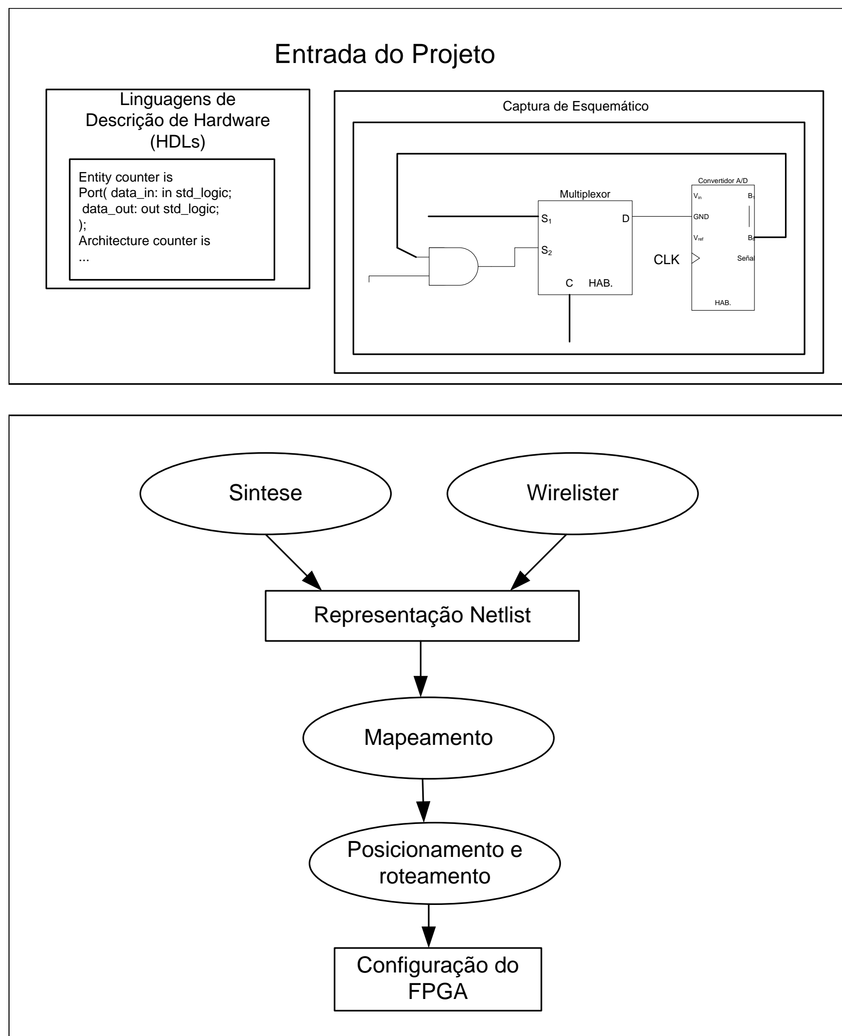

Figura 2.3. Etapas de um projeto com FPGAs (modificado de [Aragão 1998])

A verificação funcional verifica 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).

A verificação da sincronização verifica a sincronização do projeto em diferentes pontos no fluxo do mesmo com sincronização estática (depois do mapeamento e/ou posicionamento e roteamento) e simulação de sincronização (depois do mapeamento e/ou posicionamento e roteamento).

Os *FPGAs* não possuem nenhum conjunto de instruções fixas, sendo apropriadas para implementar soluções no contexto do *floware* e *configware* [Hartenstein 2006]. 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* [Hartenstein 2001].

2.4.1 Descrição geral da família Virtex-5

A família *Virtex-5* da Xilinx combina a segunda geração de avançados blocos modulares de silício com uma grande variedade de características flexíveis [Xilinx Product Specification Virtex-5 Family Overview (2009)]. A família *Virtex*-5 contém cinco plataformas diferentes (sub-famílias). Cada plataforma contém uma relação de características diferentes para atender às necessidades de uma ampla variedade de modelos de lógica avançada. Adicionalmente, as *FPGAs Virtex-5* contém muitos blocos hard-IP ao nível de sistema, incluindo poderosos blocos RAM de 36 Kbit / FIFOs, segunda geração de slices de DSP 25 x 18, tecnologia de seleção IO com controladores digitais de impedância embarcados, PLL, entre outros.

2.4.2 Arquitetura da Virtex-5

Os dispositivos Virtex-5 são arranjos de portas programáveis pelo usuário com vários elementos configuráveis e núcleos embarcados para o projeto de sistemas de alto desempenho e alta densidade. Os dispositivos Virtex-5 implementam as seguintes funcionalidades [Xilinx Product Specification Virtex-5 Family Overview (2009)]:

1. **Blocos I/O**: provêem a interface entre os pinos do circuito e a lógica de configuração interna. Os padrões I/O mais populares e de vanguarda são suportados por blocos I/O programáveis (*IOBs*).

2. **Blocos de lógica configurável (CLBs)**: os elementos lógicos básicos para os FPGAs da Xilinx provêem lógica síncrona e combinacional. Os *CLBs* das FPGAs Virtex-5 são baseados em LUTs de 6 entradas e fornecem recursos e desempenhos superiores em comparação com gerações anteriores de lógica programável.

3. **Blocos RAM**: provêem uma RAM flexível de 36Kbit tipo *true dualport* que são organizadas em cascata para formar blocos de memória maiores. Adicionalmente, os blocos de memória RAM da FPGA Virtex-5 podem ser configurados como dois blocos independentes de RAM de 18Kbit tipo *true dual port.*

4. **DSP48E**: são embarcados e contêm dois multiplicadores complementados de 25 x 18-bit e subtrator/somador/acumulador de 48-bit.

2.4.3 Conclusão do capítulo

As aplicações geralmente requerem diferentes características de desempenho dependendo da plataforma. Este é um problema inerente ao projeto e ao estilo de programação da plataforma. Para uma melhor escolha da plataforma de desenvolvimento, fatores tais como a programabilidade, desempenho e custo de programação devem ser levados em consideração.

Em geral, os *FPGAs* provêem a melhor expectativa de desempenho e flexibilidade, enquanto as *GPUs* e as *MPPAs* tendem a ser mais fácies de programar e exigem menos recursos de *hardware*. As *FPGAs* são altamente personalizáveis, enquanto *MPPAs* e *GPUs* fornecem grandes recursos de execução paralela. Estes dispositivos são processadores de propósito específicos e podem ser usados para apoiar processadores com o propósito de acelerar os cálculos complexos e intensivos das aplicações.

Concluindo este capitulo se apresentam na tabela 2.1 as vantagens e desvantagens das *FPGAs*, *GPUs* e *MPPAs* com o qual, para a nossa aplicação se escolheu as FPGAs como a melhor plataforma para o desenvolvimento.

Tabela 2.1. Comparação entre GPU, MPPA e FPGA

Dispositivo	Vantagem	Desvantagem
FPGAs	- Apto para aplicações que envolvem operações de controle de *hardware* de baixo nível e alta taxa de acessos à memória. - Baixo consumo de potência. - Aproximação de um chip personalizado, por exemplo, ASIC.	- Pouco apto para aplicações que exigem muita complexidade na lógica e fluxo de dados. - Não é um linguagem de descrição de *hardware* fácil (Verilog e VHDL). - Aumento do tempo de desenvolvimento e os esforços.
GPUs	- Apto para aplicações que não têm interdependências no fluxo de dados e as maiorias dos cálculos podem ser feitos em paralelo. - Alta largura de banda de memória e um grande número de núcleos programáveis com milhares de processos concorrendo em *hardware*. - Fácil e flexível de programar usando um linguagem de alto nível APIs. - Esforços e tempos de desenho relativamente curtos.	- Não apto para aplicações que têm alta taxa de acessos à memória e tem paralelismo limitado. - Alto consumo de potência.
MPPAs	- Baixo consumo de potência. - Fácil de programar usando *aDesigner design tool* e esforços e tempos de desenho relativamente curtos. - Apto para aplicações que não têm interdependências no fluxo de dados e as maiorias dos cálculos podem ser feitos em paralelo.	- Difícil de dividir as aplicações dadas entre centenas de processadores.

3 Decomposição, Redução e Inversão de Matrizes

Os cálculos matriciais são parte fundamental nos diferentes algoritmos para desenvolver diversas aplicações tais como processamento de sinal, visão computacional e cálculos financeiros. Este capítulo inicia com uma descrição dos trabalhos que implementam inversões de matrizes em FPGA. Adicionalmente, com o propósito de mostrar os diferentes métodos de inversão de matrizes e ter uma clareza sobre os mesmos, assim como também entender a nomenclatura usada, são apresentadas algumas definições conceituais sobre o álgebra de matrizes, incluindo suas principais características e alguns exemplos. É muito importante observar que existem muitos caminhos para se implementar estes algoritmos, que resultam em diferentes arquiteturas em função dos vários tipos de acessos à memória e níveis de paralelismo, entre outros.

3.1 Trabalhos correlatos à implementação de inversões de matrizes em *hardware*

Alguns artigos reportam implementações de inversões de matrizes usando diferentes métodos tais como Cholesky [Burian A. 2003], LU [A. Piirainen O. 2005] e Gauss-Jordan [P. Happonen A. 2005, Matos 2006].. Existem arquiteturas VLSI para inversões de matrizes usando o método de decomposição QR, embora sejam usadas em outras plataformas que não o FPGA. Na referência [A. 1989] é apresentado um algoritmo para inverter matrizes usando o método de decomposição QR-GR com uma arquitetura VLSI de arranjos sistólicos. Adicionalmente na referencia [C.K. PrasadS.H. 2007] é apresentado um algoritmo paralelo usando a decomposição QR-MGS.

A maioria dos trabalhos anteriores usam FPGAs para matrizes de tamanhos pequenos e não são facilmente extendidos para matrizes de tamanhos maiores como apresentado em [J. Wu D. 2007], onde há uma solução prática para matrizes maiores do que 4×4. Na referencia [D. Eilert J. 2007], a abordagem é extendida para matrizes de 16×16 e o análise para esse tamanho de matriz é apresentado. Entretanto, a implementação em FPGA é mostrada somente para matrizes de tamanhos 4×4. Uma outra arquitetura é apresentada em [M.Karkooti J.R.Cavallaro 2005], onde é usado o método de decomposição QR para inverter matrizes. Essa abordagem faz uso da decomposição do algoritmo *Givens Rotations* e seu mo-

delo de arranjos sistólicos foi implementado com um comprimento da palavra de 20 bits usando representação em ponto flutuante.

No entanto, a maioria dos trabalhos apresentam apenas representações em ponto fixo e arquiteturas que não podem ser extendidas para matrizes de tamanhos grandes. Uma exceção é a abordagem apresentada na referência [Duarte Horácio Neto 2009], onde a arquitetura proposta usa representação em ponto flutuante com precisão dupla e memórias externas para alcançar os requerimentos. A desvantagem do abordagem está no processo de normalização, o qual é executado em todos os elementos de cada linha na matriz. Isso é devido ao fato que tal normalização é realizada durante o processo de eliminação, que produz uma multiplicação para cada elemento da linha. Por exemplo, para uma matriz $n \times n$, o número de multiplicações necessárias no processo de normalização é dado pela equação 3.1, tendo em conta que este processo é aplicado para a matriz aumentada.

$$\sum_{i=1}^{n-1} i + n \times \sum_{i=1}^{n-1} i \tag{3.1}$$

Executando o processo de normalização no final, operando apenas na parte direita da matriz aumentada, este processo pode reduzir à n^2. Um outro problema desse abordagem é que o estudo em termos de precisão (relacionado sobre todo o processo de inversão) não é apresentado. Adicionalmente, os resultados apresentados nesse trabalho em termos de consumo de recursos e desempenho são imprecisos; em particular, o tamanho das matrizes é omitido nos dados de consumo de recursos, assim como os resultados de tempo para matrizes de tamanhos que não são sintetizáveis para o dispositivo virtex5 usado. Outro aspecto importante a observar, é que na referência [Duarte Horácio Neto 2009] usam o algoritmo de divisão de Goldschmidt enquanto que nas arquiteturas propostas, usa-se o algoritmo de divisão de Newton-Raphson que apresentou melhor desempenho que o anterior (ver referencia [Sanchez 2009]).

3.2 Definições Gerais

Antes de apresentar os diferentes métodos de decomposição e qualquer conceito matricial, serão introduzidos alguns conceitos e definições genéricos considerados relevantes ao problema. Os cálculos matriciais podem ser vistos como uma combinação de muitas operações algébricas lineares com uma hierarquia.

Alguns exemplos são:

- Produto escalar ou produto ponto: são somas e multiplicações;

- Multiplicações matriz/vetor: são series de produtos ponto;
- Multiplicações matriz/matriz: é um conjunto de multiplicações matriz/vetor;

Definição 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 (equação 3.2),

$$A_{ij} = \begin{pmatrix} a_{11} & a_{12} & \cdots & \cdots & a_{1j} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & \cdots & a_{2j} & \cdots & a_{2n} \\ \vdots & \vdots & \cdots & \cdots & \vdots & \cdots & \vdots \\ a_{i1} & a_{i2} & \cdots & \cdots & a_{ij} & \cdots & a_{in} \\ \vdots & \vdots & & & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & \cdots & a_{mj} & \cdots & a_{mn} \end{pmatrix} \tag{3.2}$$

em que i-enésima linha de A e a j-enésima coluna de A são representadas segundo as expressões 3.3 e 3.4 respectivamente.

$$\begin{pmatrix} a_{i1} & a_{i2} & \cdots & a_{in} \end{pmatrix} (1 \leq i \leq m) \tag{3.3}$$

$$\begin{pmatrix} a_{1j} \\ a_{2j} \\ \vdots \\ a_{mj} \end{pmatrix} (1 \leq j \leq n) \tag{3.4}$$

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-enésima linha e na j-enésima coluna de A, é denominado de i,j-enésimo de A ou o elemento (i,j) de A. Frequentemente a equação 3.2 é representada como $A = \begin{pmatrix} a_{ij} \end{pmatrix}$ [Bernard Kolman. 2001, ANTON HOWARD A. 2001].

Definição 2: Uma matriz quadrada $A = \begin{pmatrix} a_{ij} \end{pmatrix}$ em que todo elemento fora da diagonal principal é zero, isto é, $a_{ij} = 0$ para $i \neq j$, é denominada de *matriz diagonal* [Bernard Kolman. 2001, ANTON HOWARD A. 2001].

Definição 3: Uma matriz diagonal $A = \begin{pmatrix} a_{ij} \end{pmatrix}$ 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* [Bernard Kolman. 2001, ANTON HOWARD A. 2001].

Definição 4: Se $A = \begin{pmatrix} a_{ij} \end{pmatrix}$ e $B = \begin{pmatrix} b_{ij} \end{pmatrix}$ são matrizes $m \times n$, a *soma* de A e B é uma matriz $C = \begin{pmatrix} c_{ij} \end{pmatrix}$, $m \times n$, definida pela equação 3.5,

$$c_{ij} = a_{ij} + b_{ij} \tag{3.5}$$

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 [Bernard Kolman. 2001, ANTON HOWARD A. 2001].

Definição 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 [Bernard Kolman. 2001, ANTON HOWARD A. 2001].

Definição 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 [Bernard Kolman. 2001, ANTON HOWARD A. 2001].

Definição 7: O *produto escalar* ou *produto interno* dos vetores de dimensão n **a** e **b** é a soma dos produtos dos elementos correspondentes. Dessa maneira, 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* $\mathbf{a} \times \mathbf{b}$ está definido segundo a equação 3.6 [Bernard Kolman. 2001, ANTON HOWARD A. 2001].

$$a \cdot b = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n = \sum_{i=1}^{n} a_i b_i \tag{3.6}$$

Definição 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 3.7 [Bernard Kolman. 2001, ANTON HOWARD A. 2001].

$$C_{ij} = a_{i1} b_{1j} + a_{i2} b_{2j} + \cdots + a_{ip} b_{pj} = \sum_{k=1}^{p} a_{ik} b_{kj} \quad (1 \leq i \leq m, 1 \leq j \leq n) \tag{3.7}$$

Definição 9: A matriz escalar $n \times n$, cujos elementos da diagonal principal são todos iguais a 1 (expressão 3.8), é denominada de *matriz identidade de ordem n* [Bernard Kolman. 2001, ANTON HOWARD A. 2001].

$$\begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \\ 0 & 0 & \cdots & 1 \end{pmatrix} \tag{3.8}$$

Definição 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) esta na *forma escalonada por linhas* [Bernard Kolman. 2001, ANTON HOWARD A. 2001].

Definição 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-enésima 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-enésima 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 12: Uma matriz $An \times n$ é chamada *triangular superior* se todos os elementos a baixo da diagonal principal são zero. Assim, a matriz ilustrada na equação 3.9 é uma matriz *triangular superior* [Bernard Kolman. 2001, ANTON HOWARD A. 2001].

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22} & a_{23} \\ 0 & 0 & a_{33} \end{pmatrix} \tag{3.9}$$

Definição 13: 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 3.10 é uma matriz *triangular inferior* [Bernard Kolman. 2001, ANTON HOWARD A. 2001].

$$\begin{pmatrix} a_{11} & 0 & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \tag{3.10}$$

Definição 14: Uma matriz $An \times n$ é chamada *invertível* (ou *não-singular*) 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 existir, então A é chamada de *singular* (ou *não-inversível*) [Bernard Kolman. 2001, ANTON HOWARD A. 2001].

Definição 15: Seja $A = \left(a_{ij}\right)$ uma matriz $n \times n$. Definimos o *determinante* de A (escreve-se $det(A)$ ou $|A|$) pela expressão 3.11,

$$det(A) = |A| = \sum_{1}^{n} (\pm) a_{1,j1} a_{1,j2} \cdots a_{n,jn}, \tag{3.11}$$

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 [Bernard Kolman. 2001, ANTON HOWARD A. 2001].

Definição 16: 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-enésima linha e da j-enésima 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 como se mostra na equação 3.12 [Bernard Kolman. 2001, ANTON HOWARD A. 2001].

$$A_{ij} = (-1)^{i+j} det(M_{ij}) \tag{3.12}$$

Definição 17: 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*-enésimo elemento é o co-fator de A_{ij} de a_{ij}. Assim, a matriz ilustrada na equação 3.13, é a matriz adjunta de A [Bernard Kolman. 2001, ANTON HOWARD A. 2001].

$$adj\ A = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{n1} \\ A_{21} & A_{22} & \cdots & A_{n2} \\ \vdots & \vdots & & \vdots \\ A_{1n} & A_{2n} & \cdots & A_{nn} \end{pmatrix} \tag{3.13}$$

3.3 Decomposição de Matrizes

Nesta seção, descrevem-se quatro diferentes métodos de fatoração (ou decomposição) conhecidos que permitem realizar cálculos matriciais: (a) *Método de decomposição QR*, (b) *Método de decomposição LU*, (c) *Método de decomposição Cholesky* e (d) *Redução Gauss-Jordan.* Observa-se que os métodos LU e Cholesky são geralmente usados com matrizes quadradas definidas positivas e não-singulares. A decomposição QR e a redução de Gauss-Jordan são mais gerais e se podem aplicar para qualquer matriz quadrada. Para matrizes quadradas n denota o tamanho da matriz assim como $n = 4$ denota uma matriz de 4×4.

3.3.1 Decomposição LU

É uma forma de fatoração de uma matriz não-singular como o produto de uma matriz triangular inferior(L) e uma matriz triangular superior(U). Se A é uma matriz quadrada e não-singular, a matriz A pode ser decomposta em matrizes originais triangular inferior e triangular superior. A decomposição LU de uma matriz A é mostrado como $A = L \times U$ em que L e U são as matrizes inferiores e superiores respectivamente como se mostra na figura 3.1 [ANTON HOWARD A. 2001].

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ L_{21} & 1 & 0 & 0 \\ L_{31} & L_{32} & 0 & 0 \\ L_{41} & L_{42} & L_{43} & 0 \end{bmatrix} ; U = \begin{bmatrix} U_{11} & U_{12} & U_{13} & U_{14} \\ 0 & U_{22} & U_{23} & U_{24} \\ 0 & 0 & U_{33} & U_{34} \\ 0 & 0 & 0 & U_{44} \end{bmatrix}$$

Figura 3.1. Decomposição LU (obtido de [Irturk 2009])

3.3.2 Decomposição Cholesky

Foi assim denominada em homenagem a André-Louis Cholesky que estabeleceu que uma matriz simétrica e definida positiva [ANTON HOWARD A. 2001], pode ser decomposta em uma matriz triangular inferior e sua transposta. Esta decomposição toma a matriz simétrica definida positiva, transformando-a em uma única matriz triangular inferior com entradas diagonais positivas. O método de decomposição de Cholesky de uma matriz A é mostrada como $A = G \times G^T$, em que G é uma única matriz triangular inferior, triangulo de Cholesky, e G^T é a transposta desta matriz triangular inferior, assim como é mostrado na figura 3.2.

$$G = \begin{bmatrix} G_{11} & 0 & 0 & 0 \\ G_{21} & G_{22} & 0 & 0 \\ G_{31} & G_{32} & G_{33} & 0 \\ G_{41} & G_{42} & G_{43} & G_{44} \end{bmatrix}; G^T = \begin{bmatrix} G_{11} & G_{21} & G_{31} & G_{41} \\ 0 & G_{22} & G_{32} & G_{42} \\ 0 & 0 & G_{33} & G_{43} \\ 0 & 0 & 0 & G_{44} \end{bmatrix}$$

Figura 3.2. Decomposição Cholesky (obtido de [Irturk 2009])

3.3.3 Decomposição QR

A decomposição QR é uma operação elementar que decompõe a matriz dada em uma ortogonal e em uma matriz triangular superior. A decomposição QR da matriz A é mostrada 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 se mostra na figura 3.3.

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

Figura 3.3. Decomposição QR (obtido de [Irturk 2009])

3.4 Redução de Gauss-Jordan

A redução de Gauss-Jordan, é uma versão da redução de Gauss que zera os elementos a cima e a baixo do elemento pivô, conforme ele percorre a matriz [ANTON HOWARD A. 2001, Bernard Kolman. 2001]. 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 um 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 redução de Gauss-Jordan transforma a matriz em uma *forma escalonada reduzida*, enquanto a redução de Gauss transforma na *forma escalonada*.

O procedimento de redução 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 (seção 3.2).
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.

3.5 Os Métodos de Inversão de Matrizes

Para encontrar a inversão da matriz, deve-se considerar este problema transformando-o em um problema de fácil decomposição, o que resultará em uma maior simplicidade tanto analítica quanto computacional. Os métodos de decomposição são geralmente vistos como métodos preferidos para inversão de matrizes, devido ao fato que se adapta bem para matrizes de tamanhos grandes, enquanto a complexidade dos métodos analíticos incrementa dramaticamente a medida que o tamanho da matriz aumenta. Entretanto, para matrizes pequenas, os métodos analíticos podem explorar uma grande quantidade de paralelismo funcionando melhor que os métodos de decomposição. Nesta seção se apresentam as diferentes formas de inverter matrizes baseados nos métodos de decomposição e redução anteriormente explicados nas seções 3.3 e 3.4.

3.5.1 Inversão de Matriz baseada na decomposição LU

A solução para a inversão da matriz A, A^{-1}, usando decomposição LU, é mostrada segundo a equação 3.14,

$$A^{-1} = U^{-1} \times L^{-1} \tag{3.14}$$

onde a solução consiste de quatro diferentes etapas: (a) decomposição LU da matriz dada, (b) inversão da matriz triangular inferior, (c) inversão da matriz triangular superior e (d) multiplicação de matrizes, como ilustra a figura 3.4.

A decomposição LU é o cálculo dominante e mais complexo computacionalmente, sendo as outras 3 etapas relativamente simples, devido ao fato das estruturas de matrizes triangulares.

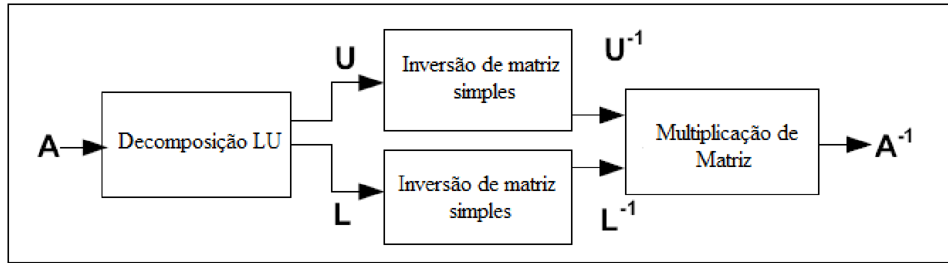

Figura 3.4. As etapas da solução da inversão de matrizes utilizando decomposição LU (modificado de [Irturk 2009])

3.5.2 Inversão de Matriz baseada na decomposição Cholesky

A solução para a inversão da matriz A, A^{-1}, usando a decomposição de Cholesky, é mostrada segundo a equação 3.15,

$$A^{-1} = G^{-1} \times G^{T^{-1}} \tag{3.15}$$

em que a solução consiste de quatro diferentes etapas: (a) decomposição de Cholesky, (b) inversão da transposta da matriz triangular inferior, (c) inversão da matriz triangular inferior e (d) multiplicação das matrizes, tal como se mostra na figura 3.5. A decomposição de Cholesky é o cálculo dominante e mais complexo computacionalmente, sendo as outras 3 etapas relativamente simples, devido ao fato das estruturas de matrizes triangulares.

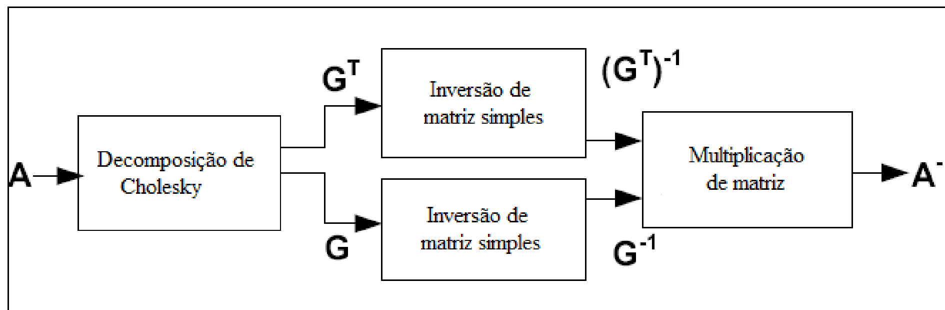

Figura 3.5. As etapas da solução da inversão de matrizes utilizando decomposição Cholesky (modificado de [Irturk 2009])

3.5.3 Inversão de Matriz baseada na decomposição QR

A solução para a inversão da matriz A, A^{-1}, usando decomposição QR é mostrada segundo a equação 3.16,

$$A^{-1} = R^{-1} \times Q^T \tag{3.16}$$

onde a solução consiste de três etapas diferentes: (a) decomposição QR, (b) inversão da matriz triangular superior e (c) multiplicação de matrizes (figura 3.6). A decomposição QR é o cálculo dominante e mais complexo computacionalmente, sendo as outras 2 etapas relativamente simples, devido ao fato da estrutura de matriz triangular superior de R.

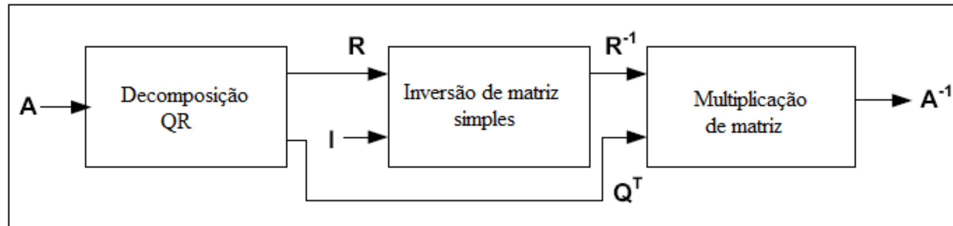

Figura 3.6. As etapas da solução da inversão de matrizes utilizando decomposição QR (modificado de [Irturk 2009])

3.5.4 Inversão de Matriz baseada na Redução de Gauss-Jordan

A inversão da matriz baseada na redução de Gauss-Jordan apresenta uma forma direta e rápida de obter a inversa da matriz A, A^{-1}. Este método baseia-se em se ter a matriz A com uma matriz identidade I para obter uma matriz aumentada, e transformar a matriz A na matriz identidade I, assim como todos os cálculos feitos para transformar esta matriz A na matriz I, são executados na matriz I para transformar está na matriz A^{-1}, tal como ilustrado na equação 3.17.

$$\left[\begin{array}{c|c} A & I \end{array}\right] \Rightarrow A^{-1}\left[\begin{array}{c|c} A & I \end{array}\right] \Rightarrow \left[\begin{array}{c|c} I & A^{-1} \end{array}\right] \tag{3.17}$$

Os passos para calcular a inversa da matriz A são os seguintes:

1. Formar a matriz $[A|I_n]$ $n \times 2n$ obtida juntando-se a matriz identidade I_n e a matriz A dada.

2. Calcular a forma escalonada reduzida por linhas da matriz obtida no passo 1 utilizando operações elementares nas linhas. Lembrar que o que fizer em uma linha de A também deverá fazer na linha correspondente de I_n.

3. Supor que o passo 2 produz a matriz $[C|D]$ na forma escalonada reduzida por linhas, em que se cumprem qualquer das seguintes condições:

(a) Se $C = I_n$, então $D = A^{-1}$.

(b) Se $C \neq I_n$, então C tem uma linha nula. Neste caso, A é singular e A^{-1} não existe.

Observa-se que o método de inversão de matriz baseado na redução de Gauss-Jordan apenas precisa três operações algebraicas básicas: (a) soma,(b) multiplicação e (c) divisão.

3.5.5 Inversão de Matriz usando método analítico

Outro método para inverter uma matriz A é o método analítico o qual usa a matriz adjunta, $Adj(A)$, e o determinante, $detA$. Este cálculo é dado pela equação 3.18.

$$A^{-1} = \frac{1}{det(A)} \times Adj(A) \tag{3.18}$$

A matriz adjunta é a transposta da matriz de cofatores, em que a matriz de cofatores é formada usando os determinantes da matriz de entrada com os sinais dependendo da posição [ANTON HOWARD A. 2001].

3.5.6 Complexidade dos algoritmos de inversão de matrizes

Finalmente a complexidade dos algoritmos é apresentada nas figuras 3.7 e 3.8, ilustrando que o algoritmo baseado na redução de Gauss-Jordan usa menos operações para realizar a tarefa de inversão de matrizes.

Os métodos de inversão baseados na decomposição usam entre suas operações: (1) A decomposição, (2) inversões de matrizes e (3) multiplicações matriciais. Alguns desses métodos usam outras operações aritméticas como é o caso da raiz quadrada no cálculo da inversão baseada no método de decomposição QR, quando usado o processo de Gram-Schmidt. Já o método matemático de inversão de matrizes baseado na redução de Gauss-Jordan usa apenas 3 operações diferentes, assim como são necessários menos acessos à memória na implementação em *hardware*.

3.6 Conclusão do Capítulo

Neste capitulo foram apresentados diferentes métodos para inverter matrizes ilustrando as etapas necessárias para realizar a operação. Cada método tem suas vantagens e desvantagens, em que, parâmetros tais como a complexidade das operações aritméticas, 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.

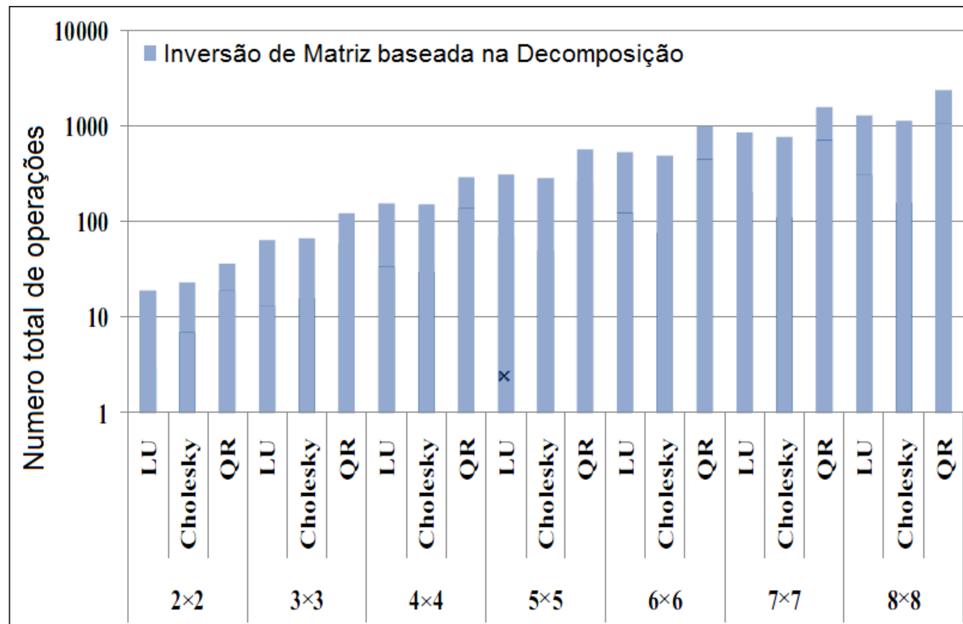

Figura 3.7. Número total de operações no domínio logarítmico para a inversão de matrizes baseada nos métodos de decomposição (modificado de [Irturk 2009])

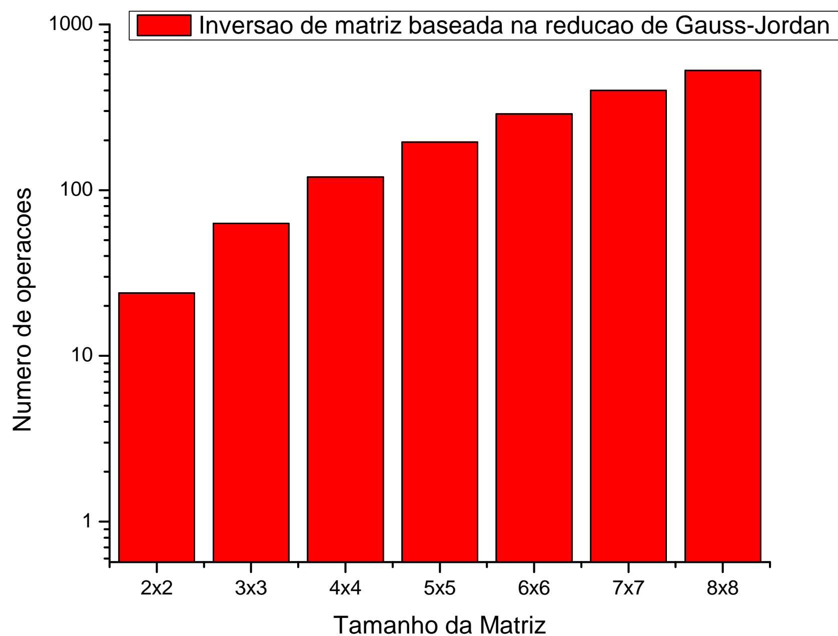

Figura 3.8. Número total de operações no domínio logarítmico para a inversão de matrizes baseada na redução de Gauss-Jordan

A redução de Gauss-Jordan é um método direto, o qual, precisa só de três operações aritméticas diferentes (parâmetro importante devido ao consumo de recursos das operações em ponto

flutuante), ao contrario da decomposição QR, quando usado o processo de Gram-Schmidt, que precisa de uma operação a mais (raiz quadrada) [Bernard Kolman. 2001]. Adicionalmente, os métodos de inversão de matrizes baseados na decomposição usam internamente decomposição, inversão e multiplicação de matrizes, o que aumenta o número de acessos à memória (métodos não diretos). Por outro lado, o método analítico não é uma opção viável, uma vez que usa o determinante entre seus cálculos. Portanto, a redução de Gauss-Jordan tem as melhores características buscadas para resolver o problema da inversão de matrizes de grande tamanho implementadas em *hardware*, fato pelo qual foi escolhido.

4 Implementação

Neste capítulo serão descritas as implementações e modificações das arquiteturas para inversão de matrizes, baseadas na redução de Gauss-Jordan. Detalhes das soluções abordadas serão fornecidos, de modo a mostrar as diferentes formas de implementação. Também serão explicadas algumas características das bibliotecas de ponto flutuante utilizadas, assim como o tipo de memória interna da FPGA usada na implementação e o acesso à mesma.

4.1 Bibliotecas de ponto flutuante

As bibliotecas de ponto flutuante usadas neste projeto foram desenvolvidas no laboratório GRACO (Grupo de Automação e Controle) que dá suporte ao Programa de Sistemas Mecatrônicos da Universidade de Brasília. Esta seção explica brevemente os algoritmos utilizados, assim como a sua implementação em *hardware* [Sanchez 2009].

4.1.1 Unidade de Soma/Subtração

O algoritmo mostrado na figura 4.1 implementa o operador de soma/subtração em ponto flutuante.

O primeiro passo do algoritmo é detectar se os operandos de entrada são valores inválidos: zero ou infinito. Se os números são válidos, o bit implícito é adicionado às mantissas. Um deslocamento a direita deveria ser feito sobre o menor dos dois números. O número de posições deslocadas à direta depende da diferença entre o expoente maior e menor. Posteriormente, é determinado o tipo de operação a ser executada ($\pm$). Subsequentemente, expoente e a mantissa atuais são normalizadas. O algoritmo finaliza com a concatenação do sinal, expoente e mantissa resultantes.

Esta unidade permite calcular a soma e/ou subtração de dois números em ponto flutuante, selecionando o tipo de operação a ser realizada com a porta de entrada *op*. Além do anterior, esta unidade tem as portas de entrada *reset*, *clock*, *start_ i*, *op_ a*, *op_ b*, em que *op_ a* e *op_ b* são os números e *start_ i* serve para dar início à operação desejada. Como portas de saída têm-se:

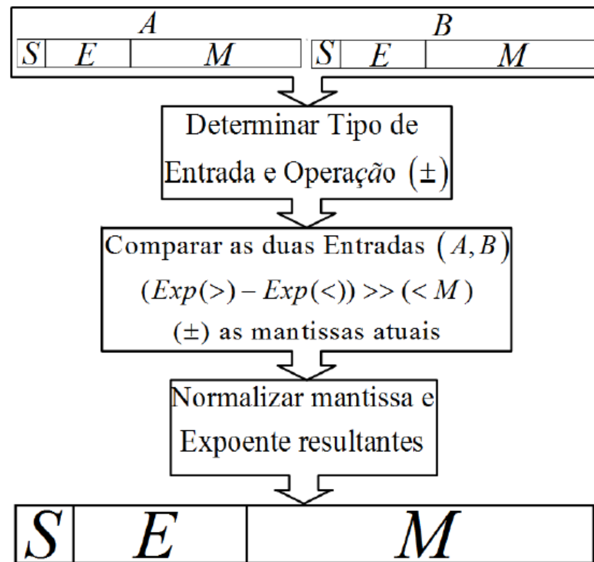

Figura 4.1. Algoritmo seguido na implementação da unidade de Soma/Subtração (Obtido de [Sanchez 2009])

addsub_ out sendo o resultado e *ready_ as* indica se a operação terminou ou não (ver figura 4.2).

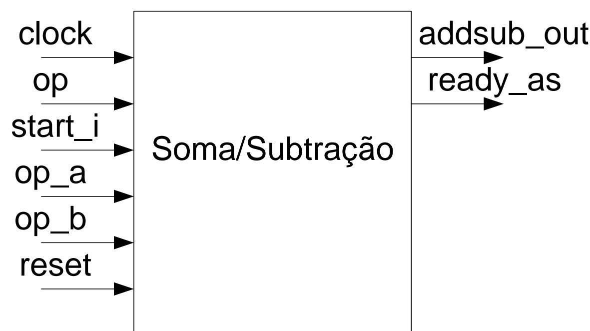

Figura 4.2. entradas/saidas para o bloco soma/subtração

Está unidade foi desenvolvida usando máquinas de estado finitas (*Finite State Machine*- FSM) contendo os estados de *waiting*, *add*, *sub* e *postnorm*. No estado de *waiting*, o circuito encontra-se em um estado de espera até que o sinal de *start_ i* seja levado para 1. Neste momento, as entradas são validadas, sendo possível detectar se são zero, ou entradas inválidas (tais como infinito ou números não definidos). A seguir, é feita a operação lógica *op_ a(S)XOR(opXORop_ b(S))*, em que S é o sinal dos operandos. O resultado desta operação define o estado seguinte *add* ou *sub*. Para finalizar, no estado de *postnorm* são concatenados em um registro o sinal, o expoente e a mantissa resultante, e é ativada a porta de saída *ready_ as*. Os valores do erro quadrático médio do resultado da operação de Soma/Subtração comparado com o respectivo resultado em MatLab são mostrados na tabela 4.1 para diferentes tamanhos de palavras.

Tabela 4.1. MSE da unidade Soma/Subtração para diferentes larguras de bits (Obtido de [Sanchez 2009])

Bit-width (*Exp*, *Man*)	MSE
16(5, 10)	2,24E-03
24(6,17)	2,27E-07
32(8,23)	2,76E-11
48(9,38)	1,31E-15
64(11,52)	1,26E-17

4.1.2 Unidade de Multiplicação

Os passos seguidos para implementar o operador de multiplicação são mostrados na figura 4.3.

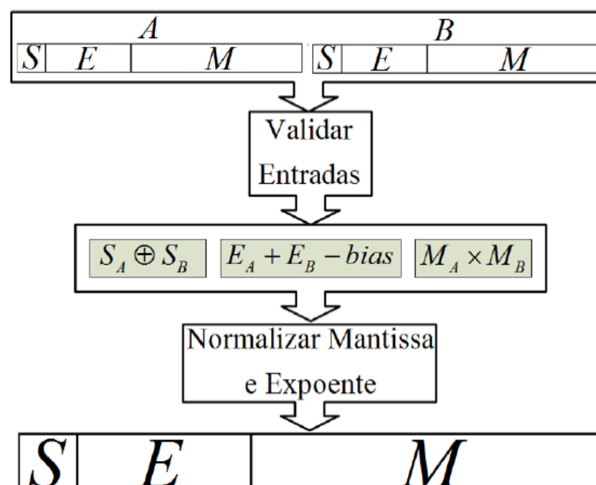

Figura 4.3. Algoritmo seguido na implementação da unidade de Multiplicação (Obtido de [Sanchez 2009])

Esta unidade permite calcular a multiplicação de dois números em ponto flutuante no qual temos como portas de entrada o *reset*, o *clock* como sinal de relógio, assim como os dois operandos de entrada *op_ a* e *op_ b* e a porta que da início à realização dos cálculos *start_ i*. Adicionalmente, têm-se como portas de saída o resultado da multiplicação *mul_ out* e *ready_ mul* é usada como o sinal de indicação do final da operação (ver figura 4.4).

Esta unidade também usa uma *FSM*, tendo como estados *waiting*, *multiplier* e *postnorm*. No estado de *waiting*, o circuito fica em espera até que a entrada *start_ i* seja ativada. Neste momento, as entradas são validadas. Depois, no estado *multiplier* (ver figura 4.5, linha 84), são somados os expoentes e subtraído o *bias*, as mantissas são multiplicadas levando em conta o bit implícito e, finalmente, é feita a normalização do resultado. Por último, no estado de *postnorm*

Figura 4.4. Entradas/Saidas para o bloco Multiplicação

são concatenados o sinal, o expoente e a mantissa resultante, assim como é atribuído o valor de 1 na sinal *ready_ mul*. A tabela 4.2 mostra os valores do erro quadrático médio do resultado da operação de multiplicação comparado com o resultado no MatLab [Sanchez 2009].

```
84  when multiplier =>
85     s_add_exp := ('0' & op_a(FRAC_WIDTH+EXP_WIDTH-1 downto FRAC_WIDTH)) + ('0' & op_b(FRAC_WIDTH+EXP_WIDTH-1 downto FRAC_WIDTH));
86     s_add_exp := s_add_exp - Bias;
87     s_mul_man := ('1' & op_a(FRAC_WIDTH-1 downto 0)) * ('1' & op_b(FRAC_WIDTH-1 downto 0));
88
89     if s_mul_man((FRAC_WIDTH*2)+1) = '1' then
90        s_add_exp := s_add_exp + '1';
91        s_mantisa := s_mul_man(FRAC_WIDTH*2 downto FRAC_WIDTH+1);
92     else
93        s_mantisa := s_mul_man((FRAC_WIDTH*2)-1 downto FRAC_WIDTH);
94     end if;
```

Figura 4.5. Código da Unidade de Multiplicação em ponto flutuante (Obtido de [Sanchez 2009])

Tabela 4.2. MSE da unidade multiplicação para diferentes larguras de bits (Obtido de [Sanchez 2009])

Bit-width (Exp, Man)	MSE
16(5, 10)	2,21E-02
24(6,17)	9,53E-04
32(8,23)	1,53E-07
48(9,38)	3,96E-12
64(11,52)	5,87E-16

4.1.3 Unidade de Divisão

Na literatura existem diferentes algoritmos usados na implementação do operador aritmético de divisão como *Goldschmidt* e *Newton - Raphson* [Muñoz-Arboleda D. F. Sanchez e Ayala-Rincón 2009], Nesta seção é mostrado um algoritmo geral para o cálculo deste operador (figura 4.6), para finalmente descrever o algoritmo *Newton - Raphson* usado no desenvolvimento deste trabalho. Este algoritmo (*Newton - Raphson*) foi usado por ter características de desempenho

melhores do que o algoritmo *Goldschmidt*, segundo [Sanchez 2009].

Sejam a e b dois números reais representados no padrão IEEE-754, em que a representa o dividendo e b o divisor. Os passos podem ser descritos da seguinte forma: (a) detectar divisões por zero e entradas inválidas e separar o sinal, o expoente e a mantissa de a e b, adicionando o bit implícito na mantissa; (b) executar em paralelo o cálculo do expoente resultante usando a fórmula 4.1; (c) determinar o produto dos sinais e calcular a mantissa resultantes usando o algoritmo *Newton Raphson* que posteriormente será descrito, e finalmente, (d) concatenar o sinal, expoente e mantissa resultantes.

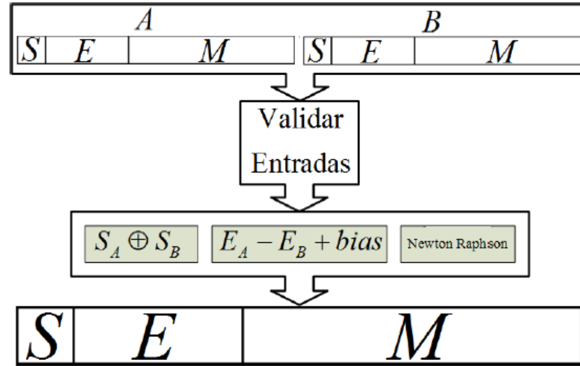

Figura 4.6. Algoritmo seguido na implementação da unidade de Divisão (Modificado de [Sanchez 2009])

$$E_a - E_b + bias \tag{4.1}$$

4.1.3.1 Algoritmo Newton Raphson

Este algoritmo tem como parâmetros de entrada as mantissas normalizadas do dividendo (M_N) e do divisor (M_D), que implicitamente satisfazem que ($M_N \geq 1, M_D \leq 2$), e o algoritmo calcula o quociente, partindo de uma semente aproximada ao valor $y_0 \approx 1/D$, em que D é o divisor. Depois as equações 4.2 e 4.3 devem ser executadas iterativamente. Depois da i^{th} iteração o produto da multiplicação de $N \times y_{i+1}$ produz uma aproximação a N/D [Muñoz-Arboleda D. F. Sanchez e Ayala-Rincón 2009, Sanchez 2009].

$$p = Dy_i \tag{4.2}$$

$$y_{i+1} = y_i(2 - p) \tag{4.3}$$

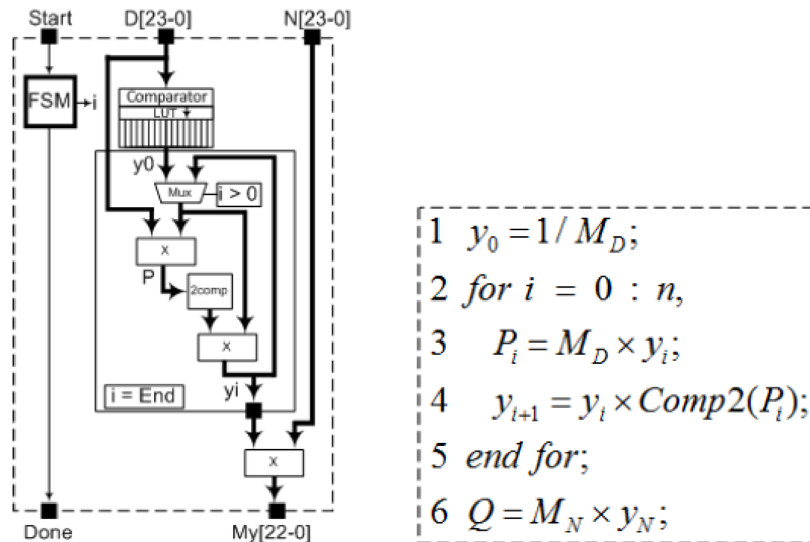

Figura 4.7. Implementação do algoritmo Newton - Raphson para divisão (Obtido de [Sanchez 2009])

Na figura 4.7 é apresentada a implementação *hardware* e *software* deste algoritmo. Na arquitetura *hardware* as aproximações inicias a $1/MD$ são armazenadas em uma *look-up table* e contínuos refinamentos de y_0 são realizados. Este algoritmo calcula o quociente na iteração final. Nesta arquitetura as operações de multiplicação são executadas usando os multiplicadores internos disponíveis no FPGA [Sanchez 2009].

4.2 Implementação do Algoritmo de Gauss-Jordan

Nesta seção apresentam-se os passos normalmente seguidos para a implementação do algoritmo, assim como as arquiteturas desenvolvidas para se obter a inversão da matriz baseada na redução de Gauss-Jordan. Antes disso, é importante esclarecer que tais arquiteturas partem do suposto que as matrizes a inverter são todas inversíveis, ou seja, que o determinante de A (*det* A), sendo A a matriz a inverter, seja diferente de zero ($det\,(A) \neq 0$).

4.2.1 O Algoritmo Gauss-Jordan

A redução de Gauss-Jordan é um método baseado na redução Gaussiana, que põe zeros a cima e a baixo de cada pivô dado para obter a matriz inversa [ANTON HOWARD A. 2001].

O método de Gauss-Jordan, primeiramente, aplica a redução de Gauss a uma matriz aumentada $n \times 2n$, $\tilde{A} = [AB]$, dando como resultado uma matriz da forma $[CD]$ na qual C é uma matriz triangular superior. Depois, $[CD]$ é transformada aplicando operações elementares de

linhas em uma matriz aumentada da forma $[IK]$. Por último, é extraído o componente direito K da matriz aumentada, o que corresponde à matriz inversa.

A seguir, são apresentadas duas modificações feitas no algoritmo que foram implementadas em *hardware* para melhorar a precisão nos cálculos, assim como algumas mudanças importantes na arquitetura, para alcançar melhores resultados tanto em consumo de recursos quanto no desempenho. Por outro lado, são apresentadas outras modificações feitas tais como a inclusão de *pipeline*, o uso de dois multiplicadores e dois somadores e o uso de 10 multiplicadores e 10 somadores para se obter assim mais paralelismo.

4.2.1.1 Algoritmo GJ-Pivô_A

Os quatro principais passos do algoritmo a partir da linha $i = 1$ da matriz aumentada $[AI]$ são apresentados a seguir:

- *Lider parcial*: localizar o pivô, que corresponde a qualquer número diferente de zero da coluna i, considerando os elementos nas linhas i até n. Posteriormente, é permutada a linha i com a linha que contém o pivô.
- *Forward elimination*: eliminar todos os elementos a baixo da diagonal da coluna i, subtraindo cada linha a baixo da linha do pivô por um múltiplo da mesma. Seguidamente, incrementa-se i e repete-se a partir do passo 1 (se $i < n$), até formar a matriz triangular superior.
- *Back substitution*: eliminar todos os elementos por cima da diagonal da coluna i, subtraindo cada linha por cima da linha pivô por um múltiplo da mesma. Depois, decrementa-se i e repete-se este passo até que todos os elementos por cima da diagonal sejam zero.
- *Normalização*: dividindo cada linha i pelo pivô, o elemento da diagonal assume o valor de 1 e os restantes elementos são dimensionados. Assim, o componente esquerdo é a matriz identidade e o direito se torna a matriz inversa.

Analisando os passos do algoritmo pode-se observar que no mínimo o mesmo precisa pelo menos uma comparação. Assim, para uma matriz $n \times n$ o número de comparações necessárias é dado pela equação 4.4.

$$(n-1) \times 2 \tag{4.4}$$

4.2.1.2 Algoritmo GJ-Pivô_B

A seguir é apresentada a diferença do algoritmo GJ-Pivô_B em relação ao algoritmo GJ-Pivô_A.

- *Lider parcial*: localizar o pivô, que corresponde ao maior elemento da coluna i, considerando os elementos nas linhas i até n. Assim, é permutada a linha i com a linha que contém o pivô.

O resto dos passos são idênticos ao do algoritmo explicado na subsecão . Adicionalmente, o algoritmo GJ-Pivô_B necessariamente avalia qual dos elementos por baixo da coluna j é o maior, para logo armazenar o mesmo na linha i em relação ao pivô, o que significa que para uma matriz $n \times n$ o número de comparações que efetuara esta dado pela equação 4.5.

$$(n-1) \times n \tag{4.5}$$

4.2.2 Arquitetura reconfigurável da inversão da matriz

Os elementos da matriz são representados usando o sistema padrão de ponto flutuante IEEE-754. Os blocos de memória RAM internos do dispositivo escolhido (*Virtex-5*) são usados para armazenar os valores da matriz. Estes valores da matriz são extraídos dado uma ordem para achar o elemento Pivô, em que este elemento será o maior número de toda a coluna da matriz.

Na figura 4.8 pode-se observar dois tipos de arquiteturas propostas para resolver a inversão da matriz, em que a primeira, chamada de Inv_GJ-A, é uma arquitetura completamente sequencial, a qual não aproveita as capacidades de paralelismo das *FPGA*. No entanto a segunda arquitetura Inv_GJ-B apresenta modificações para resolver o problema abordando um enfoque concorrente, usando ao máximo as capacidades presentes na *FPGA*. Contudo, cada arquitetura faz uso das mesmas unidades chamadas: (a) *Pivô*, (b) *Trocador de linhas*, (c) *Eliminação de Matriz* e (d) *Normalização*. O uso destas unidades é controlado por uma FSM dentro do bloco chamado de *Inv_ GJ*, que gera os sinais necessários dentro de cada estado do processo de inversão, dando origem às diferentes arquiteturas anteriormente citadas.

Observa-se que as arquiteturas começam com uma etapa de inicialização na qual são atribuídos os parâmetros básicos em cada unidade. Na etapa seguinte encontram-se duas unidades que são *Pivô*, no qual é achado o valor do pivô e *Trocador de linhas*, em que é feita uma permutação de linhas deixando sempre o pivô na correspondente linha i da coluna j a zerar. Adicionalmente, é achado o correspondente valor de pivotamento X (equação 4.6),

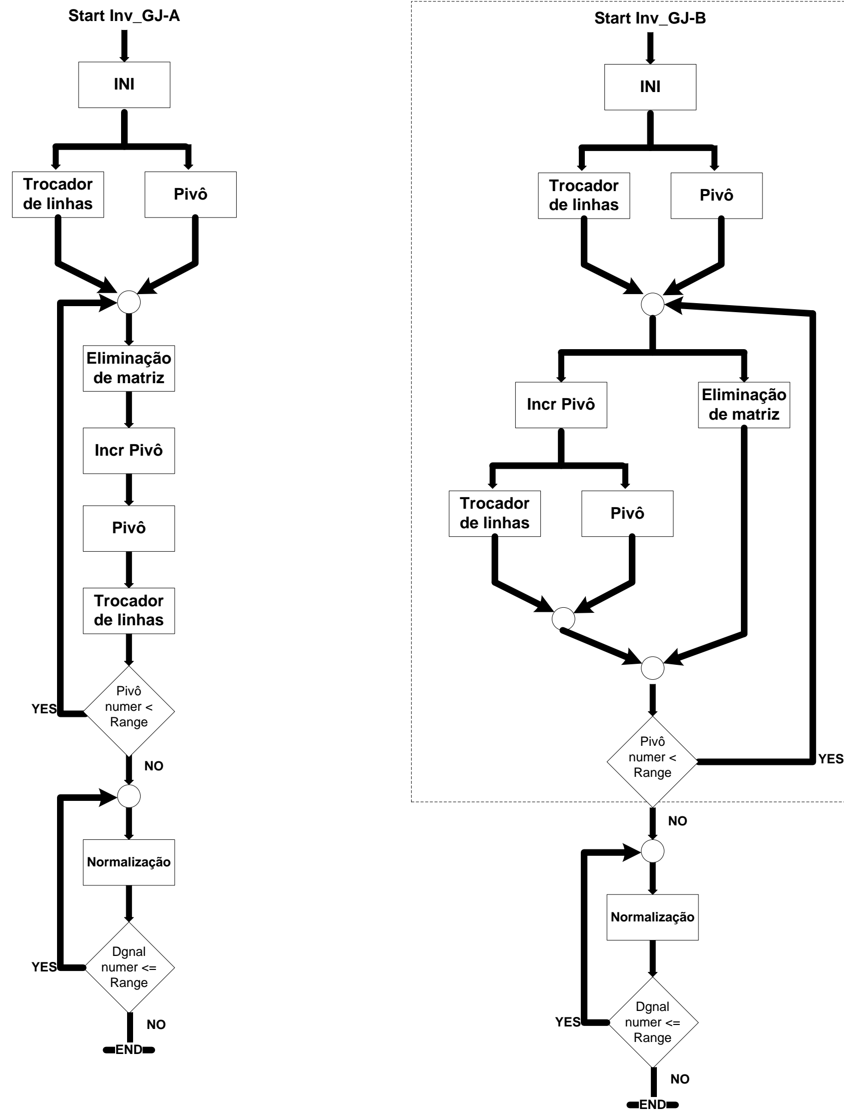

Figura 4.8. Estrutura das arquiteturas propostas. A seção dentro do quadro na arquitetura Inv_GJ-B foi modificado de [Duarte Horácio Neto 2009]

$$X = \frac{A(i,p)}{A(p,p)} \tag{4.6}$$

em que $A(p,p)$ é o elemento pivô e $A(i,p)$ é o elemento a baixo do elemento pivô.

A seguinte etapa contém a unidade *Eliminação de Matriz*, responsável por executar a equação 4.7,

$$A(i,j) = A(i,j) - A(p,j) * X \tag{4.7}$$

em que o novo valor de $A(i,j)$ corresponde à subtração do elemento a baixo do pivô com o elemento pivô multiplicado pelo pivotamento achado, alcançando o pivotamento parcial. Na

parte direita da matriz aumentada são feitos os mesmos cálculos anteriormente descritos, ou seja é realizada a seguinte equação 4.8 em paralelo, na qual usamos o mesmo valor encontrado do pivotamento.

$$I(i, j) = I(i, j) - I(p, j) * X \tag{4.8}$$

Observa-se que é usado um laço de repetição que vai se incrementando enquanto são zerados os elementos de cada coluna a baixo do elemento pivô. Adicionalmente, esta unidade *Eliminação de Matriz* tem a tarefa de zerar os elementos por cima da diagonal. Assim, o início do processo consiste em alcançar uma matriz triangular superior para logo eliminar os elementos restantes diferentes de zero da diagonal, e executar as mesmas equações 4.7 e 4.8 obtendo, no final, uma matriz diagonal no lado esquerdo e, no lado direito da matriz aumentada, uma matriz diferente da matriz identidade chamada de matriz resultante. Tanto as operações de soma/subtração quanto as operações de multiplicação ou divisão são desenvolvidas pelas unidades de ponto flutuante explicadas na seção 4.1.

O número de multiplicações necessárias executadas na unidade de *Eliminação de Matriz* é aumenta com o tamanho da matriz. Este valor pode ser calculado segundo a equação 4.9, no qual n é o tamanho da matriz.

$$\sum_{i=1}^{n-1}(i+1)i + \sum_{i=1}^{n-1} i + 2 \times n \times \sum_{i=1}^{n-1} i + n^2 \tag{4.9}$$

Quando o laço de repetição termina, a arquitetura passa à quarta etapa, em que se encontra a unidade *Normalização*. Esta unidade realiza a normalização dos elementos da diagonal seguindo a equação 4.10, o que significa que primeiro é achado o valor recíproco R (equação 4.11), para logo executar a equação 4.10.

$$A(p, j) = A(p, j) \times R \tag{4.10}$$

$$R = \frac{A(p, i)}{A(p, p)} \tag{4.11}$$

A figura 4.9 apresenta a sequência de passos para realizar a Eliminação da Matriz, na qual existem dois laços de repetição que dependem do valor da coluna e da linha para chegar a cada elemento na matriz, e obter tanto a matriz triangular superior em um primeiro passo como a matriz diagonal no final.

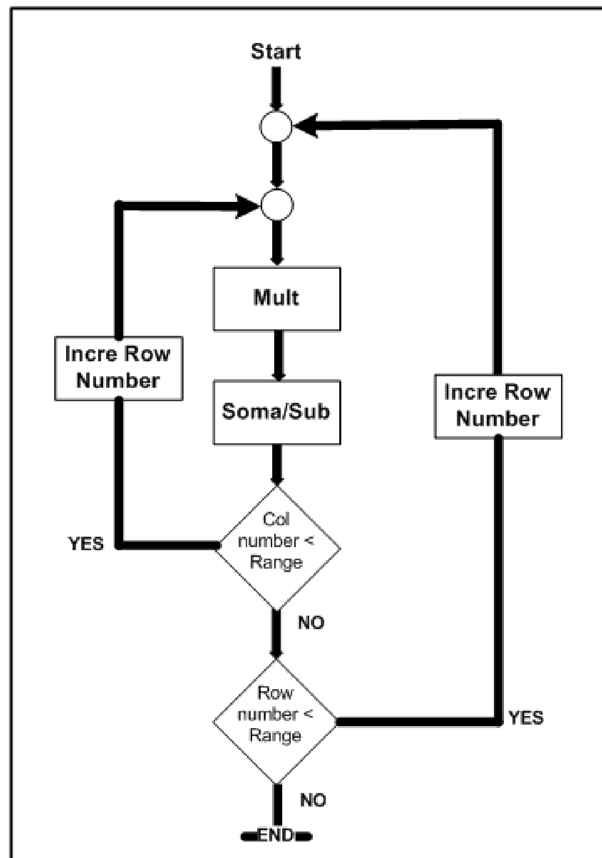

Figura 4.9. Estrutura sequencial da unidade de eliminação matricial (modificado de [Duarte Horácio Neto 2009])

A unidade de normalização no final do processo, tem como objetivo diminuir o erro gerado pela unidade de multiplicação, que como foi mostrado na seção 4.1.2 é de 10^{-7}. Adicionalmente, usando esta unidade no final, serão feitas unicamente multiplicações na parte direita da matriz aumentada para obter a matriz inversa, enquanto se for usada em outro lugar diferente do final, seria necessário realizar cálculos nas duas partes da matriz aumentada, fato que propagaria o erro gerado pela unidade de multiplicação.

Na figura 4.10 está indicada a sequência de passos para realizar a normalização, na qual encontra-se um laço de repetição que depende do valor da coluna, quando este valor é alcançado, o processo termina.

4.2.3 O fluxo de dados desde/para o bloco Inver_GJ

O bloco *Inv_ GJ* tem o trabalho de controlar os acessos à memória RAM interna, assim como os acessos às diferentes unidades aritméticas de ponto flutuante, e distribuir estes dados para cada unidade que precise. A figura 4.11 mostra o fluxo de dados desde/para o bloco Inver_GJ, que contém a unidade de controle *Gauss-Jordan*, a qual é uma FSM de tipo *Mealy*. A figura mostra

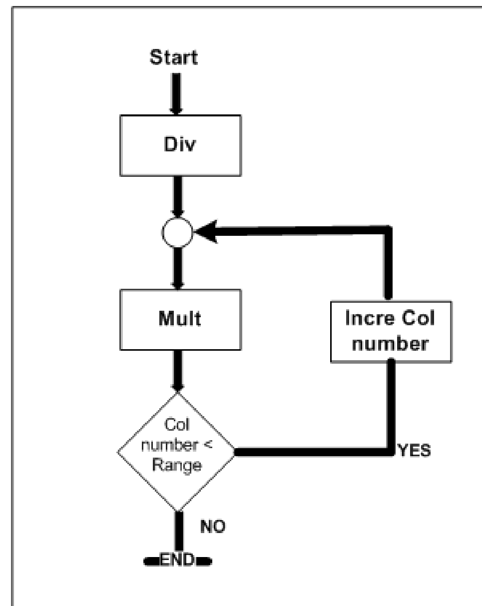

Figura 4.10. Estrutura sequencial da unidade de normalização (modificado de [Duarte Horácio Neto 2009])

os sinais *Dato_ out* e *Dato_ in*, com comprimento da palavra de 1152 bits, desde/para a memória RAM. Esta opção de se ter um comprimento da palavra de até 1152 bits nas memórias internas da FPGA *Virtex-5* é bem aproveitada e faz parte das diferentes arquiteturas apresentadas. Por um lado, a arquitetura *Inver_ GJ-A* implementa um comprimento da palavra de apenas 32 bits ou 64 bits correspondentes ao valor de um número só, entanto na arquitetura *Inver_ GJ-B*, é usado exatamente o valor mostrado para cada sinal (figura 4.11), ou seja 1152 bits de comprimento da palavra. Neste caso, é possível armazenar até 36 ou 18 números em cada posição da memória se for para 32 bits ou 64 bits respectivamente.

Esta diferença no comprimento da palavra permite implementações como *pipeline* ou diferentes arquiteturas concorrentes. No entanto os valores enviados desde/para cada uma das operações em ponto flutuante são de 32 bits para a arquitetura *Inver_ GJ-A* e de 160 bits para a arquitetura *Inver_ GJ-B*, como mostrado na figura 4.11 (64 bits até 320 bits quando se usa precisão dupla). Nesta caso, cada resultado destas operações volta para a unidade de controle, que define o correspondente uso.

O bloco *Inver_ GJ* também controla os acessos em uma outra memória *RAM_ I*, que contém os valores da matriz identidade, e que no final do processo armazenará os valores da matriz invertida como mostra a figura 4.12. Adicionalmente, a mesma controla os sinais que enviam os valores para cada operação em ponto flutuante e os resultados são também retornados para a unidade de controle.

Tal como mostrado na figura 4.13, a unidade de controle *Gauss-Jordan*, determina o começo e o fim do processo de inversão, controlando assim as restantes unidades encarregadas de obter

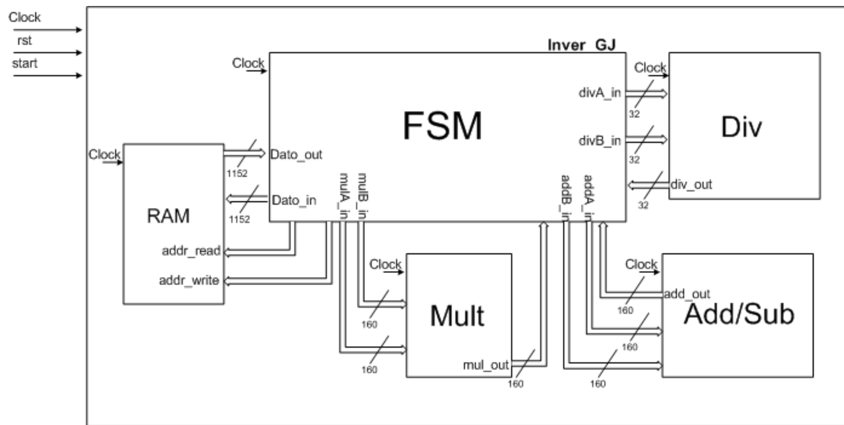

Figura 4.11. Diagrama de bloco do módulo de inversão GJ para a matriz A

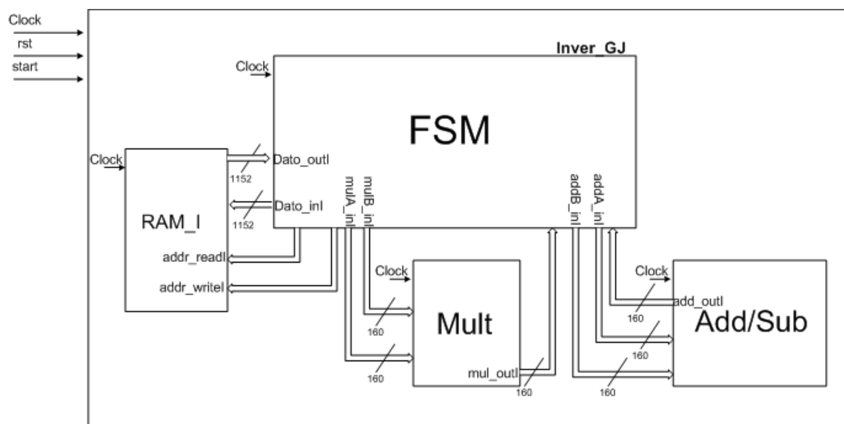

Figura 4.12. Diagrama de bloco do módulo de inversão GJ para a matriz I

a inversão da matriz. A unidade de controle *Gauss-Jordan* calcula os valores de i e j, e por sua vez envia as outras unidades permitindo o controle dos endereços na matriz.

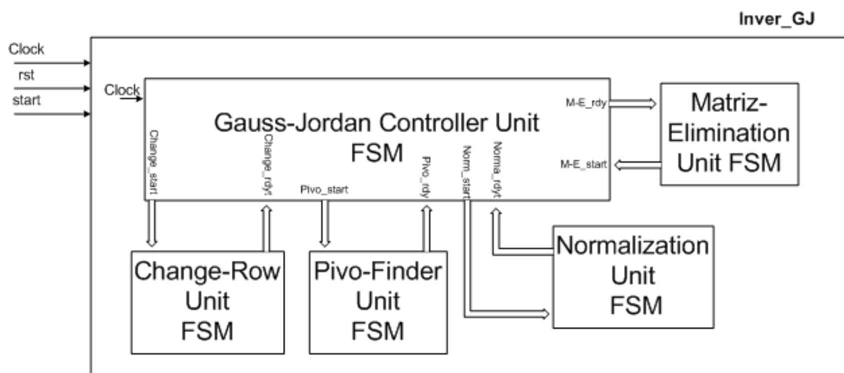

Figura 4.13. Controle interno no bloco Inver_GJ

4.2.4 Unidade Trocador de linhas

A unidade de *Trocador de linhas* foi implementada em *hardware* de 3 formas: (a) *Trocador de linhas-01*, (b) *Trocador de linhas-02* e (c) *Trocador de linhas-03*.

A seguir são descritas as 3 formas de implementação:

- *Trocador de linhas-01*: consiste na permutação de linhas elemento por elemento, armazenando na memória RAM, logo depois é feita a comparação a fim de encontrar o maior elemento correspondente na coluna j da linha i avaliada. Esta permutação foi feita fisicamente, ou seja, acessando na memória RAM, e escrevendo na mesma a cada vez segundo a quantidade de elementos na matriz.

- *Trocador de linhas-02*: explora as vantagens de se ter um comprimento da palavra da quantidade de elementos armazenados em cada linha. Esta implementação reduz o tempo total do processo de inversão, devido ao fato que a permutação de linhas é feita de uma vez só no momento em que acha o maior elemento da coluna, permitindo assim a escrita na memória RAM de todos os elementos da linha na hora.

- *Trocador de linhas-03*: faz uso de um registro chamado *pos_memoria* com um comprimento da palavra correspondente ao número de linhas usando 6 bits para representar cada posição, uma vez que 36 posições representam o número de linhas da maior matriz invertida. Na figura 4.14 se mostra a interpretação na qual é atribuído um valor a cada 6 bits que corresponde a cada posição da linha. Por exemplo, se o número atribuído como é apresentado na figura 4.14 na posição 1 corresponde ao valor de zero, indicado pelos 6 bits, significa que o valor da primeira linha da matriz estará armazenado na posição zero da memória RAM. O que finalmente atribui os valores de cada posição das linhas da matriz no registro *pos_memoria*, sem passar por um armazenamento físico ou ordenamento na memória RAM evitando assim escrever na mesma.

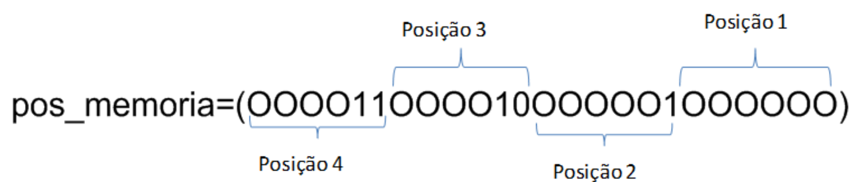

Figura 4.14. Registro pos_memoria

Assim, o tempo de execução (em ciclos de relógio) necessário para obter a permutação das linhas segundo as unidades *Trocador de linhas-01*, *Trocador de linhas-02* e *Trocador de linhas-03*

está descrito pelas equações 4.12, 4.13 e 4.14 respectivamente, em que as mesmas correspondem ao número de ciclos mínimo que tarda cada unidade em executar a tarefa.

$$10 * n \sum_{1}^{n-1} K \tag{4.12}$$

$$10 * n * (n - 1) \tag{4.13}$$

$$n * (n - 1) \tag{4.14}$$

Parte do código desenvolvido para a unidade de *Trocador de linhas-03* é apresentada na figura 4.15, na qual se observam nas linhas 598, 607, 620, 636 e 639 os estados: (a) *Change_ Row_ começa*, (b) *Change_ Row_ pausa*, (c) *Change_ Row_ Exchange*, (d) *Change_ Row_ Troca_fila* e (e) *Change_ Row_ volta* respectivamente.

A FSM tipo Mealy mostrada na figura 4.16 ilustra estes 5 estados mencionados os quais são descritos a continuação:

- *Change_ Row_ começa*: aguarda até *change_ start* ter o valor de 1, assim inicia o processo atribuindo o valor de *N* à *Fila* e *N_ bits* à *Fila_ principal*, para depois passar para o estado seguinte.
- *Change_ Row_ pausa*: espera até *ram_ rdy_ ler* ter o valor de 1, para atribuir o valor armazenado em *Salida* em um registro auxiliar chamado de *aux_ salida*, mas, incrementando o valor de *Fila* em 1 antes de passar para o estado *Change_ Row_ Exchange*.
- *Change_ Row_ Exchange*: espera até *ram_ rdy_ ler* ter o valor de 1 e seguidamente avaliar se o número na linha *i* e coluna *j* é maior do que o número na linha *i+1* coluna *j*. Se for verdadeiro o seguinte estado será *Change_ Row_ Troca_fila*, ou caso contrario o estado será *Change_ Row_ volta*. Neste mesmo estado é atribuído o valor de *Fila_ secundaria* no registro *pos_ memoria*.
- *Change_ Row_ Troca_fila*: neste estado é colocado o valor *Fila_ principal* no registro *pos_ memoria* passando seguidamente para o estado *Change_ Row_ volta*.
- *Change_ Row_ volta*: avaliamos se todas as linhas por baixo da coluna *j* foram comparadas, para assim dar fim no processo.

```
595 ------------ Change_Row: troca as filas e deixa o maior numero na posição da diagonal evaluada ----------------------------------------
596 --------------------------------------------------------------------------------------------------------------------------------------
597     CASE Change_Row IS
598         WHEN Change_Row_comeca=>
599              change_rdy<='0';
600              IF change_start='1' THEN
601               FILA<=N;
602               FILA_principal<=N_bits;
603               Change_Row_nx_state<=Change_Row_pausa;
604              ELSE
605               Change_Row_nx_state<=Change_Row_comeca;
606              END IF;
607         WHEN Change_Row_pausa=>
608              IF ram_rdy_ler='1' THEN
609               aux_salidaA<=salidaA;
610               aux_salidaI<=salidaI;
611               FILA_secundaria<=FILA_principal+1;
612               FILA<=FILA+1;
613               Change_Row_nx_state<=Change_Row_Exchange;
614               ram_action<="00";
615              ELSE
616               FILA<=N;
617               ram_action<="01";
618               Change_Row_nx_state<=Change_Row_pausa;
619              END IF;
620         WHEN Change_Row_Exchange=>
621              IF ram_rdy_ler='1' THEN
622                 IF aux_salidaA((32*N)+31 downto (32*N))<salidaA((32*N)+31 downto (32*N)) THEN
623                    pos_memoria((6*N)+5 downto (6*N))<=FILA_secundaria;
624                    aux_salidaA<=salidaA;
625                    aux_salidaI<=salidaI;
626                    Change_Row_nx_state<=Change_Row_Troca_fila;
627                    ram_action<="00";
628                 ELSE
629                    ram_action<="00";
630                    Change_Row_nx_state<=Change_Row_volta;
631                 END IF;
632              ELSE
633               ram_action<="01";
634               Change_Row_nx_state<=Change_Row_Exchange;
635              END IF;
636         WHEN Change_Row_Troca_fila=>
637               pos_memoria((6*(FILA))+5 downto (6*(FILA)))<=FILA_principal;
638               Change_Row_nx_state<=Change_Row_volta;
639         WHEN Change_Row_volta=>
640               IF FILA<rango THEN
641                 FILA<=FILA+1;
642                 FILA_principal<=FILA_principal+1;
643                 FILA_secundaria<=FILA_secundaria+1;
644                 Change_Row_nx_state<=Change_Row_Exchange;
645                 change_rdy<='0';
646               ELSE
647                 Change_Row_nx_state<=Change_Row_comeca;
648                 change_rdy<='1';
649               END IF;
650         WHEN OTHERS =>
651               NULL;
```

Figura 4.15. Código em VHDL da Unidade Trocador de linhas

4.2.5 Sobre a memoria RAM

Dois tipos de memórias RAM foram usadas nas arquiteturas desenvolvidas: (a) memória *True_ Dual-Port_ RAM* (figura 4.17) e (b) memória *Simple_ Dual-Port_ RAM* (ver figura 4.18). A primeira foi usada para implementar a arquitetura *Inver_ GJ-A*, entanto que a segunda foi usada na implementação da arquitetura *Inver_ GJ-B*.

A memória tipo *True_ Dual-Port_ RAM* tem duas portas, *a* e *b* como se mostra na figura 4.17. O acesso a leitura/escrita na memória é permitido em cada porta. A memória tipo *Simple_ Dual-Port_ RAM* possui também duas portas, *a* e *b* (figura 4.18). O acesso à escrita na memória é permitido pela porta *a*, entanto que o acesso à leitura é permitido pela porta *b* [LogiCORE IP Block Memory Generator v4.2 2010].

Primeiramente foi abordado o acesso à memória *True_ Dual-Port_ RAM*, com o uso da arquitetura *Inver_ GJ-A*, a qual teve como objetivo transformar os endereços matriciais em endereços vetoriais tal como mostrado na figura 4.19. Para armazenar os elementos da matriz em um

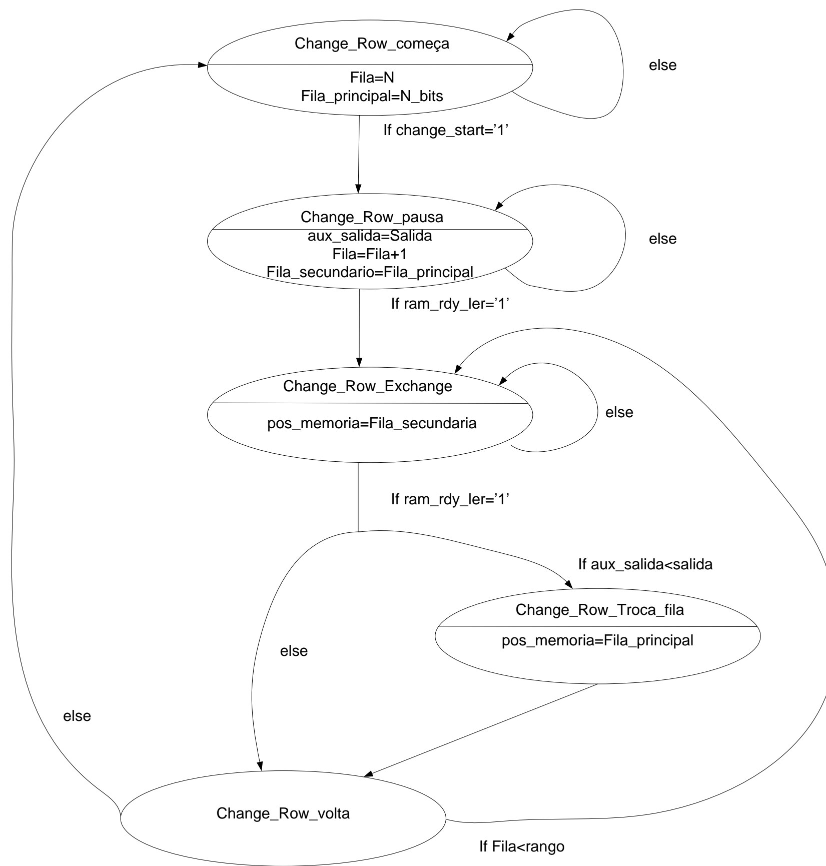

Figura 4.16. FSM da unidade Trocador de linhas

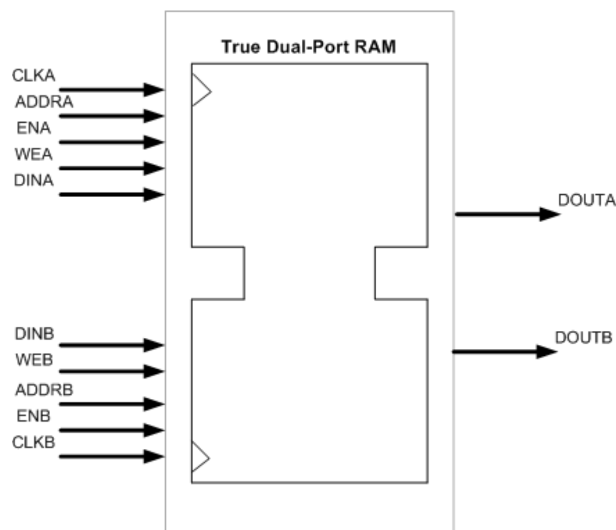

Figura 4.17. Memória True Dual-Port RAM (Obtido de [LogiCORE IP Block Memory Generator v4.2 2010])

vetor, sem perder a posição de cada elemento dentro deste vetor, foi executada a equação 4.15. Onde, cada par de valores de i e j gera uma posição no vetor que será a forma de armazenar os valores na memória RAM. Nesta equação 4.15 i e j são os valores das posições matriciais e

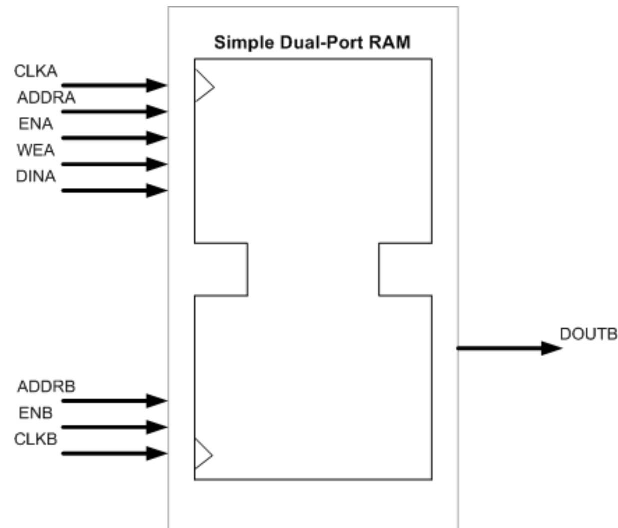

Figura 4.18. Memória Simple Dual-Port RAM (Obtido de [LogiCORE IP Block Memory Generator v4.2 2010])

range corresponde ao tamanho da matriz.

$$Pos = (i \times range) + j \tag{4.15}$$

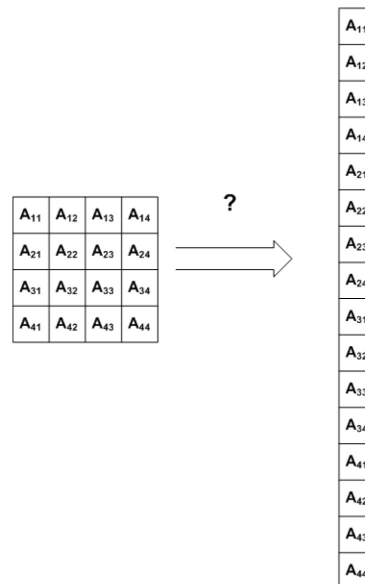

Figura 4.19. Transformação dos endereços matriciais em endereços vetoriais

No entanto, com este tipo de acesso não é possível a leitura/escrita na memória RAM para gerar implementações em *hardware* de *pipeline* ou arquiteturas concorrentes. Uma outra solução foi implementada usando a memória *Simple_ Dual-Port_ RAM* na arquitetura *Inver_ GJ-B*.

Neste caso, a finalidade foi concatenar os valores de toda uma linha da matriz, armazenando assim os valores em uma posição só dentro da memória RAM, tal como mostrado na figura 4.20,

permitindo ler/escrever toda uma linha em troca de ler/escrever n vezes para extrair os dados de toda uma linha. Esta forma de acessar aos dados da memória apresentou vantagens significativas uma vez que foi possível implementar outras arquiteturas aproveitando o poder que apresentam os FPGAs na paralelização.

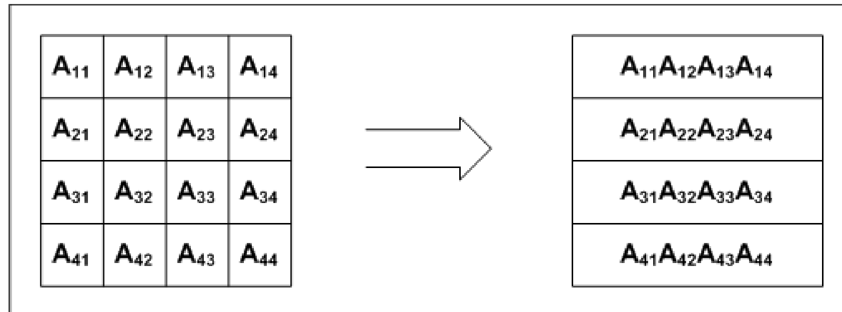

Figura 4.20. Transformação dos endereços matriciais em endereços vetoriais concatenados

Os diferentes endereços são enviados para a memória RAM através dos sinais *address_a* e *address_b*, que de forma concorrente atribui os valores nas entradas *addr_a* e *addr_b* respectivamente(figura 4.21). Por outro lado, um processo chamado de *RAM_Controller_sequential* é usado para realizar a troca de estado tal como mostrado nas linhas 327 − 336 na figura 4.21. Adicionalmente, a figura 4.22 mostra os estados de *ler* assim como *escrever* nas linhas 421 e 442, os mesmos desenvolvem as tarefas de ler da memória RAM e escrever na mesma.

```
310  ----------------------------------------------------
311  ---------------Controlador da Memória RAM---------
312  direccion_A: addr_a   <=
313  "0000000" WHEN address_a = 0 ELSE
314  "0000001" WHEN address_a = 1 ELSE
315  "0000010" WHEN address_a = 2 ELSE
316  "0000011" WHEN address_a = 3 ELSE
317
318  (OTHERS=>'0');
319
320  direccion_B: addr_b   <=
321  "0100100" WHEN address_b = 0 ELSE
322  "0100101" WHEN address_b = 1 ELSE
323  "0100110" WHEN address_b = 2 ELSE
324  "0100111" WHEN address_b = 3 ELSE
325  (OTHERS=>'0');
326  ---------- Lower section: ------------------------
327  RAM_Controller_sequential: PROCESS (clock,rst)
328   BEGIN
329     IF rst   = '1' THEN
330           ram_start<= inicio;
331           accion<= ler_escrever;
332     ELSIF (RISING_EDGE(clock)) THEN
333           ram_start<= nx_state;
334           accion<=pr_state;
335     END IF;
336  END PROCESS;
```

Figura 4.21. Código em VHDL do Controlador da memória RAM_sequencial

A FSM mostrada na figura 4.23 ilustra os estados resumidos do controlador da memória

```
420            ----------LER DA RAM-------------
421                WHEN ler =>
422                    ram_rdy_ler<='0';
423                    address_a<=FILA;
424                    address_b<=FILA;
425                    ena  <='1';
426                    enb  <='1';
427                    wea  <="0";
428                    web  <="0";
429                    pr_state<=tiempo;
430                WHEN tiempo=>
431                    pr_state<= mando;
432                WHEN mando =>
433                    ena  <='0';
434                    enb  <='0';
435                    wea  <="0";
436                    web  <="0";
437                    salidaA<=dado_ina(127 downto 0);
438                    salidaI<=dado_inb(127 downto 0);
439                    ram_rdy_ler<='1';
440                    pr_state<= ler_escrever;
441            ----------ESCREVENDO NA RAM-------------
442                WHEN escrever =>
443                    ram_rdy_esc<='0';
444                    address_a<=FILA;
445                    address_b<=FILA;
446                    ena  <='1';
447                    enb  <='1';
448                    wea  <="1";
449                    web  <="1";
450                    pr_state<=receber;
451                WHEN receber =>
452                    ena  <='1';
453                    enb  <='1';
454                    wea  <="1";
455                    web  <="1";
456                    dado_outa(127 downto 0)<=entradaA;
457                    dado_outb(127 downto 0)<=entradaI;
458                    ram_rdy_esc<='1';
459                    pr_state<= tiempo_escr;
460                WHEN tiempo_escr=>
461                    ena  <='0';
462                    enb  <='0';
463                    wea  <="0";
464                    web  <="0";
465                    pr_state<= ler_escrever;
466                WHEN OTHERS =>
467                    NULL;
```

Figura 4.22. Código em VHDL do Controlador da memória RAM_combinacional

RAM, em que se tem 4: (a) *inicio*, (b) *ler_escrever*, (c) *ler* e (d) *escrever*.

A seguir são descritos os 4 estados da FSM:

- *inicio*: aguarda até que *start* alcance o valor de 1 para assim passar para o próximo estado.
- *ler_escrever*: revisa o valor de *ram_action* e se for "01" passa para o estado *ler*, ou se for "10" passa para o estado *escrever*, ao contrario aguarda nesse estado.
- *ler*: lê a saída da memória RAM *dado_in* e atribui este valor no registro *Salida*, assim como atribui o valor de 1 em *Ram_rdy_ler*.
- *escrever*: escreve na memória RAM através de *Dado_out* o valor de *Entrada*, assim como atribui o valor de 1 em *Ram_rdy_esc*.

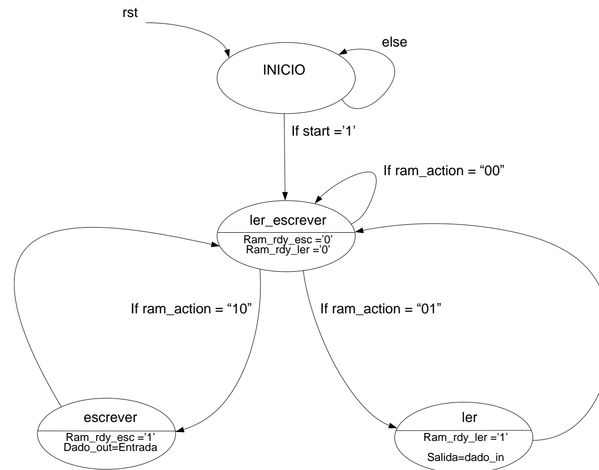

Figura 4.23. FSM do controlador da memória RAM

4.2.6 O *pipeline*

Fazendo uso da vantagem que representa a possibilidade de armazenar na memória palavras com um tamanho de até 1152 bits, foi gerada uma arquitetura que implementou um *pipeline*, acelerando assim o processo da inversão da matriz. Na figura 4.24, apresenta-se como este *pipeline* funciona.

O *pipeline* inicia com uma multiplicação, e logo depois em cada estado é feita uma outra multiplicação e uma soma/subtração. O resultado da multiplicação é uma das entradas da soma/subtração no seguinte estado, até alcançar o último elemento da linha finalizando com uma soma/subtração, este processo se repete até passar por todas as linhas da matriz. O *pipeline* esta implementado apenas na arquitetura *Inver_ GJ-B* proposta, e é introduzido na unidade de *Eliminação de Matriz*.

Com essa estrutura configurada como um *pipeline*, após um período inicial de latência (tempo para a ocupação de todos os estágios do Pipeline) de dois ciclos de relógio, o desempenho de saída seria de apenas um ciclo de relógio por elemento zerado da matriz, ou seja, a cada ciclo de relógio seria disponibilizado um elemento zerado da linha da matriz.

4.2.7 O paralelismo

O paralelismo é uma das principais vantagens encontradas nos FPGAs, porém apresenta um problema no acesso aos dados armazenados na memória RAM. Este problema foi resolvido aproveitando a flexibilidade de se ter um comprimento da palavra de 1152 bits, como foi explicado em subseções anteriores. Assim, facilmente tem-se uma grande quantidade de números para serem usados, e desta forma foi implementada uma arquitetura concorrente (figura 4.25).

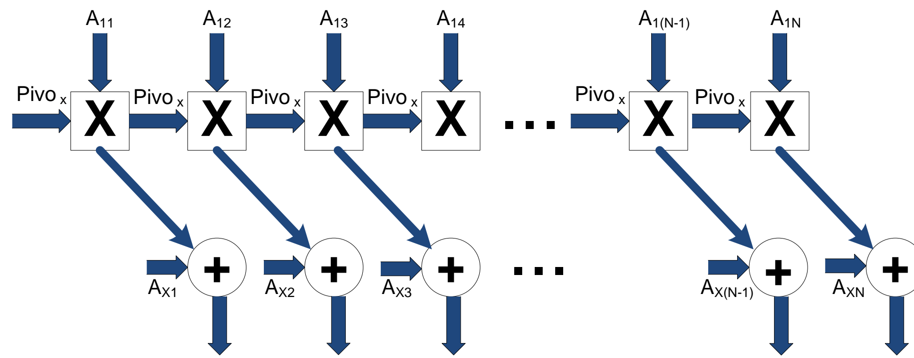

Figura 4.24. *Pipeline* implementado em *hardware*

Esta arquitetura usa cinco multiplicadores em linha executados todos no mesmo estado, e no seguinte estado é feita a soma/subtração. Logo depois, é deslocado o apontador 5 posições até passar por todos os elementos da linha, sendo que a cada passo são avaliados o número de posições faltantes, deslocando essa quantidade de multiplicadores e somadores/subtração. Ou seja, para uma matriz de $36x36$, o último elemento da linha será deslocado apenas de um multiplicador assim como de um somador/subtrator. Na figura 4.25 é apresentada a arquitetura implementada, descrevendo o comportamento com a primeira linha da matriz. A implementação foi feita apenas na arquitetura *Inver_ GJ-B* proposta senda introduzida na unidade de *Eliminação de Matriz* e *Normalização*.

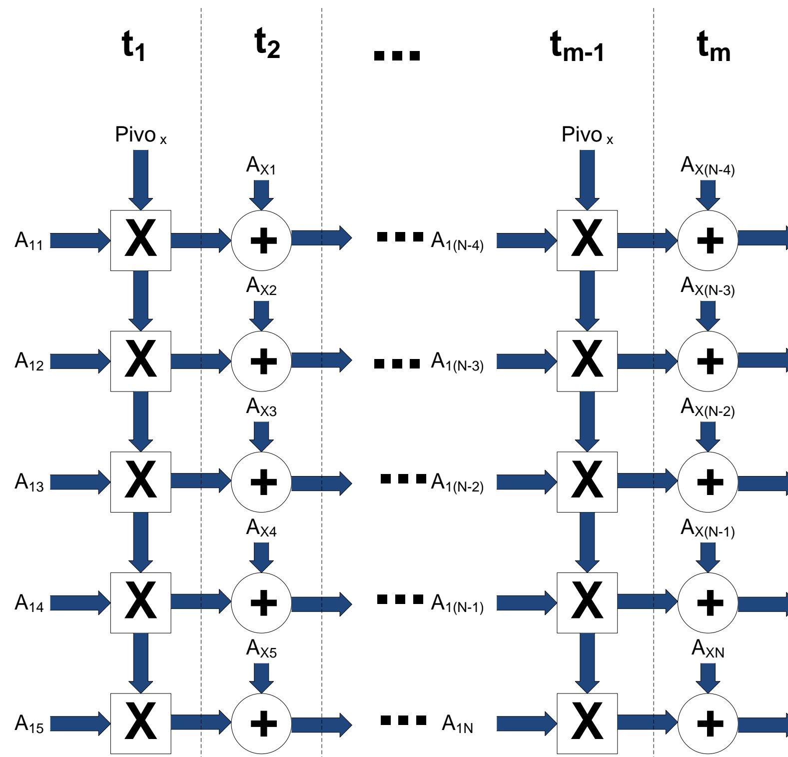

Figura 4.25. Implementação em *hardware* dos multiplicadores e somadores em paralelo

4.3 Etapas no desenvolvimento do Projeto

Por último, a sequência de etapas seguida para desenvolver o projeto de inversão de matrizes, sintetizando o descrito nas seções acima, começa com a modificação do algoritmo Gauss-Jordan, avaliando a precisão no cálculo. A seguir foram realizadas modificações na unidade de *Trocador de linhas* visando melhorar o desempenho, assim como modificações na unidade de controle de acesso à memória RAM, visando minimizar o gargalo de von Neumann. Essas modificações permitiram usar métodos como o *pipeline* e o paralelismo nas unidades de *Eliminação de Matriz* e *Normalização* (figura 4.26).

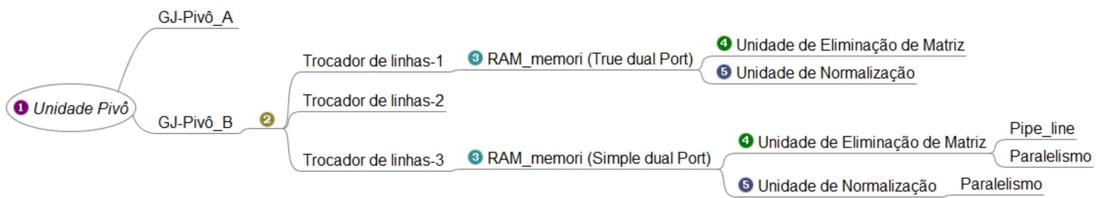

Figura 4.26. Caminho seguido para obter a melhor solução em *hardware* para inverter matrizes

4.4 Conclusão do Capítulo

As diferentes arquiteturas desenvolvidas para inverter matrizes de diferentes tamanhos, usando diferentes comprimentos da palavra na representação numérica foram apresentadas neste capítulo. Diferentes modificações das arquiteturas implementadas permitem realizar uma análise do *tradeoff* entre diversas soluções alternativas. Adicionalmente durante o capítulo, foram mostradas as características de cada unidade desenvolvida, encarregadas das diferentes etapas do processo de inversão de matrizes, as quais evoluíram durante o desenvolvimento do projeto aproveitando as capacidades do FPGA usado.

Foram apresentadas também as modificações feitas em cada unidade com o objetivo de esclarecer as diferentes características das arquiteturas e suas correspondentes vantagens, assim como algumas partes do código desenvolvido em *VHDL* na implementação.

As diferentes configurações de acesso à memória RAM (internas do dispositivo FPGA usado), permitiram realizar modificações melhorando as arquiteturas propostas, modificações tais como o uso de *pipeline* assim como a inclusão de operações concorrentes, reduzindo o comumente conhecido *gargalo de von Neumann.*

5 Resultados

Este trabalho abordou a implementação do algoritmo de redução de Gauss-Jordan para inverter matrizes, usando representação em ponto flutuante de precisão simples e dupla, com o objetivo de gerar uma biblioteca parametrizável tanto pelo tamanho da matriz quanto pelo comprimento da palavra. Esta biblioteca pode ser usada em diferentes aplicações, por exemplo, em reconhecimento de voz, aplicações de comunicações sem fio e robótica, entre outros, nos quais são necessários cálculos com alta precisão e alto desempenho de matrizes de grande tamanho.

Os resultados das duas diferentes arquiteturas implementadas do Algoritmo Gauss-Jordan, assim como as comparações de cada modificação feita nas diferentes unidades desenvolvidas são apresentadas. As arquiteturas são mostradas separadamente para, primeiramente, obter uma análise individual de cada uma e, ao final, realizar uma comparação entre as arquiteturas. São apresentadas simulações em que se observa a precisão dos resultados calculados, assim como o tempo de execução para calcular os mesmos. Para medir a precisão nos cálculos são comparados os resultados obtidos com o *software* MATLAB como estimador estático, e os resultados obtidos da simulação das arquiteturas implementados em *VHDL*, calculando o erro médio (ME- *Mean Error*) e o erro quadrático médio (MSE - *Mean Square Error*). Adicionalmente, são apresentados resultados da síntese lógica do algoritmo implementado, realizando uma estimativa do consumo de recursos da arquitetura.

As arquiteturas *hardware* propostas foram desenvolvidas em uma linguagem de descrição de *hardware* (*VHDL*) usando como software de desenvolvimento o *Integrated Software Environment(ISE)* 10.1 da *Xilinx*. O *FPGA* usado foi o *Virtex-5 XC5VLX110T*.

5.1 Algoritmos GJ-Pivô_A e GJ-Pivô_B

Utilizando precisão simples com diferentes tamanhos de matrizes quadradas entre o intervalo de $n = 4$ até $n = 36$, usando 100 amostras aleatórias gerados no MatLab entre o intervalo de [0,1], para cada valor de n, foi feita uma comparação entre o erro médio(ME) (figura 5.1) do algoritmo GJ-Pivô_A e o erro médio(ME) (figura5.3) do algoritmo GJ-Pivô_B de cada elemento da matriz. Mostrou-se que o erro gerado pelo algoritmo GJ-Pivô_B é menor cem vezes (10^{-3})

do que o erro gerado pelo algoritmo GJ-Pivô_A (10^{-2}), para uma matriz com $n = 36$. Por outro lado, o erro médio quadrático (MSE) confirma que a resposta gerada pelo algoritmo GJ-Pivô_B (figura 5.4) tem um erro menor do que a resposta do algoritmo GJ-Pivô_A (figura 5.2).

Adicionalmente, foram feitas comparações dos erros usando precisão dupla dos algoritmos anteriormente mencionados. Onde, pode-se observar que tanto o (ME) (ver figura 5.5) como (MSE) (ver figura 5.6) do algoritmo *GJ-Pivô_B* apresentam um valor menor que quando comparado com o (ME) (figura 5.7) e o (MSE) (figura 5.8) do algoritmo *GJ-Pivô_A*.

Os dados de cada tamanho da matriz foram gerados usando $A = double(rand(x; x))$ em *MatLab*, que define a matriz com números reais no intervalo de [0, 1]. As variações de (ME) e (MSE) são devidas ao erro de truncamento que produz a biblioteca de multiplicação, a qual gera um erro de 10^{-7}. Adicionalmente, quando se representam números com precisão finita, não todos os números no intervalo disponível se podem expressar exatamente.

O método de regressão polinomial de ordem cinco aplicado aos resultados de erro, ilustra uma tendência do crescimento enquanto o tamanho da matriz é aumentado. As oscilações do erro sobre a curva podem ser explicadas devido ao método aleatório usado para representar os dados das matrizes, e a sua respectiva representação em ponto flutuante. Observe que a curva cresce tanto para o *ME* quanto para o *MSE*, mostrando que a tendência do erro é crescer enquanto o tamanho da matriz é aumentada, explicado pelo número de multiplicações necessárias para a inversão da matriz que traz consigo um erro de truncamento.

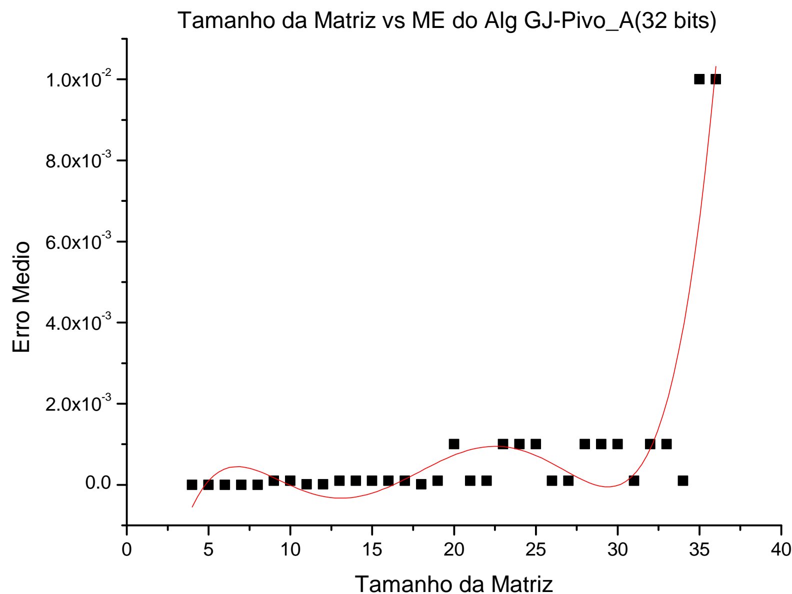

Figura 5.1. Erro médio para o algorítimo GJ-Pivô_A usando precisão simples

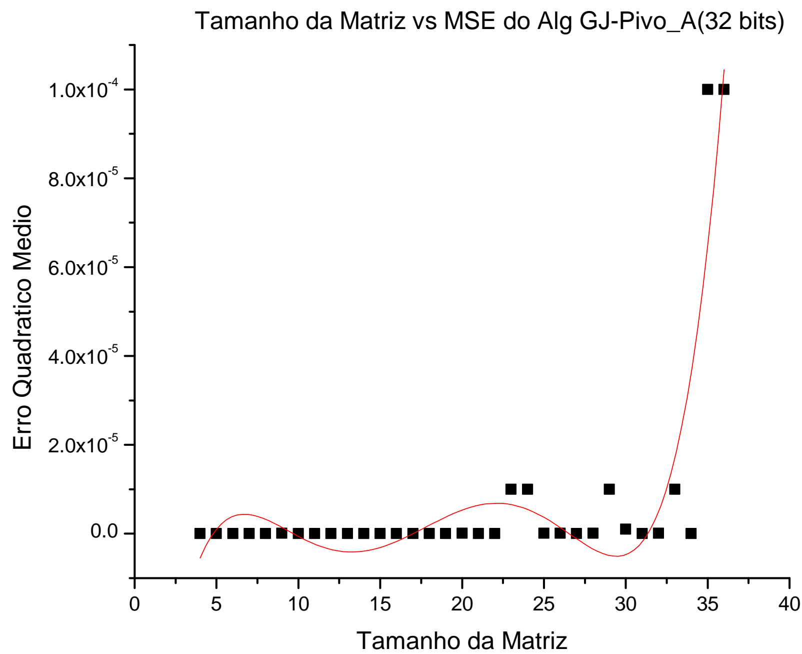

Figura 5.2. Erro quadrático médio para o algorítimo GJ-Pivô_A usando precisão simples

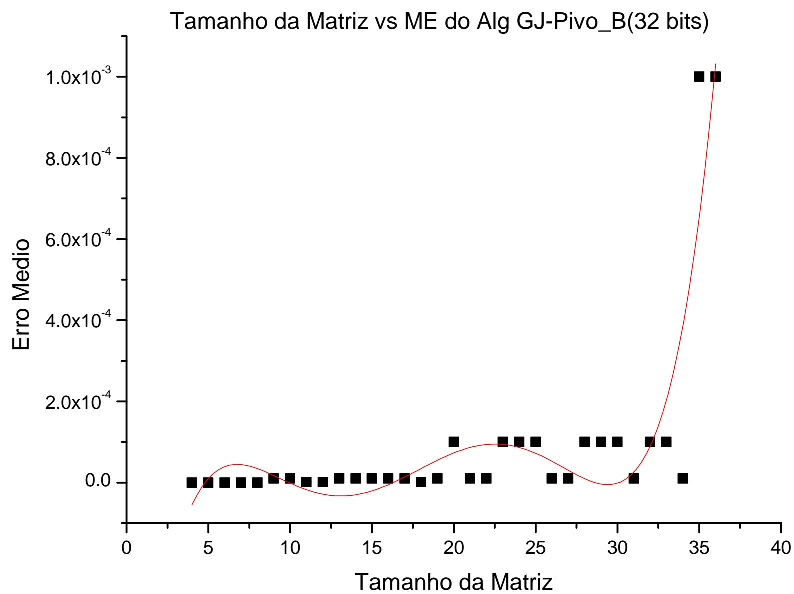

Figura 5.3. Erro médio para o algorítimo GJ-Pivô_B usando precisão simples

Para matrizes de tamanhos 20×20 ou menores a precisão mostrada é menor que 10^{-4} para o ME e de 10^{-7} para o MSE usando precisão simples, e de 10^{-8} para o ME e de 10^{-14} para o

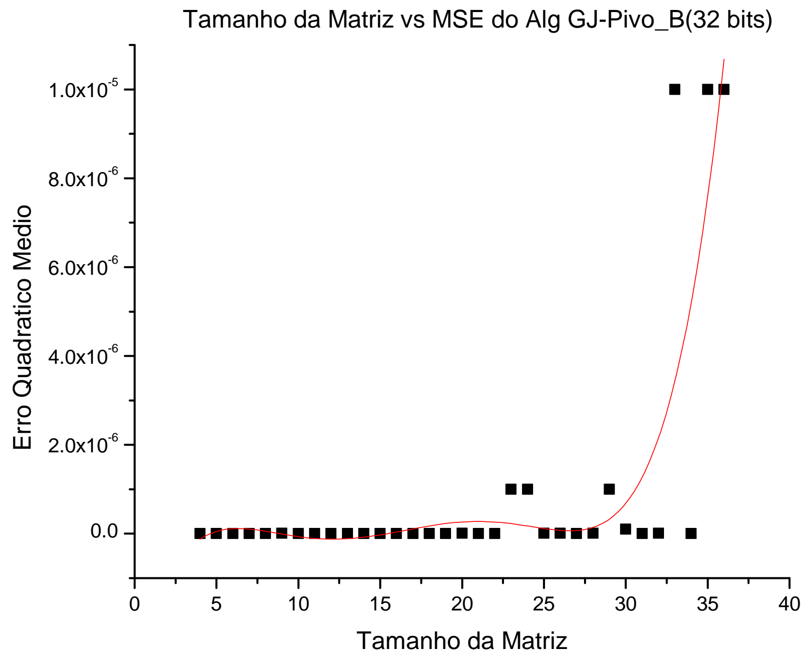

Figura 5.4. Erro quadrático médio para o algorítimo GJ-Pivô_B usando precisão simples

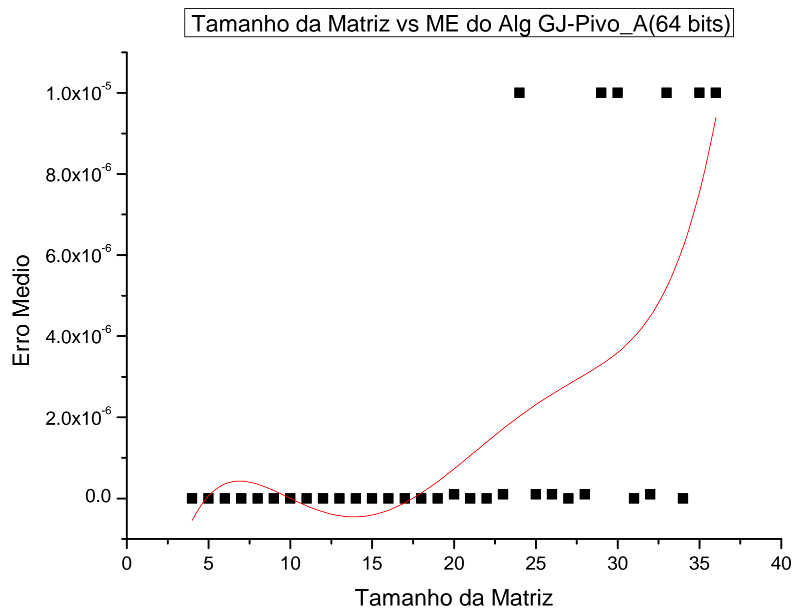

Figura 5.5. Erro médio para o algorítimo GJ-Pivô_A usando precisão dupla

MSE para precisão dupla. No entanto, o desempenho do algoritmo GJ-Pivô_A é melhor, o que é devido a que este apenas precisa avaliar se o elemento da linha i e coluna j é diferente de zero, o que implica em, no mínimo, fazer uma comparação.

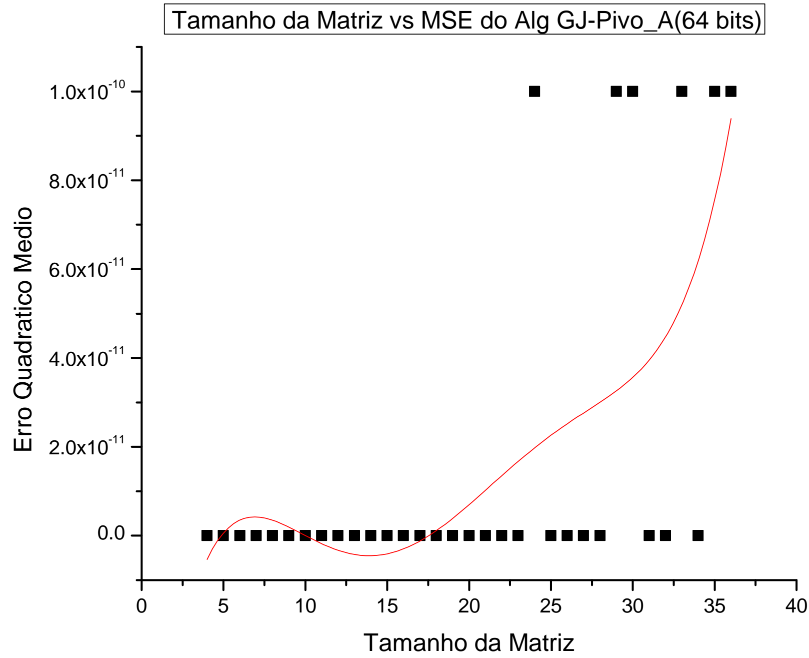

Figura 5.6. Erro quadrático médio para o algorítimo GJ-Pivô_A usando precisão dupla

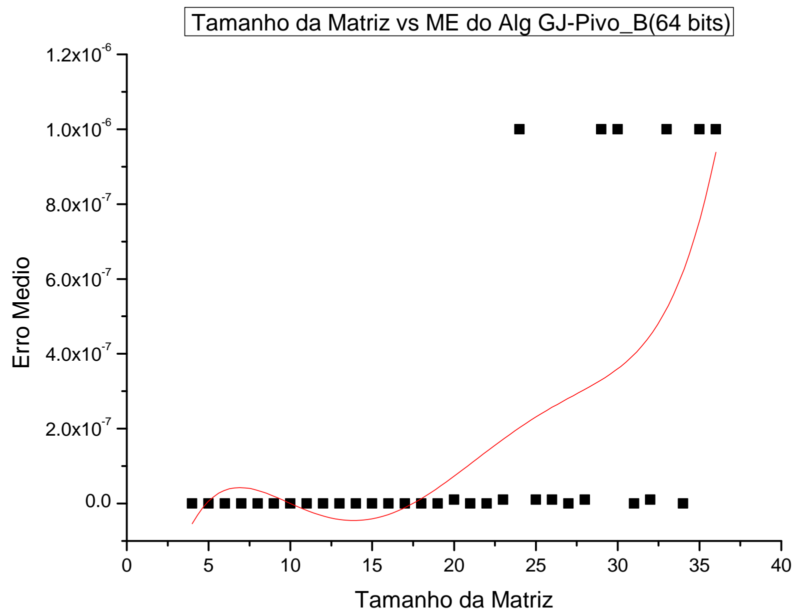

Figura 5.7. Erro médio para o algorítimo GJ-Pivô_B usando precisão dupla

Por outro lado, o algoritmo GJ-Pivô_B necessariamente avalia qual dos elementos a baixo da coluna j é o maior, para logo armazenar o mesmo na linha i em relação ao pivô.

Como se pode observar na figura 5.9, é apresentado o tempo de execução mínimo que utiliza

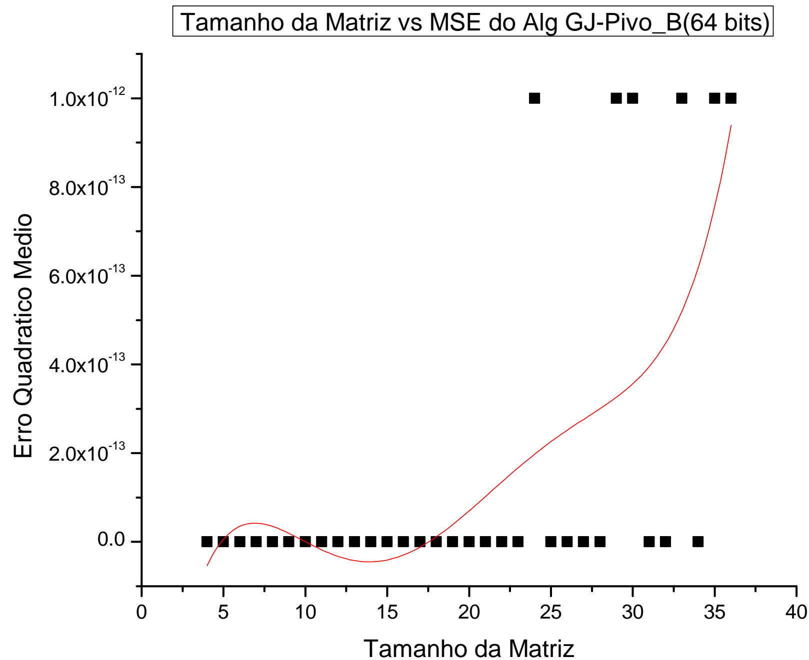

Figura 5.8. Erro quadrático médio para o algorítimo GJ-Pivô_B usando precisão dupla

cada algoritmo nas comparações, apenas para encontrar o pivô sem considerar o acesso na memória RAM. É importante ressaltar que o tempo de execução (em ciclos de relógio) necessários para terminar as comparações, segundo o algoritmo GJ-Pivô_A, foi inferior ao utilizado pelo algoritmo GJ-Pivô_B.

A escolha do algoritmo GJ-Pivô_B foi feita pela precisão do mesmo, pois trata-se de um fator importante devido a que os cálculos da inversão de matrizes apresentam um erro menor quando usado.

5.2 A unidade de Change_Row

Como se explicou no capitulo **??** foram feitas 3 modificações na unidade de *Change_Row*, os resultados de desempenho mostram que a unidade *Change_Row-03* (figura 5.12) apresenta melhores resultados quando comparada com *Change_Row-01* (figura 5.10). Note-se que a unidade *Change_Row-01* foi desenvolvida usando o acesso sequencial (elemento por elemento da matriz), e além disso, foi feita uma permutação física das linhas, ou seja parte do tempo de execução é devido aos acessos de leitura/escritura na memória RAM. Porém, nas duas ultimas unidades *Change_Row-02*(figura 5.11) e *Change_Row-03* foi usado o acesso por linhas, devido a que todos os elementos da linha foram armazenados em cada posição da memória RAM. Adi-

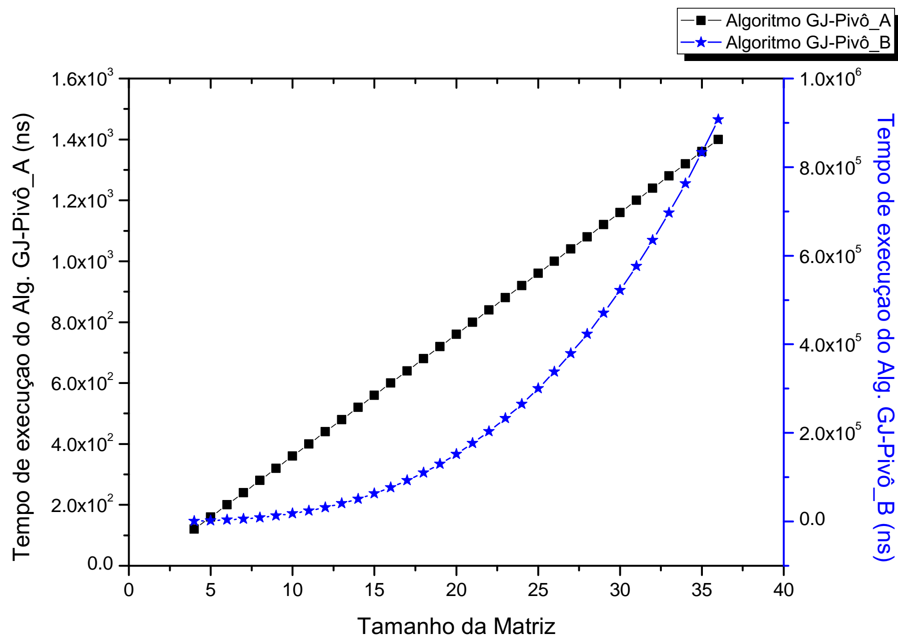

Figura 5.9. Número de comparações mínimo para encontrar o Pivô

cionalmente, esta última unidade não implementa uma permutação física de linhas, fato que melhora notavelmente o desempenho.

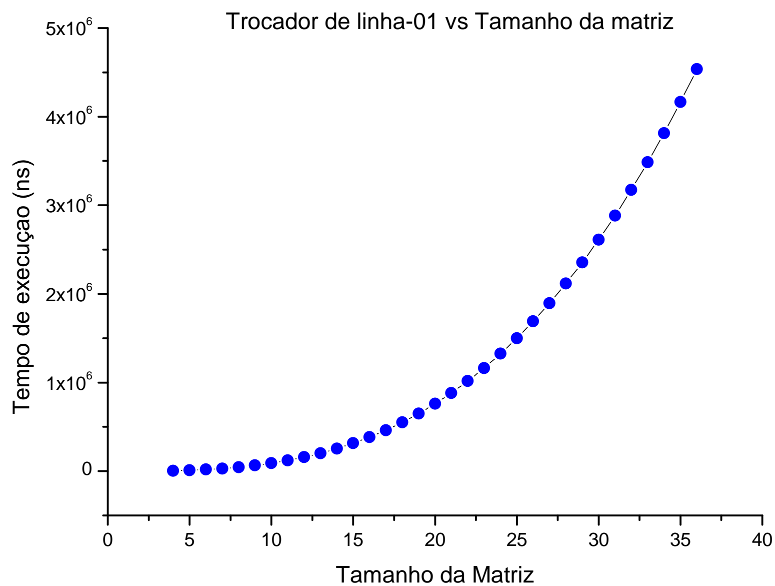

Figura 5.10. Desempenho da unidade Trocador de linhas-01

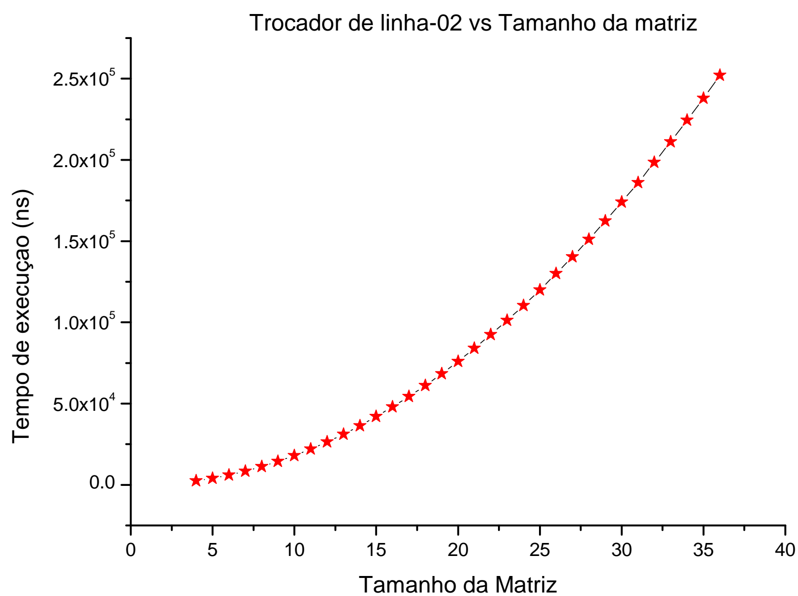

Figura 5.11. Desempenho da unidade Trocador de linhas-02

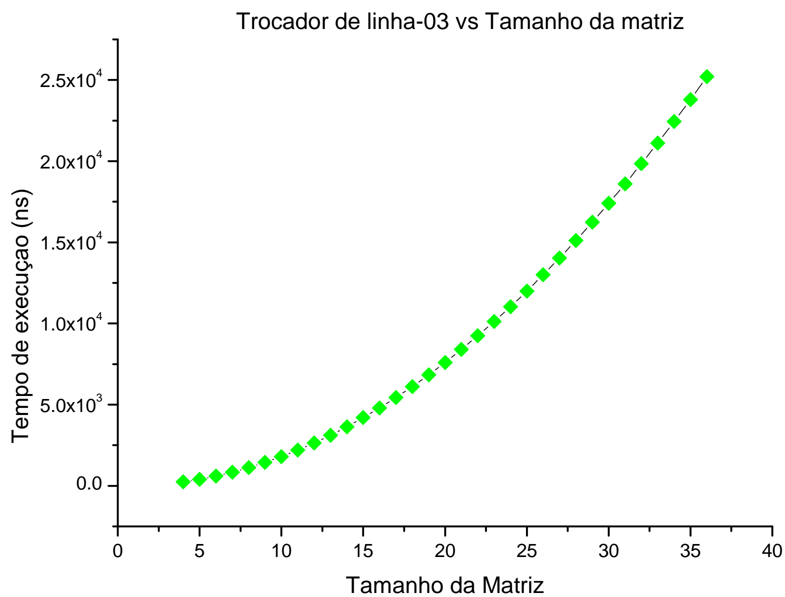

Figura 5.12. Desempenho da unidade Trocador de linhas-03

5.3 Arquiteturas de inversão de matrizes

As 3 arquiteturas desenvolvidas junto a seus correspondentes consumos de recursos e desempenho são apresentadas nesta seção. As figuras 5.13 e 5.14 mostram o tempo de execução

usado pela *FPGA* nas diferentes arquiteturas para calcular a inversão da matriz usando um *clock* de 50MHz comparado contra *MatLab*, usando uma PC com processador: AMD Turion-X2 Dual Core Mobile RM-72 2.10 GHz, 4GB de memória RAM e um sistema operacional de 32 bit (*Microsoft Windows Vista Home Premium*).

Uma aplicação foi executada durante cada teste para evitar uma sobrecarga da CPU. Para começar nas figuras 5.13 e 5.14 se apresentam os desempenhos das arquiteturas que são do tipo *sequencial* e *pipeline*. Observa-se que as arquiteturas mantêm sempre um desempenho abaixo do alcançado pelo *MatLab*. Este baixo desempenho é devido ao pouco paralelismo tanto na arquitetura *Sequencial* quanto na *pipeline*, assim como pelos múltiplos acessos na memória RAM, no caso da arquitetura sequencial.

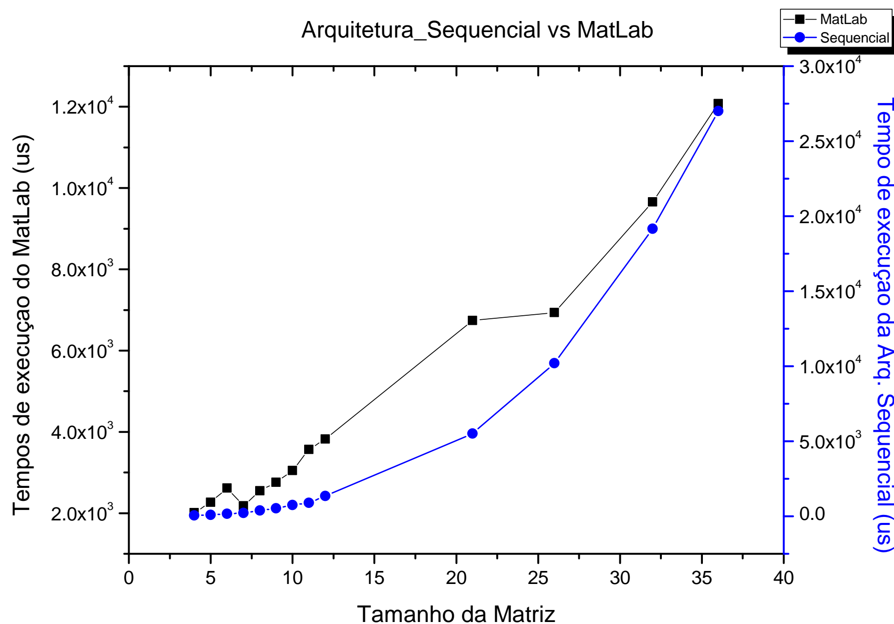

Figura 5.13. Desempenho da Arquitetura Sequencial

A pouca ou nula exploração do paralelismo que fornecem as *FPGAs* resultam em uma arquitetura com um consumo baixo de recursos totais disponíveis neste caso pela *Virtex-5* usada. Nas tabelas 5.1 e 5.2 se apreciam os diferentes recursos usados pelas arquiteturas após do processo de *posicionamento* e *roteamento*, para matrizes 4×4 e 36×36. A máxima frequência de operação da arquitetura proposta para a matriz de 4×4 é 150.246MHz e para a matriz de 36×36 é de 88.964MHz. Adicionalmente, o desempenho do circuito é reduzido devido à que o número de operações aritméticas é incrementado na arquitetura. O consumo de recursos mostra que este FPGA especifico pode suportar matrizes com grandes tamanhos. Por exemplo, a tabela 5.2 mostra o consumo de LUT de 5467, o qual representa aproximadamente o 8% das LUTs totales disponíveis na FPGA.

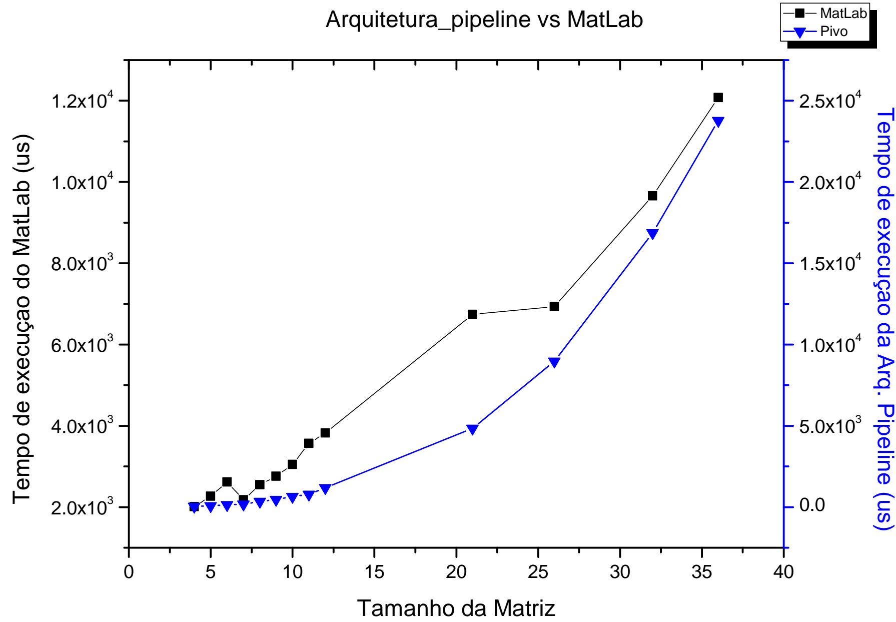

Figura 5.14. Desempenho da Arquitetura Pipeline

Tabela 5.1. Resultados da Sínteses para N=4 na arquitetura sequencial

Unit	LUTs	DSP48(E)	Freq. [MHz]
Subtractor	845	–	504.286
Multiplier	146	4	616.797
Division	426	6	598.223
GJ	2113	–	150.246
Total	3530(5.10%)	10(15.6%)	150.246

As duas arquiteturas apresentadas anteriormente indicam pouco consumo de recursos, porém um alto custo no desempenho quando comparado com o estimador estático (*MatLab*) usando as características do PC mencionado. No entanto a última, mas não menos importante arquitetura *Paralelo*, desenvolve a tarefa de inversão com um alto desempenho novamente comparado com o *MatLab* como mostra a figura 5.15. Esta arquitetura apresenta um desempenho melhor, e em nenhum dos casos os valores alcançados por esta arquitetura são maiores que os valores alcançados pelo *MatLab*, ainda usando um *clock* de 50MHz para realizar as simulações na *FPGA*. Assim, esta última proposta melhora o desempenho alcançado tanto pelas duas arquiteturas anteriores quanto pelo *MatLab*.

Ao analisar os recursos usados, se detecta um aumento no consumo de *DSP48* como esperado, pois foram implementados 10 multiplicadores (8 á mais que as arquiteturas anteriores), o que resultou em um aumento considerável nos *DSP48* como mostrado na tabela 5.3. Além as tabelas

Tabela 5.2. Resultados da Sínteses para N=36 na arquitetura sequencial

Unit	LUTs	DSP48(E)	Freq. [MHz]
Subtractor	845	–	504.286
Multiplier	146	4	616.797
Division	426	6	598.223
GJ	4050	–	88.964
Total	5467(7.9%)	10(15.6%)	88.964

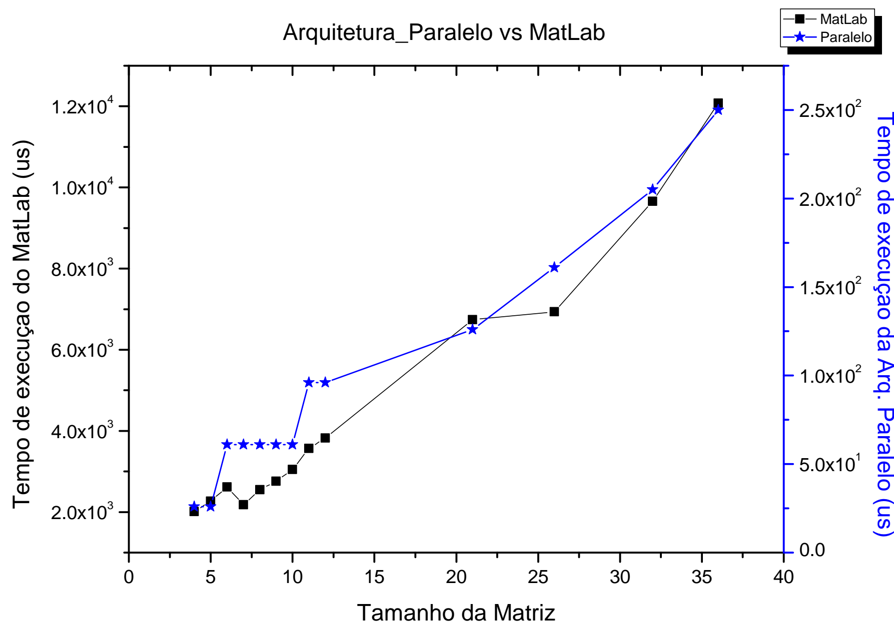

Figura 5.15. Desempenho da arquitetura Paralelo

5.3 e 5.4 mostram o consumo de *LUTs* e a frequência máxima que a arquitetura suporta, no caso de uma matriz de 4×4 o valor é de 301.416MHz, sendo que para uma matriz de 36×36 o valor é de 166.984MHz.

O comportamento da frequência a medida que o tamanho da matriz aumenta, significa uma redução no desempenho pelo fato de gerar laços de repetição maiores para realizar as operações de inversão (figura 5.16).

5.4 Conclusão do Capítulo

Os resultados das implementações em *hardware* das arquiteturas propostas foram mostrados neste capítulo, de acordo com as diferentes modificações feitas em cada unidade, visando manter o *tradeoff* entre área e desempenho.

Tabela 5.3. Resultados da Sínteses para N=4 na arquitetura Paralelo

Unit	LUTs	DSP48(E)	Freq. [MHz]
Subtractor	845	–	504.286
Multiplier	146	16	616.797
Division	426	6	598.223
GJ	7132	–	301.416
Total	8549(12.36%)	22(34.3%)	301.416

Tabela 5.4. Resultados da Sínteses para N=36 na arquitetura Paralelo

Unit	LUTs	DSP48(E)	Freq. [MHz]
Subtractor	845	–	504.286
Multiplier	146	16	616.797
Division	426	6	598.223
GJ	7132	–	166.984
Total	11723(16.9%)	22(34.3%)	166.984

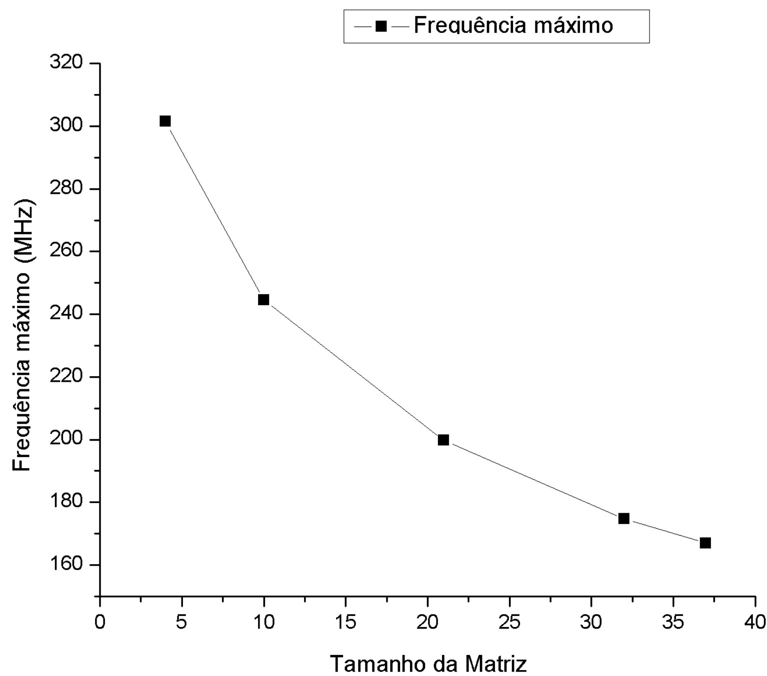

Figura 5.16. Comportamento da frequência á medida que o tamanho da matriz aumenta

A arquitetura *Paralela* é sem duvidas uma proposta em hardware interessante, apresentou melhor desempenho quando comparada com as arquiteturas *pipeline* e *sequencial*, inclusive me-

lhora o tempo de execução, comparado com o *MatLab* rodando no PC com as características anteriormente mencionadas(Seção 5.3). Adicionalmente, pode-se apreciar que a mesma implementa eficazmente o acesso à memoria RAM, explorando a vantagem do comprimento da palavra. No entanto, esta arquitetura usa mais área para satisfazer o paralelismo quando comparado novamente com as arquiteturas *pipeline* e *sequencial*.

Ao contrario da arquitetura *Paralela*, a arquitetura *pipeline* apresenta um consumo menor de recursos permitindo implementações de baixo *footprint*.

6 Conclusões

6.1 Comentários finais e conclusões

O algoritmo Gauss-Jordan foi implementado em um *hardware* reconfigurável baseado em *FPGAs*, desenvolvendo três diferentes arquiteturas parametrizáveis para calcular inversões de matrizes, permitindo assim realizar o análise do *tradeoff* entre área e desempenho. A implementação usa representação em ponto flutuante com precisão simples e dupla, e consegue realizar os cálculos de inversão de matrizes de tamanhos 4×4 ou até maiores como no caso da matriz de 36×36.

A análise detalhada das sequências de operações do algoritmo Gauss-Jordan selecionado levou à separação deste em etapas: (a) Pivô, (b) Trocador de linhas, (c) Eliminação de Matriz e (d) Normalização, permitindo a transcrição deste algoritmo em arquiteturas que exploram eficazmente sua característica de paralelismo assim como a implementação de um pipeline de operações.

Duas modificações possíveis para o algoritmo de Gauss-Jordan na etapa de Pivô, foram analisadas e implementadas, considerando como fator determinante o erro propagado ao longo dos cálculos de inversão, alcançando resultados de 10^{-4} para o *ME* e de 10^{-7} para o *MSE*, usando precisão simples, e de 10^{-8} para o *ME* e de 10^{-14} para o *MSE* para precisão dupla.

Analisando os recursos apresentes na *Virtex-5*, implementaram-se duas formas de acesso nas memórias RAM, assim como também foram usados dois tipos de memórias RAM para alcançar as vantagens do paralelismo, diminuindo consideravelmente o acesso nas mesmas. Adicionalmente, aproveitando a flexibilidade que estas memórias RAM têm no comprimento da palavra, foi reduzido o gargalo de von Neumann e aproveitadas as vantagens de paralelismo dos FPGAs.

A etapa de Trocador de linhas teve 3 modificações importantes devido as duas memórias RAM usadas, passando por uma permutação de linhas física escrevendo/lendo cada elemento da linha na memória RAM, até uma última modificação na qual foi desenvolvida uma arquitetura que implementou um registro evitando esta escrita na memória RAM. O uso do paralelismo aumentou o desempenho notavelmente na etapa de Eliminação de Matriz e Normalização, o que

ilustra a aplicabilidade de FPGAs para processamento de cálculos matriciais.

Houve uma preocupação com o consumo de recursos de *hardware* das arquiteturas. O uso do paralelismo foi testado em simulação e teve resultados positivos de síntese, os quais mostram que o método é uma boa alternativa para a otimização de processos. Porém, a implementação dessas novas arquiteturas é mais complexa, pois exige um controle mais sofisticado de sincronização entre os processos.

Adicionalmente, a etapa de Normalização foi usada no final do processo, no intuito de diminuir a propagação de erro ao longo dos cálculos, pois é necessário aplicar apenas esta etapa no lado direito da matriz aumentada.

6.2 Propostas de futuros trabalhos

Á continuação são apresentadas 4 propostas:

6.2.1 GECO

A primeira proposta de continuação de trabalho é desenvolver uma ferramenta em C (*GECO-Gerador de COREs*) com interfase de usuário, que gera de maneira automática a parametrização das arquiteturas para inversão de matrizes.

6.2.2 Explorar as RAM internas

Este trabalho focou no acesso das memórias RAM internas na Virtex-5, em que no caso do dispositivo usado conta com 148 memórias de 36Kb. A capacidade destas ainda não foi totalmente explorada e implementações de matrizes de tamanhos maiores poderia ser facilmente avaliável.

6.2.3 Paralelismo em um pipeline de operações

As vantagens demostradas do paralelismo sendo esta uma das características mais importantes dos FPGAs, seria notavelmente melhorada com uma implementação deste em um pipeline de operações.

6.2.4 Métodos e correção do erro

O erro é um fator importante que marca o critério de aceitação do cálculo da inversão da matriz. Estudar e implementar métodos de arredondamento assim como a eliminação do erro de truncamento das bibliotecas de ponto flutuante usadas, incrementaria a precisão dos cálculos.

REFERÊNCIAS BIBLIOGRÁFICAS

[A. 1989]A., D. K. E.-A. Parallel vlsi algorithm for stable inversion of dense matrices. *IEEE Proceedings-Computers and Digital Techniques*, v. 136, 1989.

[A. Piirainen O. 2005]A. PIIRAINEN O., B. A. H. Gsm channel estimator using a fixed-point matrix inversion algorithm. 2005.

[A.Irturk e R.Kastner 2008]A.IRTURK, B.; R.KASTNER. Automatic generation of decomposition based matrix inversion architectures. 2008.

[A.Irturk B.Benson 2007]A.IRTURK B.BENSON, S. a. R. Gusto:an automatic generation and optimization tool for matrix inversion architectures. 2007.

[ANTON HOWARD A. 2001]ANTON HOWARD A., R. C. *Álgebra Linear Com Aplicações.* BOOKMAN COMPANHIA ED, 2001.

[Bernard Kolman. 2001]BERNARD KOLMAN., D. R. H. *Álgebra Linear Com Aplicações.* LTC ED, 2001.

[Bigdeli Morteza Biglari-Abhari e Lai 2006]BIGDELI MORTEZA BIGLARI-ABHARI, Z. S. A.; LAI, Y. T. A new pipelined systolic array-based architecture for matrix inversion in fpgas with kalman filter case study. 2006.

[Burian A. 2003]BURIAN A., T. J.-Y. M. A fixed-point implementation of matrix inversion using cholesky decomposition. *Proceedings of the 46th IEEE International Midwest Symposium on Circuits and Systems.*, p. 1431–1434, 2003.

[CARPENTIER 1993]CARPENTIER, M. P. J. *ANÁLISE NUMÉRICA.*
: [s.n.], 1993.

[C.K. PrasadS.H. 2007]C.K. PRASADS.H., B. S. Vlsi architecture for matrix inversion using modified gram-schmidt based qr decomposition. 2007.

[D. Eilert J. 2007]D. EILERT J., L. D. W. D. A.-D. N. M. H. W. Fast complex valued matrix inversion for multi-user stbc-mimo decoding. 2007.

[Duarte Horácio Neto 2009]DUARTE HORÁCIO NETO, M. V. R. Double-precision gauss-jordan algorithm with partial pivoting on fpgas. 2009.

[Edman e Öwall 2005]EDMAN, F.; ÖWALL, V. A scalable pipelined complex valued matrix inversion architecture. 2005.

[FPGA Design Flow Overview]FPGA Design Flow Overview. `http://www.xilinx.com/itp/xilinx8/help/iseguide/html/ise_fpga_design_flow_overview.htm`.

[Gathen e Gerhard 2003]GATHEN, J. von zur; GERHARD, J. *Modern Computer Algebra.* : [s.n.], 2003.

[Hartenstein 2001]HARTENSTEIN, R. Coarse grain reconfigurable architectures. p. 564 –569, 2001.

[Hartenstein 2003]HARTENSTEIN, R. Are we really ready for the breakthrough? [morphware]. p. 7 pp., 22-26 2003. ISSN 1530-2075.

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

[Hartenstein 2010]HARTENSTEIN, R. Reconfigurable computing: boosting software education for the multicore era: Why we need to reinvent computing. /, p. 1 –1, 24-26 2010.

[Huang e Hon 2001]HUANG, A. A. X.; HON, H.-W. *Spoken Language Processing: A Guide to Theory, Algorithm and System Development.* [s.n.], 2001.

[Irturk 2009]IRTURK, A. U. *GUSTO: General architecture design Utility and Synthesis Tool for Optimization.* Tese (Doutorado) — Computer Science, 2009.

[Irturkt Bridget Bensont e Kastnert 2008]IRTURKT BRIDGET BENSONT, S. M. A.; KASTNERT, R. An fpga design space exploration tool for matrix inversion architectures. 2008.

[Kehtarnavaz N; Gamadia 2006]KEHTARNAVAZ N; GAMADIA, M. Real-time image and video processing: From research to reality. *Editora Morgan & Claypool*, 2006.

[Kelly F.; Kokaram 1999]KELLY F.; KOKARAM, A. Fast image interpolation for motion estimation using graphics hardware. *Proceedings of SPIE The International Society for Optical Engineering*, 1999.

[Kovar J. Kloub 2008]KOVAR J. KLOUB, J. S. A. H. P. Z. A. H. B. Rapid prototyping platform for reconfigurable image processing. *Proceedings of Mezinarodni Conference Technical Computing Program*, 2008.

[LogiCORE IP Block Memory Generator v4.2 2010]LOGICORE IP Block Memory Generator v4.2. 2010. http://www.xilinx.com/support/documentation/ip_documentation/blk_mem_gen_ds512.pdf.

[M.Karkooti J.R.Cavallaro 2005]M.KARKOOTI J.R.CAVALLARO, C. Fpga implementation of matrix inversion using qrd-rls algorithm. 2005.

[Muñoz-Arboleda D. F. Sanchez e Ayala-Rincón 2009]MUñOZ-ARBOLEDA D. F. SANCHEZ, C. H. L. D.; AYALA-RINCóN, M. Tradeoffs of fpga design of floating-point transcendental functions. *In Proc. The 17th IFIP/IEEE International Conference on Very Large Scale Integration - VLSI-SOC*, 2009.

[Pavel F. Otto 2001]PAVEL F. OTTO, R. M. V. P. Z. Imaging algorithm speedup using co-design. *In Summaries Volume Process Control,Strbske Pleso*, 2001.

[Sanchez 2009]SANCHEZ, D. F. *Implementacao em VHDL de uma Biblioteca Parametrizavel de Operadores Aritmeticos em Ponto Flutuante Para Serem Usados em Problemas de Robotica.* Dissertação (Mestrado), 2009.

[Sánchez D. Muñoz e Ayala-Rincón 2009]SÁNCHEZ D. MUÑOZ, C. L. D.; AYALA-RINCÓN, M. Parameterizable floating-point library for arithmetic operations in fpgas. 2009.

[Silva 2010]SILVA, J. Y. M. A. da. *Implementação de técnicas de processamento de imagens no domínio espacial em sistemas reconfiguráveis.* Dissertação (Mestrado), Master's thesis(in Portuguese), Faculdade de Tecnologia, Universdidade de Brasilia- Brazil, 2010.

[Thomas L. Howes 2009]THOMAS L. HOWES, W. L. D. A comparison of cpus, gpus, fpgas, and massively parallel processor arrays for random number generation. *Proceedings of the International Symposium on Field Programmable Gate Arrays*, 2009.

[Trefethen 1997]TREFETHEN, D. B. I. L. N. *Numerical Linear Algebra.* : SIAM: Society for Industrial and Applied Mathematics, 1997.