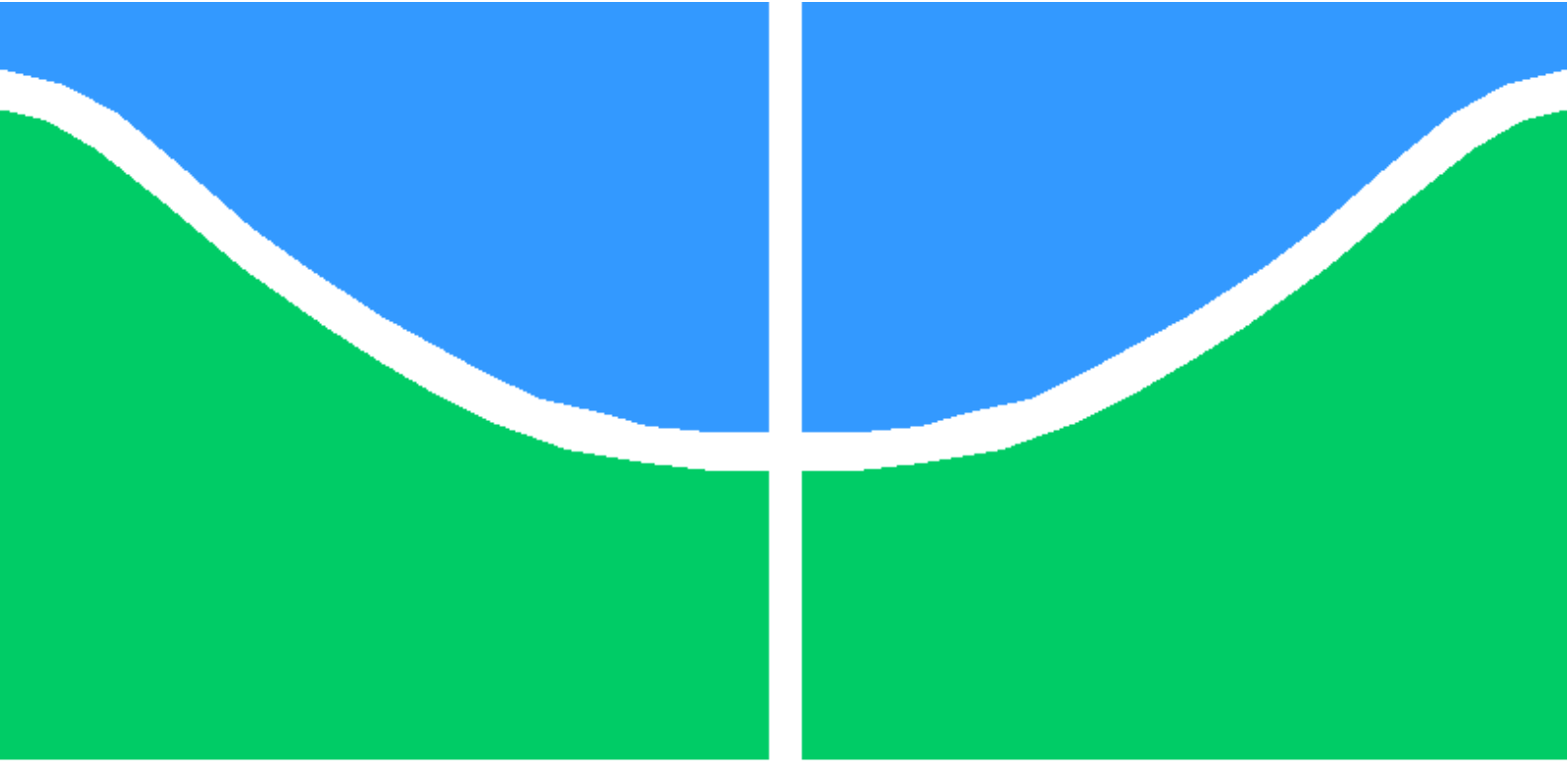


DISSERTAÇÃO DE MESTRADO

IMPLEMENTAÇÃO EM VHDL DE UMA BIBLIOTECA PARAMETRIZÁVEL DE OPERADORES ARITMÉTICOS EM PONTO FLUTUANTE PARA SER USADA EM PROBLEMAS DE ROBÓTICA

DIEGO FELIPE SÁNCHEZ GÓMEZ

Brasília, dezembro de 2009

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASILIA
Faculdade de Tecnologia
Departamento de Engenharia Mecânica

IMPLEMENTAÇÃO EM VHDL DE UMA BIBLIOTECA PARAMETRIZÁVEL DE OPERADORES ARITMÉTICOS EM PONTO FLUTUANTE PARA SER USADA EM PROBLEMAS DE ROBÓTICA

DIEGO FELIPE SÁNCHEZ GÓMEZ

ORIENTADOR: CARLOS HUMBERTO LLANOS QUINTERO

DISSERTAÇÃO DE MESTRADO EM SISTEMA MECÂTRONICOS

Brasília, dezembro de 2009

UNIVERSIDADE DE BRASILIA
Faculdade de Tecnologia
Departamento de Engenharia Mecânica

DISSERTAÇÃO DE MESTRADO

IMPLEMENTAÇÃO EM VHDL DE UMA BIBLIOTECA PARAMETRIZÁVEL DE OPERADORES ARITMÉTICOS EM PONTO FLUTUANTE PARA SER USADA EM PROBLEMAS DE ROBÓTICA

DIEGO FELIPE SÁNCHEZ GÓMEZ

Relatório submetido ao Departamento de Engenharia Mecânica da faculdade de Tecnologia da Universidade de Brasília como parte dos requisitos necessários para a obtenção do grau de Mestre em Sistemas Mecatrônicos

Banca Examinadora

Prof. Carlos Llanos Quintero, Dr., ENM/UnB _____
Orientador

Prof. Edward D. Moreno Ordoñez Dr., UFS _____
Examinador externo

Prof. Geovany Araujo Borges Dr., ENE/UNB _____
Examinador externo

Brasília 11 de dezembro de 2009

FICHA CATALOGRÁFICA

SÁNCHEZ G., DIEGO FELIPE
IMPLEMENTAÇÃO EM VHDL DE UMA BIBLIOTECA PARAMETRIZÁVEL DE OPERADORES ARITMÉTICOS EM PONTO FLUTUANTE PARA SER USADA EM PROBLEMAS DE ROBÓTICA [Distrito Federal] 2009.
vii, 87p. 210 × 297 mm (ENM/FT/UnB, Mestre, Sistemas Mecatrônicos, 2009). Dissertação de Mestrado – Universidade de Brasília. Faculdade de Tecnologia.
Departamento de Engenharia Mecânica.

1. Aritmética de Ponto flutuante
2. FPGAs
3. Cinemática de Manipuladores
4. Cinemática direta

I. ENM/FT/UnB
II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

SÁNCHEZ, DIEGO F. (2006). Implementação em VHDL de uma biblioteca parametrizável de operadores aritméticos em ponto flutuante para ser usada em problemas de robótica. Dissertação de Mestrado em Sistemas Mecatrônicos, Publicação ENM.DM-30A/09, Departamento de Engenharia Mecânica, Universidade de Brasília, Brasília, DF, 87p.

CESSÃO DE DIREITOS

Diego Felipe Sánchez Gómez
SCLN 407 Bloco C, Apartamento 216
70855-030 Brasília-DF-Brasil

Agradecimentos

Quero agradecer inicialmente a Deus por ter colocado no meu caminho pessoas maravilhosas que fizeram possível o desenvolvimento deste trabalho. A minha mãe, Malely Gómez e aos meus irmãos Johana Sánchez Gómez e José Omar Sánchez Gómez, por seu apoio e estimulo.

Aos professor Carlos Llanos e Mauricio Ayala, no desenvolvimento da pesquissa.

Aos meus colegas e amigos do GRACO, especialmente ao Daniel Muñoz, Jones Yudi, Janier Arias, Ronald Hurtado, Evandro Teixeira, Maria Cristina Gallego pela ajuda, conselhos, também pelas alegrias compartidas e pela companhia nos momentos difíceis.

Á 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.

DIEGO FELIPE SÁNCHEZ GÓMEZ

RESUMO

Este trabalho descreve a implementação em FPGA de uma biblioteca parametrizável em ponto flutuante abrangendo operadores aritméticos e trigonométricos comumente encontrados em aplicações robóticas (a saber, soma, subtração, divisão, raiz quadrada, seno, cosseno e arco-tangente). A biblioteca é parametrizável pelo tamanho da palavra (expoente e mantissa) para o conjunto de operadores. Esta biblioteca foi usada na implementação hardware da cinemática direta de um robô manipulador de configuração esférica com cinco graus de liberdade. No intuito de gerar automaticamente o código VHDL dos operadores em ponto flutuante para diferentes tamanhos de palavra, definida pelo usuário, uma ferramenta CAD foi desenvolvida em MATLAB. Em este contexto, são usados algoritmos baseados em Goldschmidt e Newton-Raphson para calcular as operações de divisão e raiz quadrada. Um análise entre os parâmetros consumo de recursos, consumo de multiplicadores embebidos do FPGA e aspetos relacionados à latência e precisão, para as duas arquiteturas (Goldschmidt e Newton-Raphson), permite escolher a melhor arquitetura para uma aplicação especifica. Para ambas implementações são obtidas altas vazões (Mresultados/s). A análise mostra uma vantagem da arquitetura Newton-Raphson sobre a arquitetura Goldschmidt.

Para o cálculo das funções transcendentais são implementadas duas arquiteturas baseada em CORDIC e expansão em séries de Taylor. Similarmente ao caso da divisão e raiz quadrada, é feito uma análise entre os parâmetros consumo de recursos, consumo de multiplicadores embebidos do FPGA, latência e precisão. A análise mostrou que a arquitetura baseada em expansão em séries de Taylor apresentou um melhor desempenho para o cálculo das funções seno e cosseno. Porém, a arquitetura CORDIC apresenta um melhor desempenho para o cálculo da função arco-tangente. Para validar os circuitos propostos, varias simulações usando MATLAB foram desenvolvidas (os resultados são usados como um estimador estático dos operadores em ponto flutuante). Os resultados do erro quadrático médio (MSE) mostram uma adequada precisão dos operadores implementados.

Finalmente, no intuito de validar a funcionalidade do conjunto de operadores, foi desenvolvida uma arquitetura hardware para o cálculo da cinemática direta de um robô manipulador de configuração esférica com cinco graus de liberdade. A arquitetura proposta foi concebida usando uma abordagem de planejamento por restrição de tempo e área. Em este caso, as unidades de ponto flutuante foram usadas, 8 unidades para o cálculo de funções transcendentais (seno e cosseno), e 4 multiplicadores e 2 unidades de soma/subtração para o cálculo de operações aritméticas. Os resultados de síntese mostram que a arquitetura proposta é viável em famílias de FPGAs modernas, nas quais há uma abundancia de elementos lógicos disponíveis. Todos os cálculos foram executados com sucesso e também validados usando os resultados em MATLAB como um estimador estatístico. O tempo de processamento da arquitetura hardware foi comparada com o mesmo modelo cinemático em software, usando um processador *hard* PowerPC (embebido na FPGA) e uma implementação em MATLAB executada em um processador AMD Atlhon Dual-Core de 2,1GHz. O tempo de cálculo da cinemática direta implementada em hardware é 0,67μs (relógio de 100MHz), considerando a mesma formulação implementada em software usando um processador PowerPC necessita 1,61ms, executada no mesmo dispositivo FPGA e com o mesmo relógio. Adicionalmente, a implementação em MATLAB gasta 0,013ms (relógio de 2.1GHz).

ABSTRACT

This work describes an FPGA implementation of a parameterizable floating-point library including arithmetic and trigonometric operators commonly found in robotic applications (namely, addition, subtraction, division, square root, sine, cosine and arctangent). The library is parameterizable by word length (exponent and mantissa) for the set of operators. This library was used in the hardware implementation of a direct kinematics for a robot manipulator with a spherical topology with five degrees of freedom. In order to generate automatically the VHDL code of the floating point operators for different word length representations, which are defined by the designer, a CAD tool were developed in MATLAB. In this context, algorithms based on Goldschmidt and Newton-Raphson has been implemented for calculating both division and square root operators. A tradeoff analysis among the area cost, FPGA embedded multipliers consumption and aspects related to the latency and precision, for both developed architectures (Goldschmidt and Newton-Raphson), allows the choice of the better architecture for a specific application. High throughputs (Mresults/s) are achieved for both implementations. The trade-off analysis shows advantages of the Newton-Raphson over the Goldschmidt architecture.

For computing the transcendental functions two architectures based on CORDIC and Taylor series expansion were implemented. Similarly to the case of the division and square root, a tradeoff analysis among the area cost, FPGA embedded multipliers consumption, latency and precision issues has been developed. The trade-off analysis shows that the architecture based on Taylor series expansion presents a better performance for computing sin a cosine functions. However, the CORDIC architecture presents a better performance for computing the arctangent function. For validating the proposed circuits, several simulations were performed using the MATLAB (these results are used as a statistical estimator of the implemented floating point operators). The Mean Square Error (MSE) results have demonstrated the suitability the precision of the implemented operators.

Finally, in order to validate the functionality of the set of operators, a hardware architecture for computing the direct kinematics for a robot manipulator with a spherical topology with five degrees of freedom was developed. The proposed architecture was designed using a time and area constrained scheduling approach. In this case, cores in floating poitn were used, 8 cores for computing transcendental functions (sine and cosine), 4 multiplier and 2 addition/subtraction cores for computing the arithmetic operations. Synthesis results show that the proposed architecture is feasible for modern FPGAs families, in which there are plenty of logic elements available. The overall computations were successfully performed and also validated using the MATLAB results as a statistical estimator. The processing time of the hardware architecture was compared with the same kinematics model implemented in software, using both a hard PowerPC processor (embedded in the FPGA) and a MATLAB implementation running in an AMD Atlhon Dual-Core at 2.1GHz processor. The computation time to implement the direct kinematics in hardware is 0.67μs (clock of 100MHz), whereas the same formulation implemented in software using a PowerPC processor needs 1.61ms, running in the same FPGA device and with the same clock. Additionally, the MATLAB implementation spends 0.013ms (clock of 2.1GHz).

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

Capítulo 1

Introdução

Aplicações como cálculo da cinemática direta e inversa na área da robótica, processamento digital de sinais e imagens, sistemas de controle e muitas outras aplicações cientificas precisam de alto número de operações aritméticas e trigonométricas que devem ser calculadas com alta precisão [4, 5]. De esta forma, a adoção de aritmética de ponto flutuante baseadas no padrão IEEE-754 é um requisito indispensável para este tipo de aplicações, comumente implementadas sobre processadores de propósito gerais (GPPs). A natureza seqüencial destes processadores (baseados na arquitetura ou modelo de von Neumann), e o fato de em alguns casos estes emulam operações de ponto flutuante em software, faz com que estes não sejam o suficientemente rápidos para algumas aplicações [1, 6, 7].

O continuo avance nas tecnologias de fabricação de circuitos integrados fez com que hoje tenha-se uma alta capacidade de integração, permitindo implementar em hardware várias operações em ponto flutuante (comumente implementadas em software), aproveitando assim o paralelismo intrínseco tanto dos algoritmos como dos dispositivos reconfiguráveis como FPGAs, tendo como conseqüente um ganho no desempenho (velocidade de processamento) [8, 9, 10, 11].

Propor uma biblioteca de ponto flutuante para FPGAs que seja usável em aplicações como robótica é um grande desafio, já que permite explorar as potencialidades dos dispositivos FPGAs e comparar com processadores embarcados no FPGA, tendo uma medida de velocidade de processamento entre as duas abordagens.

1.1 Generalidades

Uma das principais vantagens ao usar implementações em hardware de algoritmos é a execução em paralelo que pode ser obtida. Na figura 1.1 é mostrado um comparativo entre uma implementação hardware e uma implementação software [1]. Na implementação software, se cada linha do programa é executada em 1 ciclo de relógio o resultado G, é produzido em 12 ciclos de relógio; porém, na implementação hardware esta mesma implementação levaria 2 ciclos de relógio.

Por outro lado, para que um robô poda-se movimentar e manipular objetos precisa-se de uma adequada localização, baseada em ferramentas matemáticas que permitem localizar em posição e orientação um objeto (extremo do robot) no espaço tridimensional. O estudo da cinemática do

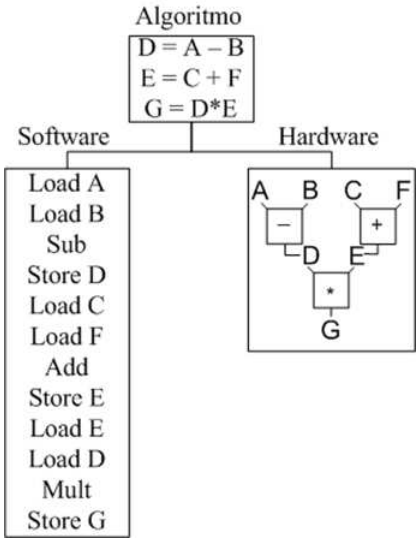


Figura 1.1: Software versus Hardware (Modificado [1])

robô permite relacionar a posição dos atuadores com a posição e orientação do extremo final do robô [12].

Um clássico pero importante problema na área da robótica de manipuladores é a cinemática direta que permite a localização cartesiana do extremo final do robô a partir do conhecimento da posição/ângulos nas articulações [13, 12]. A cinemática direta é uma formulação matemática que freqüentemente necessita de um elevado número de operações aritméticas e trigonométricas que, idealmente, devem ser calculadas em aritmética de ponto flutuante.

1.2 Descrição do Problema

Em uma arquitetura digital os números são representados usando seqüências binárias de largura finita, o que leva a erros de quantização na representação de parâmetros da implementação e erros de arredondamento nas operações aritméticas. Além do anterior, os erros dependem da aritmética usada na implementação [14, 15].

Em muitas aplicações cientificas, devido a limitações da faixa de valores de números representados usando aritmética de ponto fixo, pode-se gerar erros de *overflow* e *underflow*. De esta forma, para se representar tanto números grandes como pequenos precisa-se de uma grande quantidade de bits. Uma solução a este problema é a utilização de aritmética de ponto flutuante, que permite representar em um registro de bits com um tamanho adequado, números pequenos e grandes ao mesmo tempo [16, 5].

Se uma rotina de cálculo é implementada em um GPP dispondo de aritmética de ponto flutuante, esta é a escolha mais natural. Por outro lado, é comum em implementações em hardware usar aritmética de ponto fixo visando economia em térmos de área do chip [17, 15].

Uma má escolha do formato numérico no qual se operam e representam os números pode levar a falhas no cálculo, causando catástrofes que levam à perda de vidas e recursos materiais. Uma

das falhas catastróficas encontradas na literatura foi a do míssil Patriot descrito embaixo.

Em 25 de fevereiro de 1991, durante a guerra do Golfo, o míssil americano Patriot falhou no caminho de interseção do míssil Scud iraquiano [1] [2]. O incidente causou a morte de 28 soldados e o ferimento de outras 100 pessoas. Um relatório técnico[3] chegou à conclusão que a causa da falha foi uma incerteza nos cálculos de tempo devido a erros aritméticos computacionais. Especificamente, o tempo em décimas de segundos, medido pelo relógio interno do sistema foi multiplicado por 1/10 para produzir o tempo em segundos. Este cálculo foi executado usando um registrador de 24 bits em ponto fixo. Em particular, o valor 1/10, não é exatamente representável em uma expansão binária usando 24 bits. O erro acumulado em cada iteração foi acumulando-se, produzindo finalmente um erro final significante.

Por outro lado, na área de robótica atualmente há uma alta demanda de aplicações que operam a altas velocidades, procurando uma melhora em desempenho e obtendo benefícios em termos de eficiência da manufatura, precisão e tempo de processamento. Em geral, um sistema robótico executado em tempo real precisa de iterativas execuções de complexos algoritmos, que envolvem o uso de várias funções transcendentais e operações aritméticas. Muitos destes algoritmos, precisam ser computados na ordem de milissegundos (ms) ou microssegundos (μs), para alcançarem as restrições de tempo do sistema. Além disso, estes algoritmos necessitam de alto poder computacional que supera as capacidades de muitos computadores seqüências baseados na arquitetura von Neumann [7].

Como descrito acima, uma implementação em FPGAs (*Field Programable Gate Array*) permitiria aproveitar o paralelismo intrínseco desta abordagem. Porém, a falta de uma completa biblioteca em aritmética de ponto flutuante há limitado o uso dos FPGAs em aplicações que requerem de alta precisão. Ainda que as FPGAs têm sido usadas para a implementação de aritmética de ponto fixo, esta não é uma boa escolha quando a aplicação precisa trabalhar em uma faixa de valores que abranja tanto números muito pequenos como números muito grandes, como apresentado no caso da falha do míssil Patriot.

Representações numéricas em uma amplia faixa dinâmica, jogam um papel importante no projeto de controle de robôs. Normalmente um robô funciona em uma faixa limitada de movimentos que facilmente poderiam ser representados e manipulados usando processadores digitais de sinais de aritmética de ponto fixo. Porém, eventos inesperados poderiam acontecer na linha de montagem. Por exemplo, o robô poderia se travar, ou algum objeto poderia bloquear seu movimento. Neste caso, o laço de retro-alimentação poderia sair da faixa de operação, e o sistema baseado em um DSP de ponto fixo poderia não incluir um efetivo tratamento de condições não-usuais. Tendo uma implementação baseada num processador de ponto flutuante, devido à amplia faixa de números que podem ser representados, dificilmente o robô superaria a faixa de operação e se torna possível trabalhar com circunstâncias não-usuais de uma forma controlada [18].

[1] `http://ta.twi.tudelft.nl/users/vuik/wi211/disasters.html`

[2] `http://www.ima.umn.edu/~arnold/disasters/patriot.html`

[3] `http://www.fas.org/spp/starwars/gao/im92026.htm`

1.3 Objetivos

1.3.1 Objetivo Geral

Este trabalho tem como objetivo principal desenvolver, implementar e validar uma biblioteca aritmética de ponto flutuante parametrizável descrita em VHDL, com a finalidade de servir para implementação de arquiteturas totalmente paralelizadas de cinemática direta e inversa de manipuladores.

A biblioteca descrita neste trabalho inclui operações de soma/subtração, multiplicação, divisão, raiz quadrada e funções trigonométricas como seno, cosseno e arco-tangente, operadores comumente encontrados na implementação de cinemática de manipuladores.

1.3.2 Objetivos Específicos

Os objetivos específicos propostos para este trabalho são:

1. Estudar, projetar simular e implementar diferentes algoritmos usados na implementação de operadores aritméticos básicos e transcendentais em hardware mediante o uso de arquiteturas reconfiguráveis.
2. Realizar um estúdio dos diferentes algoritmos implementados em quanto a área usada, desempenho, erro nos cálculos (precisão) e latência.
3. Mediante o uso da biblioteca aritmética desenvolvida, projetar, simular e implementar, direitamente em hardware, a cinemática direta de um robô manipulador de configuração esférica com 5 graus de liberdade.
4. Implementar a cinemática direta do manipulador em software mediante o uso de um processador embarcado PowerPC no FPGA.
5. Comparar o desempenho das implementações hardware e software.

1.4 Justificativa

Os primeiros trabalhos de implementação de aritmética de ponto flutuante foram projetados para primitivas FPGAs incluindo operações de soma e multiplicação [19, 20, 21]. Porém, estes trabalhos seminais mostram um excessivo custo do projeto em área. Os resultados obtidos nestes trabalhos mostram que implementar aritmética de precisão simples em FPGA era fatível mas pouco prático para a tecnologia disponível na época.

Nas primeiras implementações de unidades de ponto flutuante para FPGA observou-se uma grande quantidade de implementações dos operadores de soma/subtração e multiplicação. Por outro lado, foi dada pouca atenção a outros operadores tais como: divisão, raiz quadrada e funções transcendentais [22].

Modernas FPGAs possuem grandes quantidades de elementos lógicos, permitindo assim descrever em hardware cálculos complexos e novos algoritmos [23]. Recentemente, operações de aritmética de ponto flutuante usando precisão simples e dupla têm sido implementadas [24, 25, 26]. Mudar o tamanho da palavra nos operadores aritméticos é uma característica importante que permite satisfazer os requerimentos de precisão de acordo com a aplicação. Operadores aritméticos parametrizáveis só são possíveis em implementações baseadas em FPGAs, mas alem da precisão nos cálculo o projetista tem que se preocupar em evitar um sobre-dimensionamento do hardware.

Algumas bibliotecas parametrizáveis em aritmética de ponto flutuante são apresentadas em [24, 27]. Estas bibliotecas implementam operações de soma/subtração, multiplicação e raiz quadrada. Porém, estas não apresentam um estudo detalhado com respeito ao erro nos cálculos (precisão). Além disto, não se tem implementado operadores para o cálculo de funções transcendentais indispensáveis em aplicações em robótica (por exemplo, seno, cosseno e arco-tangente).

Uma biblioteca bastante comum entre os projetistas é apresentada em [28]. Esta biblioteca é apresentada como uma Unidade Lógica Aritmética (ULA), que inclui os operadores soma, subtração, multiplicação, divisão e raiz quadrada. Porém, estes operadores não podem ser usados para executar cálculos em paralelo, e não são parametrizáveis.

A Xilinx tem desenvolvido uma biblioteca comercial de operadores parametrizáveis em ponto flutuante incluindo soma/subtração, multiplicação, divisão, raiz quadrada, comparadores e conversores de ponto fixo a ponto flutuante. Os operadores de divisão e raiz quadrada são ser implementados usando arquiteturas *pipeline* (uma entrada pode ser aplicada em cada ciclo de relógio) apresentando um baixo consumo de recursos, porém, uma elevada latência. Por exemplo, em aritmética de precisão dupla as operações de divisão e raiz quadrada são calculadas em 56 ciclos de relógio (tempo de espera entre a entrada dos operandos e saída do resultado) [29].

Implementar uma biblioteca parametrizável em aritmética de ponto flutuante baseada no padrão IEEE-754 tem como contribuição facilitar o uso das FPGAs em variadas aplicações em engenharia, dando ao projetista as ferramentas para construir sua aplicação, de uma forma rápida e flexível. Pequenas modificações no tamanho da palavra exigem uma adaptação dos elementos internos de cada operador, o que levaria ao projetista gastar mais tempo na implementação.

Por outro lado, na literatura são encontrados poucos trabalhos que implementam a cinemática de manipuladores em FPGAs. Trabalhos prévios mostram arquiteturas de hardware para implementar controladores dos servo-motores utilizados em manipuladores, onde a cinemática direta e inversa é computada em software [30, 31, 32, 33]. Em [30, 31], uma arquitetura de hardware para controlar um robô manipulador SCARA é apresentada, implementando o controle dos servomotores em FPGAs e relevando o custo computacional aos GPPs.

Finalmente, considerando-se a importância que tem uma biblioteca parametrizável em aritmética de ponto flutuante em FPGAs, permitindo, assim, explorar o paralelismo intrínseco destes dispositivos em aplicações como cinemática de manipuladores, torna-se importante a implementação de uma arquitetura paralela para o cálculo da cinemática direta em aplicações que requerem altas velocidades de processamento com alta precisão nos cálculos.

1.5 Resultados Obtidos

Em este projeto foi desenvolvida uma biblioteca de operadores aritméticos em ponto flutuante parametrizáveis pelo tamanho da palavra. No intuito de automatizar a tarefa de mudar o tamanho da palavra de acordo com os requerimentos do projeto, foram desenvolvidos geradores de código escritos em MATLAB. Estes programas geram o código VHDL de cada operador de acordo com dados fornecidos pelo usuário em um *script*.

Para os operadores aritméticos de divisão e raiz quadrada foram propostas duas arquiteturas baseadas nos algoritmos Goldschmidt e Newton-Raphson. Da mesma forma foram propostas as arquiteturas baseadas no algoritmo CORDIC e expansão em séries de Taylor, para o cálculo de funções transcendentais seno, cosseno e arco-tangente.

Um análise de desempenho das arquiteturas propostas tendo em conta parâmetros como: (a) consumo de recursos do FPGA, (b) vazão (freqüência/latência) e (c) precisão, permitiu observar que a arquitetura baseada no algoritmo Newton-Raphson apresenta um melhor desempenho do que a arquitetura baseada no algoritmo Goldschmidt. Para o caso das funções transcendentais observa-se um melhor desempenho da arquitetura baseada em séries de Taylor para o cálculo das funções seno e cosseno. Embora para o cálculo da função arco-tangente a melhor escolha esta baseada no algoritmo CORDIC.

Finalmente, fazendo uso dos melhores algoritmos foi construída uma arquitetura para o cálculo da cinemática direta de um manipulador robótico de configuração esférica com cinco graus de liberdade. Esta arquitetura permitiu validar o funcionamento em conjunto dos operadores aritméticos em ponto flutuante e fazer um comparativo da implementação totalmente em hardware com duas implementações software. Uma das implementações em software é executada num processador *hard* PowerPC e a outra num computador usando MATLAB para executar os cálculos. Encontrou-se uma considerável aceleração no cálculo da cinemática direta em comparação com suas respectivas implementações em software.

1.6 Organização do Trabalho

O capítulo 2 apresenta a fundamentação teórica necessária para o desenvolvimento deste trabalho, abordando assuntos como representação numérica, representação numérica de números em ponto fixo e ponto flutuante, o padrão IEEE-754, cinemática direta de manipuladores e FPGAs. O capítulo 3 descreve os algoritmos de implementação de operadores aritméticos básicos e funções transcendentais. No capítulo 4 é apresentado o modelado matemático para obtenção da cinemática direta do robô manipulador de configuração esférica com cinco graus de liberdade e uma arquitetura para calcular a cinemática direta totalmente em hardware. O capítulo 5 mostra os resultados de sínteses e simulação dos operadores implementados e da arquitetura de hardware usada no cálculo da cinemática direta. Finalmente, no capítulo 6 são apresentadas as conclusões e trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Este capítulo apresenta a fundamentação teórica dos temas abordados no desenvolvimento deste projeto. Inicialmente, é tratada a representação de números em sistemas digitais, tais como a representação de números em ponto fixo e ponto flutuante baseados no padrão IEEE-754. Seguidamente é feita uma breve introdução à robótica de manipuladores, mostrando as características e classificação. Por outro lado, é feito um estudo focado, principalmente, na cinemática direita de manipuladores robóticos. Finalmente, é abordado o tema relacionado à tecnologia de hardware utilizada neste trabalho, principalmente na tecnologia de hardware reconfigurável baseada em FPGA (*Field Programmable Gate Array*).

2.1 Representação Numérica

Em cada aplicação deve ser escolhido cuidadosamente o tipo de representação numérica a usar. Neste caso, implementações em hardware usando ponto fixo têm como vantagens baixo consumo de recursos e altas velocidades de processamento. Por outro lado, aritmética de ponto flutuante abrange uma grande faixa dinâmica de representação de números reais, apresentando um maior consumo de recursos [17].

2.1.1 Representação em Ponto Fixo

Um dos formatos amplamente usados para representar valores numéricos é o formato em ponto fixo. O mesmo deriva seu nome da localização fixa de um ponto binário implícito entre a parte inteira e a parte fracionaria. Uma representação em ponto fixo é descrita, comumente, por dois números n e m, onde n representa a parte inteira e m parte fracionaria. Por exemplo, um formato 16.0 representa um inteiro de 16-bits. Por outro lado, um formato 3.2 (ver Fig. 2.1) representa um número em ponto fixo com um total de 5 bits, um bit reservado para o sinal, 2-bits na parte inteira e 2 bits a direita do ponto implícito (parte fracionaria). O maior valor representável em este formato é $011.11_b = 3,75_d$ e o menor valor é $111.11_b = -3,75_d$ [34].

A velocidade, baixo custo, e simplicidade faz com que implementações em ponto fixo sejam amplamente usadas. Porém, devido a sua faixa limitada de valores, este formato é inadequado

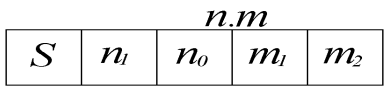


Figura 2.1: Representa em ponto Fixo

para a maioria de aplicações cientificas e de engenharia que precisam processar números muito grandes e muito pequenos [5, 35].

2.1.2 Representação em Ponto Flutuante

A representação em ponto flutuante está baseada na notação cientifica, na qual o ponto decimal é movimentado dinamicamente até uma posição desejada, usando um expoente para indicar a posição original do ponto decimal. Por exemplo, $976.000.000.000.000 = 9,76 \times 10^{14}$. Esta notação permite abranger tanto números grandes como pequenos utilizando poucos algarismos. Nesta notação tem-se que o número é representado pela equação 2.1

$$N = M \times B^E \tag{2.1}$$

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

A idéia da notação cientifica é estendida aos números binários, sendo conhecida também como representação binária em ponto flutuante. Esta representação é amplamente usada em aplicações computacionais que requerem de uma amplia faixa de representação numérica.

2.1.3 Padrão IEEE-754

O padrão IEEE-754 foi desenvolvido no intuito de padronizar a representação de números em ponto flutuante. No inicio da computação numérica, larguras de palavra de diferentes tamanhos eram criadas por cada fabricante; assim, ao executar um algoritmo em máquinas diferentes obtinha-se resultados diferentes [36].

Em 1958, no intuito de encerrar a incompatibilidade entre sistemas, a organização IEEE propôs o padrão IEEE-754, estabelecendo a representação e modo de operação de números representados em ponto flutuante. O padrão teve as seguintes diretrizes [36]:

- Facilitar a migração dos programas já existentes para os novos computadores que adotem este padrão.
- Permitir aos programadores, que tenham experiência ou não em métodos numéricos, produzir programas numéricos sofisticados.
- Motivar aos programadores experientes a produzir e distribuir programas numéricos eficientes, robustos e portáveis.
- Apoiar o diagnostico de anomalias e, facilmente, manipular exceções.

- Permitir o desenvolvimento de: (a) funções elementares tais como expoentes e cossenos, aritmética de alta precisão e (b) permitir um acoplamento entre cálculos numéricos e simbólicos.
- Permitir, em vez de impedir, novos aperfeiçoamentos e ampliações para a proposta.

O padrão IEEE-754 especifica:

- Formatos básicos e estendidos para números em ponto flutuante.
- Operações de soma, subtração, multiplicação, divisão, raiz quadrada, resto, e funções de comparação.
- Conversões entre inteiros e formatos em ponto flutuantes.
- Conversões entre diferentes formatos em ponto flutuantes.
- Conversões entre formatos básicos de números em ponto flutuante e cadeias de caracteres decimais (*strings*).
- Exceções e sua manipulação, incluindo NaN (Not a Number).
- Formatos de cadeias de bits decimais e inteiros
- Interpretação do sinal e campo de bit implícito da representação de NaN
- Conversões de binário para decimal e vice-versa.

O Formato IEEE 754

De acordo com o padrão IEEE-754 [36] um número real é representado por uma cadeia de bits caracterizados por três componentes: (a) um bit de signo S, (b) um expoente em excesso E com um largura de bits Ew e (c) uma mantissa M (parte fracionária) com uma largura de bits Mw (observar figura 2.2).

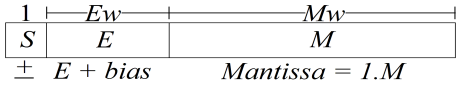


Figura 2.2: Formato IEEE-754

Um 0 no bit de sinal representa um número positivo e um 1 representaria um número negativo. O expoente tem seu próprio signo e é armazenado em excesso ($2^{q-1}-1$, onde q é o número de bits). Por exemplo, se o expoente tem 8 bits, a representação em excesso é 127. Assim, o valor verdadeiro representado no expoente é igual a o valor armazenado subtraindo o excesso. Por exemplo, para um excesso de 127, se expoente tive-se armazenado o valor de 9, o valor verdadeiro representado no expoente seria: -118 = (9 - 127).

A mantissa representa um número fracionário normalizado, o que quer dizer, que o bit mais a esquerda da mantissa é sempre 1. Como este bit no muda de valor, não é armazenado; mais,

implicitamente faz parte da representação (o bit implícito). A mantissa M se calcula segundo a equação 2.2

$$M = m_{22} \times 2^{-1} + m_{21} \times 2^{-2} + \cdots + m_1 \times 2^{-22} + m_0 \times 2^{-23} \tag{2.2}$$

O número real representado pela cadeia binária é obtido a partir da equação 2.3.

$$N = (-1)^s \times (1 + M) \times 2^{E-Bias} \tag{2.3}$$

Dentro do padrão IEEE-754 tem-se principalmente duas representações: (a) precisão simples e (b) precisão dupla. Na tabela 2.1 são resumidas as características destas representações.

Tabela 2.1: Principais representações no padrão IEEE 754

Precisão	Sinal	Expoente	Mantissa	Bais
Simples	1	8	23	127
Dupla	1	11	52	1023

Normalização do número sendo representado

Um número em ponto flutuante tem diferentes representações. Se o bit mais significativo da mantissa é diferente de zero a representação é dita normalizada [37]. Para evitar múltiplas representações para o mesmo valor, o mesmo deverá ser normalizado.

A operação de normalização em ponto flutuante consiste em deslocar a mantissa de modo que o bit más significativo seja igual a 1. Por cada deslocamento da mantissa o valor do expoente deve ser incrementado em uma unidade. Desta forma, somente o dígito 1 ficará na parte inteira (bit implícito), sendo esta informação redundante (só é necessário armazenar a parte fracionária). A única exceção a esta regra é o número zero, em que todos os bits (incluso o bit implícito) são iguais a zero [38]. Na tabela 2.2, pode-se observar o rango de representação de números normalizados usando aritmética de precisão simples.

Tabela 2.2: Número real normalizado

Sinal	Expoente	Mantissa
x	00000001 a 11111110	00000000000000000000 a 11111111111111111111

As últimas atualizações do padrão IEEE-754 incluem representação de números desnormalizados. Estes números têm o expoente em zero e a mantissa é usada para representar números entre zero e o menor número normalizado representado no formato. Em aritmética de precisão simples os números desnormalizados abrangem a faixa de valores entre (0, $1.1755E^{-38}$). Na tabela 2.3, é mostrado como é representado um número real desnormalizado usando aritmética de precisão simples.

Tabela 2.3: Número real desnormalizado

Sinal	Expoente	Mantissa
x	00000000	xxxxxxxxxxxxxxxxxxxxxxx

Valores especiais no formato IEEE 754

Neste formato são definidos valores especiais que permitem o tratamento de exceções, definindo códigos específicos dentro do formato para o posterior tratamento. Para definir estes valores específicos são usados os bits do expoente com todos seus campos em 1, com exceção do número zero [39].

O NaN (*Not a Number*) é usado para indicar operações inválidas. Uma operação é inválida se um dos operandos for inválido para a operação a ser executada. Por exemplo, soma ou subtração de valores de entrada $+\infty$ ou $-\infty$ infinito, multiplicação de $0 \times \infty$, divisão de $0/0$ ou ∞/∞, raiz quadrada de números menores que zero, entre outras.

Finalmente, o NaN tem duas representações: (a) QNaN (*Quiet NaN*) e (b) SNaN (*Signaling NaN*). O SNaN é caracterizado por ter o bit mais significativo da mantissa em 0, sendo é usado para indicar operações inválidas como as descritas anteriormente. O QNaN tem o bit mas significativo da mantissa em 1, e é usado em operações que não fornecem um resultado em ponto flutuante como nos casos de comparação e conversão de formatos. Na tabela 2.4, é mostrada a representação binária do QNAN e SNaN.

Tabela 2.4: Representação binária do QNAN e SNaN.

Sinal	Expoente	Mantissa	Valor
x	11111111	1xxxxxxxxxxxxxxxxxxxxxx	QNaN
x	11111111	0xxxxxxxxxxxxxxxxxxxxxx	SNaN

Os valores $\pm\infty$, são usados para indicar que houve uma divisão por zero, ou uma condição de *overflow* ou *underflow*. Estes são representados no formato como mostrado na tabela 2.5.

Tabela 2.5: Representação do Infinito.

Sinal	Expoente	Mantissa	Valor
0	11111111	00000000000000000000000	$+\infty$
1	11111111	00000000000000000000000	$-\infty$

Finalmente, o número zero tem duas representações no padrão IEEE-754 decorrentes da mudança do bit de sinal +0 e -0. Na tabela 2.6, é mostrada a representação do zero no formato IEEE 754.

Tabela 2.6: Representação do Zero.

Sinal	Expoente	Mantissa	Valor
0	00000000	00000000000000000000000	+0
1	00000000	00000000000000000000000	−0

2.2 Aspectos de Manipuladores Robóticos

Nos últimos anos a robótica passou de ser um mito da imaginação de autores literários a uma realidade amplamente aceita no mercado produtivo. Entre as aplicações comumente encontra-se que os robôs substituem o homem em tarefas repetitivas e hostis. Atualmente, podem ser encontrados em aplicações que incluem desde processos industrias a atividades diversas, tais como: segurança e vigilância, procura e salvamento, exploração planetária, exploração subaquática, entre outras aplicações.

Na área da robótica têm-se duas categorias principais, a saber: (a) robótica móvel e (b) robótica de manipuladores. O robô manipulador se caracteriza por ter a base fixa, e seu espaço de trabalho é limitado pelo alcance de seus elos. Por outro lado, os robôs moveis se caracterizam por sua capacidade de deslocamento em um ambiente [13].

De acordo com a ISO 8373, de 1994, um robô manipulador é definido como: "manipulador controlado automaticamente, reprogramável, de propósitos múltiplos, programável em três ou mais eixos, que podem ter base fixa ou móvel, para uso em aplicações industriais automatizadas"[40].

A configuração das juntas e a forma como são conectadas por meio dos elos definem o número de graus de liberdade em que o robô pode trabalhar, assim como a região dentro da qual o manipulador pode posicionar a ferramenta. Esta região é conhecida como espaço de trabalho do robô [13].

Os robôs podem ser classificados de acordo com a combinação entre juntas translacionais e rotacionais, assim como pela geometria do espaço de trabalho resultante. Dentre as combinações possíveis podem-se citar as 5 mais utilizadas do ponto de vista industrial: (a) cartesianos, (b) cilíndricos, (c) esféricos, (d) bi-cilíndrico ou SCARA e (e) articulado ou de revolução. Estas configurações são chamadas de clássicas e podem ser combinadas para dar lugar a novas configurações [13, 41].

2.2.1 O Robô Cartesiano

O robô cartesiano tem três articulações prismáticas atuando em cada eixo (x, y, z) (ver figura 2.3). Este se caracteriza pela alta rigidez mecânica e sua capacidade de operar com grande exatidão na localização do atuador, sendo que seu controle é simples devido ao seu movimento linear. A principal desvantagem é a sua pequena área de trabalho. Um robô com estas características é capaz de se localizar em qualquer ponto de um cubo.

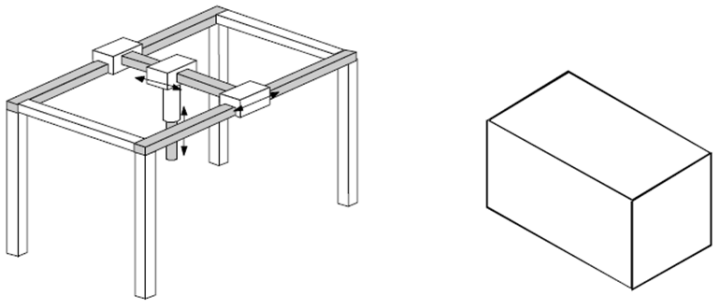


Figura 2.3: Robô Cartesiano e seu espaço de trabalho

2.2.2 O Robô Cilíndrico

Este robô se caracteriza por ter duas jutas prismáticas e uma rotacional, situadas como mostrado na figura 2.4. Possui um espaço de trabalho maior que a do robô cartesiano, se comparado ao volume ocupado pela sua estrutura, sacrificando, entretanto a rigidez mecânica, e elevando a complexidade no controle.

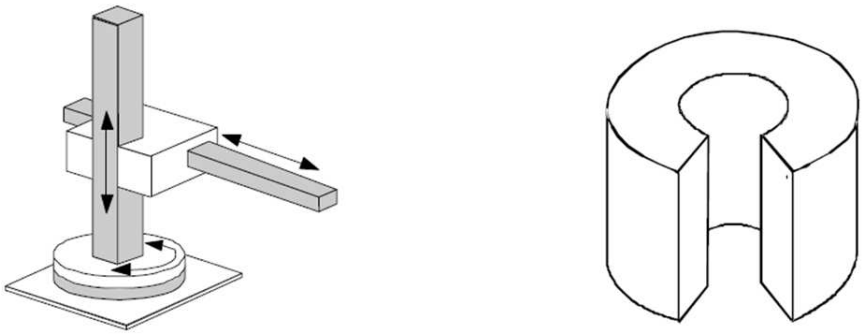


Figura 2.4: Robô Cilíndrico e seu espaço de trabalho

2.2.3 O Robô Esférico

O robô esférico consiste de duas juntas rotacionais, com eixos perpendiculares entre si, e uma junta prismática. As juntas rotacionais propiciam movimentos de azimute e de elevação e a junta prismática, de distância radial, resultando em um posicionamento representado por coordenadas polares (ver figura 2.5). A principal vantagem dos manipuladores que usam esta configuração é abranger um espaço de trabalho maior que seus pares cartesianos e cilíndrico, comparado ao volume ocupado pela estrutura do robô. Contudo, os robôs que usam esta configuração, apresentam a desvantagem de ter relativamente menor resolução e rigidez mecânica [41].

2.2.4 O Robô SCARA

O robô SCARA é formado por duas juntas rotacionais com eixos paralelos e uma junta prismática paralela aos eixos rotacionais, como mostrado na figura 2.6. Este robô tem um espaço de trabalho menor ao robô de configuração esférica comparando suas estruturas mecânicas. Entre-

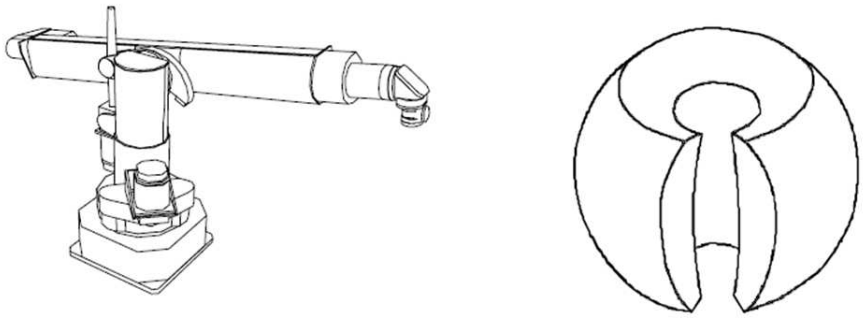


Figura 2.5: Robô Esférico e seu espaço de trabalho

tanto, o mesmo tem uma grande capacidade de manobrar no plano horizontal, alta aceleração e boa rigidez mecânica na direção vertical.

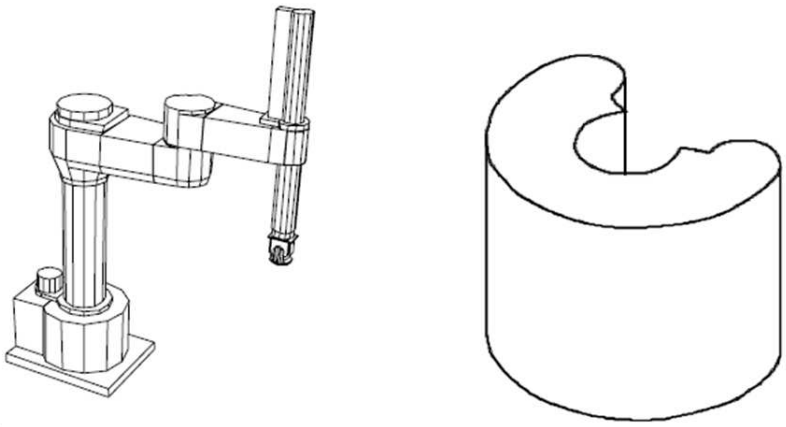


Figura 2.6: Robô SCARA e seu espaço de trabalho

2.2.5 O Robô articulado

Este robô é composto somente por juntas rotacionais dispostas como mostrado na figura 2.7. A principal vantagem desta configuração é seu grande espaço de trabalho comparado ao volume ocupado pelo robô. Entretanto, apresenta desvantagens como cinemática complexa, movimentos lineares difíceis de controlar e estrutura complacente.

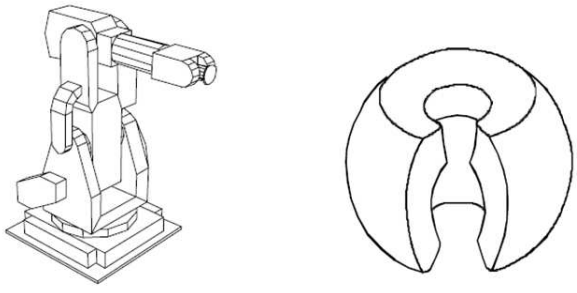


Figura 2.7: Robô Antropomórficoe seu espaço de trabalho.

2.3 Cinemática de Manipuladores

A cinemática de manipuladores tem por objetivo relacionar a posição e orientação do extremo final do robô com os valores assumidos pelas suas coordenadas articulares, existindo dois problemas fundamentais a serem resolvidos: (a) o problema cinemático direto e (b) o problema cinemático inverso.

No problema cinemático direto a posição e orientação do órgão terminal do robô, com respeito a um sistema de coordenadas fixo, são determinadas a partir dos valores das articulações e da configuração geométrica do robô. No problema cinemático inverso, pretende-se encontrar os valores das articulações para localizar o elemento terminal em uma posição e orientação desejada [13, 12].

Este trabalho tem por objetivo a implementação de uma biblioteca de operadores aritméticos em ponto flutuante para serem embarcados no FPGA, que permita calcular operações comumente encontradas na cinemática de manipuladores. Estes operadores serão usados na implementação totalmente em hardware do problema cinemático direto. A continuação será tratada com maior detalhe a cinemática direta de manipuladores.

2.3.1 Cinemática direta de manipuladores

Um clássico, porém importante problema na área de robótica industrial é a cinemática direta. A cinemática direta permite a localização cartesiana do elemento terminal do robô a partir da medida dos ângulos ou deslocamentos das juntas [13, 12, 41]. Este procedimento baseia-se, fundamentalmente na álgebra matricial e vetorial.

Um robô poderia ser considerado como uma cadeia cinemática conformada por uma série de elos que conectam o elemento terminal à base. Cada elo é conectado ao próximo por uma junta atuada [13, 12]. Associando um sistema de coordenadas para cada elo, a relação entre dois elos consecutivos pode ser descrita por uma matriz de transformação homogênea A. A seqüência de matrizes A relaciona o elemento terminal à base (ver equação 2.4) [13].

$$^{R}T_{H} = A_1 A_2 \cdots A_{n-1} A_n \tag{2.4}$$

Desta forma o problema cinemático direto é resumido em obter uma matriz de transformação homogênea T que localize (posição e orientação) do elemento terminal do robô, com referência ao sistema de coordenadas fixado em sua base. Os elementos desta matriz são funções dos ângulos e deslocamentos entre os elos, que podem ser fixos (parâmetros geométricos) ou variáveis (variáveis das juntas do manipulador) [13, 12].

Para determinar a relação de um ponto no espaço cartesiano da base, em função de um ponto no espaço das juntas é seguido o seguinte procedimento [13]:

1. Mover o robô à posição zero.
2. Atribuir um sistema de coordenadas para cada elo.
3. Descrever as relações (translações e rotações) entre os sistemas de coordenadas de cada elo.

4. Definir as matrizes A_i de localização relativa entre os sistemas de coordenadas dos elos.

5. Multiplicar as matrizes para calcular a matriz de transformação homogênea do manipulador ${}^{R}T_{H}$.

6. Igualar a matriz de transformação homogênea do manipulador e a matriz de transformação geral a qual é obtida por meio das coordenadas cartesianas da posição que se quer atingir no espaço (posição almejada), relativas ao sistema de coordenadas fixado na base do robô, e da orientação de um sistema de coordenadas fixado na mesma posição almejada descrito por uma seqüência de três rotações em torno dos eixos do sistema de coordenadas da base.

2.3.1.1 A Posição zero *(Home position)*

A posição zero de um manipulador é a posição onde todas as articulações variáveis do manipulador são zero. As articulações rotacionais estão em sua posição zero quando o eixo x dos marcos de coordenadas dos enlaces sejam paralelos e tenham a mesma direção (ver figura 2.8). Uma junta prismática está na sua posição zero quando a distancia entre os enlaces é mínima (recolhida).

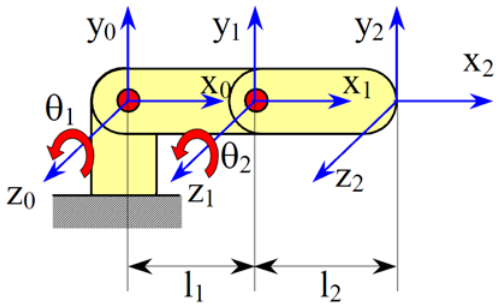


Figura 2.8: Posição zero para juntas rotacionais.

Para atribuir um sistema de coordenadas a cada elo são seguidas as convenções de Denavit e Hartenberg (D-H), por meio das quais dois elos consecutivos são relacionados por meio de 4 transformações básicas que só dependem da geometria do robô [13].

As transformações são:

1. Uma rotação ao redor do eixo z_{n-1} (eixo da junta $n-1$) por o ângulo formado entre as direções das normais comuns a dois elos consecutivos (elo $n-1$ e elo n)(θ_n).

2. Uma translação (transversal) ao longo do eixo da junta z_{n-1} da distância entre as juntas (d_n). Esta é obtida a partir da distância entre os pontos de cruzamento, no eixo z_{n-1}, das normais comuns aos elos $n-1$ e n.

3. Uma translação, ao logo da normal comum do elo n, do comprimento do elo (l_n). A direção da normal comum do elo n define a direção do eixo x_n.

4. Rotação em torno do eixo x_n por um ângulo α_n com o objetivo de modificar a direção do eixo z, inicialmente paralela ao eixo z_{n-1} da junta $n-1$, de modo a torná-lo paralelo a direção do eixo da junta n (z_n).

A multiplicação das transformações acima resulta na matriz A que define a posição relativa entre os sistemas de coordenadas atribuídos a elos consecutivos, conforme mostra a equação 2.5 [13].

$$
\begin{aligned}
A_n &= R(z,\theta_n)T(0,0,d_n)T(l_n,0,0)R(x,\alpha_n) \\
&= \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & l \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) & 0 \\ 0 & \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} \cos(\theta) & -\sin(\theta)\cos(\alpha) & \sin(\theta)\sin(\alpha) & l\cos(\theta) \\ \sin(\theta) & \cos(\theta)\cos(\alpha) & -\cos(\theta)\sin(\alpha) & l\sin(\theta) \\ 0 & \sin(\alpha) & \cos(\alpha) & d \\ 0 & 0 & 0 & 1 \end{bmatrix}
\end{aligned}
\tag{2.5}
$$

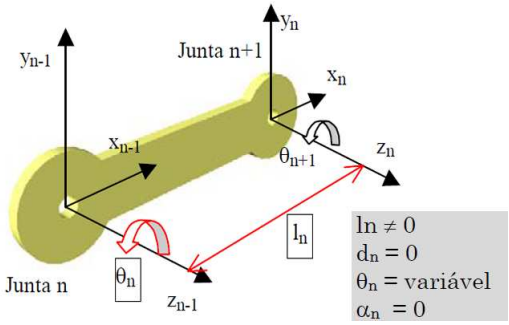
Algumas configurações de elos típicos e o comportamento dos parâmetros d_n, l_n and α_n são mostrados na figura 2.9.

θ_n, d_n, l_n and α_n representam os parâmetros de D-H do elo n. Identificando estes parâmetros a matriz A que relaciona dois elos consecutivos pode ser obtida. Finalmente, multiplicando a seqüência de matrizes A (ver equação 2.4) a matriz de transformação homogênea T é obtida. Esta, quando comparada à matriz de transformação homogênea resultante da localização almejada (matriz numérica), possibilita a obtenção da chamada cinemática direta do manipulador, a qual relaciona as coordenadas cartesianas da localização almejada como funções das coordenadas das juntas e dos parâmetros dos elos do manipulador [13].

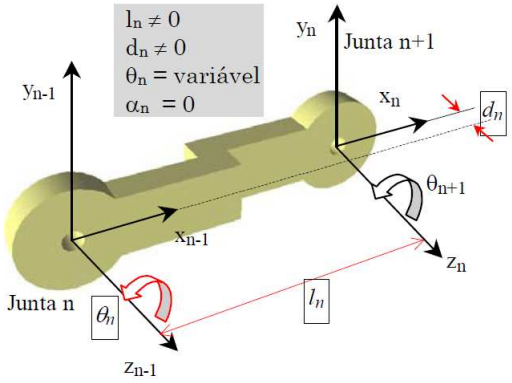
2.4 Computação Reconfigurável

O paradigma baseado no fluxo de instruções de Von Neumann tem perdido seu domínio como modelo básico e único de sistemas computacionais. Hoje é comum que muitas aplicações desenvolvidas em software sejam migradas para aceleradores de hardware. Para isto, podem ser usados Circuitos Integrados para Aplicações Específicas (ASICs) ou sistemas reconfiguráveis baseados em FPGAs. Na figura 2.10 é mostrada a arquitetura de um acelerador baseado em um ASIC.

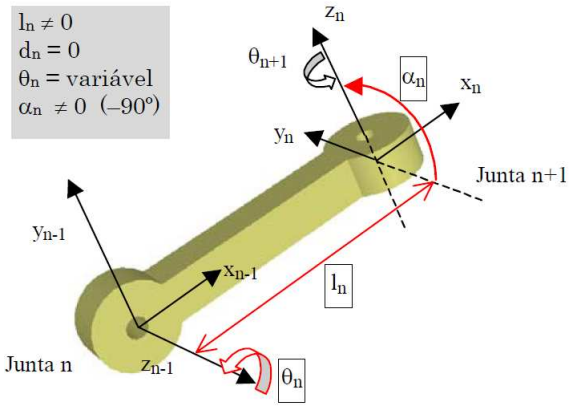
As FPGAs são dispositivos que oferecem a potencialidade de dispositivos ASICs , tais como: (a) redução de tamanho, peso, e consumo de potência, (b) alto desempenho, (c) maior segurança contra copias não autorizadas e (d) baixo custo. Além do anterior, FPGAs superam os ASICs em tempo de prototipagem, reprogramação do circuito, menor custo NRE (Non-recurring engineering), tendo como resultado projeto mais econômicos para soluções que precisem poucas quantidades de circuitos. Por outro lado, comparando com DSPs (Digital signal processing) e processadores de propósito geral, implementações em FPGAs podem explorar mais amplamente o paralelismo intrínseco dos algoritmos [17].



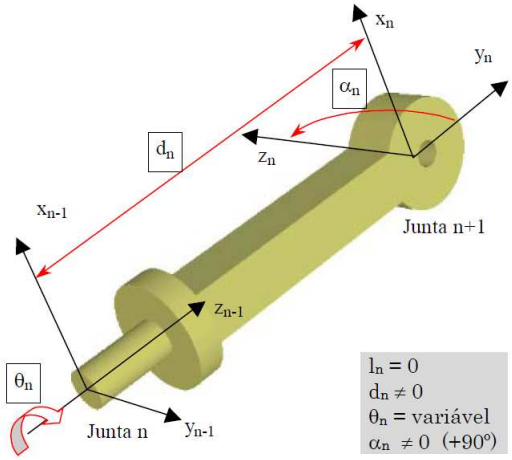
(a) Juntas Rotacionais Paralelas



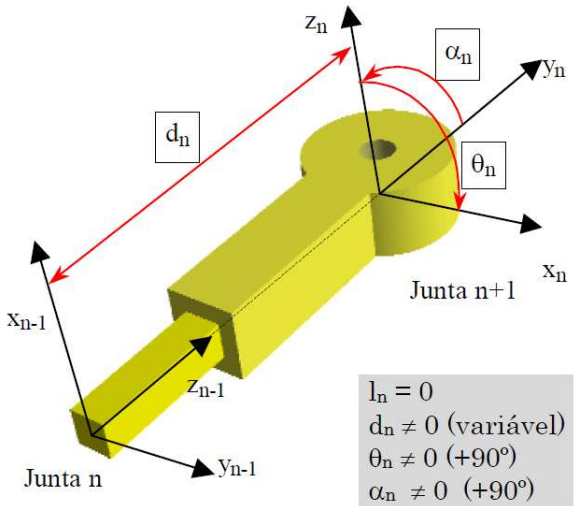
(b) Juntas Rotacionais Paralelas com deslocamento



(c) Juntas Rotacionais Ortogonais



(d) Juntas Rotacionais Ortogonais Tipo 2



(e) Junta Rotacional e Prismática, Ortogonais

Figura 2.9: Algumas configurações de elos típicos.

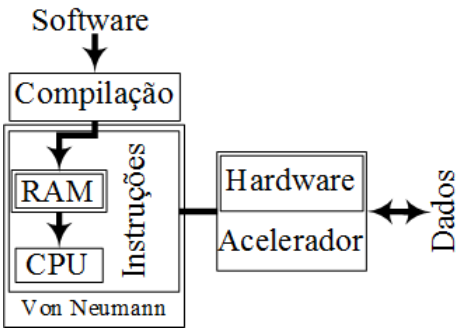


Figura 2.10: Acelerador em sistema embarcado (Modificado [2])

2.4.1 Arquitetura interna de um FPGA

Entre as arquiteturas de FPGAs mais importantes encontramos: (a) arquiteturas *island* (Xilinx), (b) arquiteturas hierárquicas (Altera) e (c) arquiteturas row-based (Actel) [42]. Neste trabalho, será tratada com mais detalhe a arquitetura *island* da Xilinx, já que, as implementações feitas no desenvolvimento deste trabalho foram realizadas principalmente nestes dispositivos.

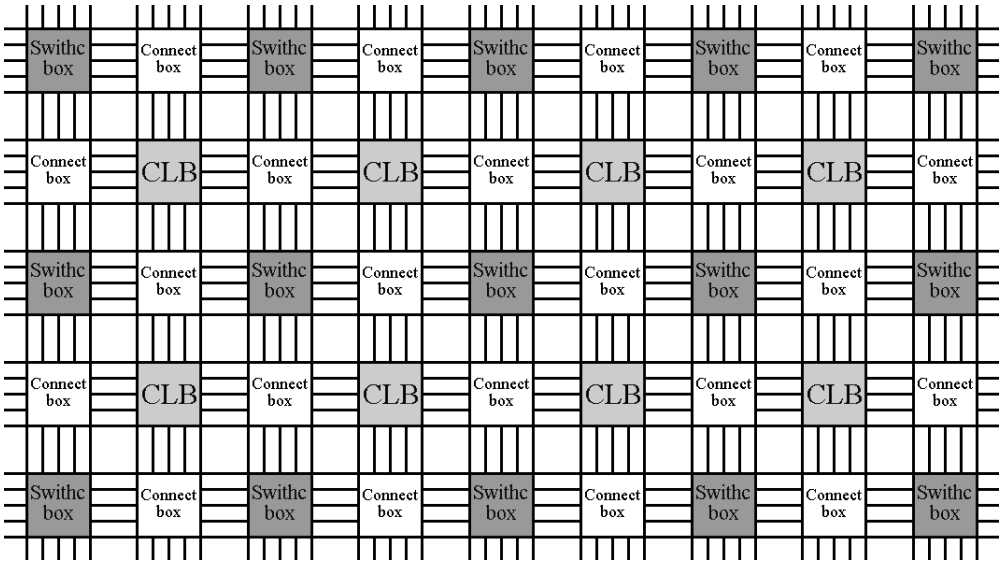


Figura 2.11: FPGA de granularidade fina baseada numa arquitetura island

Um FPGA é, basicamente, formado por uma matriz de Blocos Lógicos Configuráveis (CLB - *Configurable Logic Blocks*) incorporados em interconexões reconfiguráveis (ver figura 2.11). O código $configware$ (código de reconfiguração) é armazenado em memórias RAM distribuídas. Uma vez que a FPGA é ligada, o código $configware$ é carregado desde memórias RAM externas, geralmente de tipo FLEX.

A maioria de CLBs são baseados em LUTs (*Look Up Tables*), na figura 2.12 é mostrado um diagrama de blocos da arquitetura interna de um CLB. Seu principio de funcionamento está baseado na multiplexação de funções previamente armazenadas na LUT, como mostrado na tabela 2.7. Os elementos de roteamento do sistema (*switchbox* e os *connectbox*) servem para ligar os diferentes CLBs, sendo chamados de recursos de interconexão (ver figura 2.13). A configuração total

de todas as interconexões do dispositivo são realizadas por ferramentas de projeto assistido por computador (CAD), este procedimento é conhecido como posicionamento e roteamento (*Placement and Routing*).

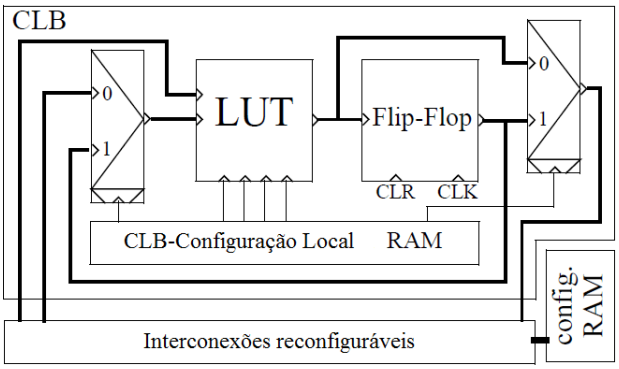


Figura 2.12: Diagrama de blocos de um CLB (Modificado [2])

Tabela 2.7: funções armazenadas numa LUT

Bits de configuração local				f(A,B)	Função
g3	g2	g1	g0		
0	0	0	0	0	Constante 0
0	0	0	1	A and B	and
0	0	1	0	B não implica A	if B then 0 else A
0	0	1	1	A	Identidade A
0	1	0	0	A não implica B	if A then 0 else B
0	1	0	1	B	Identidade B
0	1	1	0	A Xor B	Xor
0	1	1	1	A or B	or
1	0	0	0	not(A or B)	nor
1	0	0	1	A igual B	Igualdade
1	0	1	0	not(B)	Negação de B
1	0	1	1	B implica A	if A then 1 else ¬B
1	1	0	0	not(A)	Negação de A
1	1	0	1	A implica B	if B then 1 else ¬A
1	1	1	0	not(A and B)	nand
1	1	1	1	1	Constante 1

2.4.2 Granularidade de Sistemas Reconfiguráveis

Outro parâmetro importante em sistemas reconfiguráveis é a granularidade do sistema. Este fator é definido como o tamanho da menor unidade funcional que pode ser acessada por uma ferramenta de mapeamento (tamanho de grão). Como exemplo de granularidade pode-se citar o CLB que representa a menor unidade de configuração.

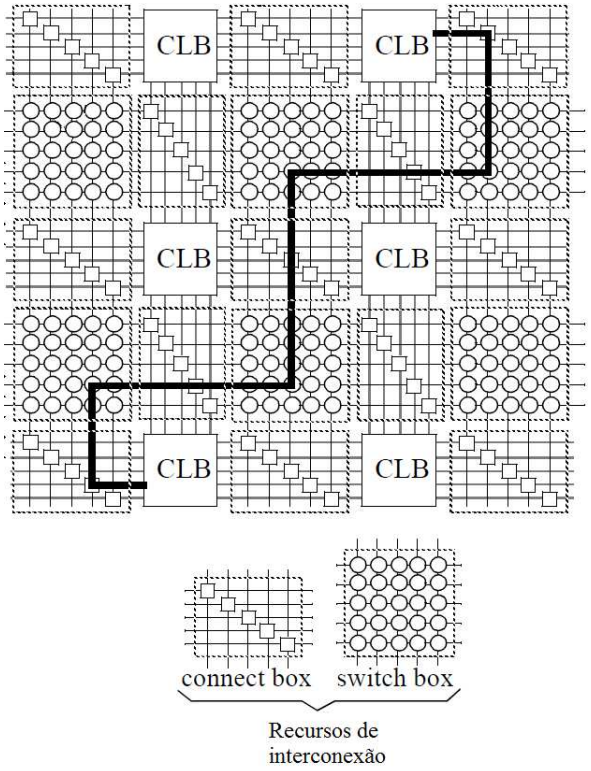


Figura 2.13: conexão de elementos internos de uma FPGA (Modificado [2])

Entre os dispositivos reconfiguráveis encontra-se os de granularidade fina e os de granularidade grossa. Os dispositivos de granularidade fina apresentam um grande número de elementos simples que podem ser interconectados para a formação de subsistemas digitais. Em sistemas complexos há um elevado grau de utilização destes elementos e, portanto, apresenta-se uma dificuldade na tarefa de roteamento. Adicionalmente, é necessário um número grande de dados de configuração (a configuração/reconfiguração é feita no nível de bits) para as unidades de processamento, o qual exige uma memória maior para o armazenamento dos dados de configuração, acarretando conseqüentemente, uma maior dissipação de potência [42].

Nos dispositivos de granularidade grossa os blocos de construção são maiores, tipicamente ULAs (Unidades lógico Aritméticas), microprocessadores, DSPs, e blocos de memória. Como pode-se ver estes dispositivos apresentam maior quantidade de lógica ou poder computacional nas unidades funcionais. Neste caso, a programação do FPGA é feito no nível de palavras (cadeias de caracteres) e não de bits (como no caso de dispositivos de granularidade fina).

2.4.3 Processadores embarcados em FPGAs

A elevada densidade de portas lógicas disponíveis em FPGAs atuais permite que hoje seja possível incorporar dentro destes dispositivos processadores que coexistem no mesmo chip. Alguns fabricantes oferecem processadores que diretamente estão incorporados dentro da FPGA ($hardIP$), e/ou processadores que podem ser baixados para a FPGA ($SoftIP$). O propósito de se ter um processador incorporado dentro da FPGA é prover uma base flexível combinando a potencialidade do software e hardware em um só dispositivo de silício [3].

Na tabela 2.8, são mostrados alguns processadores disponibilizados pelos fabricantes para serem usados com seus respectivos dispositivos.

Tabela 2.8: processadores hard e Soft disponibilizados por os fabricantes (Modificado [3])

Processador	Tipo	Barramento	Fabricante
MicroBlaze	Soft 32	IBM Coreconneect	Xilinx
Nios	Soft 32	Avalon	Altera
LatticeMico32	Soft 32	Wishbone	Lattice
CoreMP7	Soft 32	APB	Actel
ARM Cortex-M1	Soft 32	AHB	Distribuidor Independente
LatticeMico8	Soft 8	Input/Output ports	Lattice
Core8051	Soft 8	Nil	Actel
Core8051s	Soft 8	APB	Actel
PicoBlaza	Soft 8	Input/Output ports	Xilinx
PowerPC	Hard 32	IBM Coreconnect	Xilinx
AVR	Hard 8	Input/Output ports	Atmel

Processadores *Soft*

Um processador *soft* é um microprocessador completamente sintetizável para depois ser implementado usando blocos lógicos e demais recursos disponíveis no FPGA. Neste caso, encontram-se processadores *soft* de 8 e 32 bits.

Dentre os processadores *soft* de 8 bits mais comuns encontra-se o PicoBlaze da Xilinx. Adicionalmente, para o caso de 32 bits encontra-se o NIOS da Altera e o MicroBlaze da Xilinx. Estes processadores usam uma parte fixa da área FPGA, deixando outra parte para incorporar lógica digital (mediante o processo de reconfiguração), fazendo possível aproveitar o paralelismo intrínseco dos algoritmos a serem mapeados em hardware.

Processadores *Hard*

Muitos dos processadores e microcontroladores clásicos têm suas versões disponibilizadas em FPGAs (como versões *Hard*). Estes processadores fazem parte da arquitetura interna do FPGA da mesma forma que um CLB. Entre alguns destes processadores pode-se salientar o microcontrolador AVR (processador de 8 bits) oferecido pela Atmel, e o PowerPC disponível nas famílias Virtex da Xilinx. Como exemplo cita-se o FPGA Virtex II Pro XC2VP30 que inclui dois processadores PowerPC internamente na FPGA.

Finalmente, na tabela 2.9, são mostradas as características das últimas famílias de dispositivos FPGA lançados no mercado pela Xilinx, uma das empresas da área com maior aceitação no mercado. A maioria das FPGAs destas famílias tem embarcados processadores PowerPC.

Tabela 2.9: processadores hard e Soft disponibilizados por os fabricantes (Modificado [3])

Característica	Virtex-6	Virtex-5	Spartan-6	ExtendedSpartan-3A
Blocos lógicos	Até 760 K	Até 330K	Até 150K	Até 53K
Pinos de I/O para uso geral	Até 1200	Até 1200	Até 570	Até 519
Protocolos I/O suportados	Acima de 40	Acima de 40	Acima de 40	Acima de 20
Tecnologia do Relógio	PLL	DCM + PLL	DCM + PLL	DCM
Blocos RAM Embarcados	Até 38 Mbits	Até 18 Mbits	Até 4.8 Mbits	Até 1.8 Mbits
Multiplicadores Embarcados	25 x 18 MAC	25 x 18 MAC	18 x 18 MAC	18 x 18 MAC
Suporte PCI Express	Gen 1, x8, hard Gen 2, x8, hard	Gen 1, x8, hard Gen 2, x8, soft	Gen 1, x1, hard	Não
Suporte Soft MicroBlaze	Sim	Sim	Sim	Sim
Processador Hard PowerPC	Sim	Sim	Sim	Não

2.5 Conclusões do Capítulo

Neste capítulo foram tratadas algumas definições consideradas relevantes para o desenvolvimento deste projeto. Inicialmente, foram introduzidos dois formatos de representação de números em dispositivos digitais as saber, (a) formato de ponto fixo e (b) ponto flutuante. A principal vantagem do formato em ponto flutuante é a capacidade de representar números muito grandes e muito pequenos numa cadeia de bits de tamanho reduzido, garantindo alta precisão e um amplo rango dinâmico na representação. Para tanto, o padrão IEEE 754 de representação de números em ponto flutuante em cadeias de bits finitas foi descrito.

O clássico pero importante problema na área da robótica conhecido como cinemática direta, que relaciona a posição e orientação do elemento terminal do robô com respeito aos valores tomados nas articulações foi apresentada. Claramente, pode se observar que esta formulação precisa de um elevado número de operações aritméticas e trigonométricas que idealmente deveriam ser executadas usando aritmética de ponto flutuante, no intuito de ter maior precisão.

Atualmente, um dos objetivos no campo da robótica é aumentar a velocidade de operação (desempenho dos controladores), desta forma obtendo benefícios em térmos de eficiência da manufatura, precisão e tempo de processamento. Portanto, abordou-se neste capítulo a tecnologia de computação reconfigurável baseada em FPGAs . Finalmente, foi descrita a arquitetura interna de um FPGA, suas vantagens e os últimos dispositivos lançados no mercado com centos de milhões de comportas lógicas (os quais permite utilizar processadores embarcados, mesmo tendo arquiteturas complexas de 32 bits).

O objetivo principal deste trabalho é implementar a cinemática direta de um robô manipulador de configuração esférica em FPGA, espera-se com esta implementação ganhar, principalmente, no tempo de processamento.

Capítulo 3

Algoritmos Usados na Implementação de Operadores Aritméticos em Ponto Flutuante em FPGA

Operações aritméticas em ponto flutuante são um requisito essencial em um amplo tipo de aplicações computacionais e de engenharia que precisam de um bom desempenho e alta precisão. Dentro de estas aplicações encontram-se o cálculo da cinemática direta e inversa na área de robótica, processamento de imagens, processamento digital de sinais, sistemas de controle, entre outras. Este tipo de aplicações freqüentemente executam um grande número de operações aritméticas e trigonométricas que deveriam ser executadas de uma forma eficiente.

Os contínuos avanços em tecnologia de fabricação de circuitos integrados VLSI incrementou, rapidamente, a densidade de integração, permitindo ao projetista implementar diretamente em hardware diferentes funções comumente implementadas em software. Por tanto, é possível pensar que hoje as FPGAs superaram aplicações envolvendo operações em nível de bits (incluindo operações lógicas, e também aritméticas em ponto fixo), tornando possível implementações que precisem cálculos numéricos de alta precisão executados, por exemplo, em ponto flutuante.

Este capítulo inicia com uma descrição de trabalhos que implementam bibliotecas em ponto flutuante em FPGA, depois, é feita uma descrição de algoritmos usados na implementação de operadores aritméticos básicos como soma/subtração, multiplicação divisão e raiz quadrada. Em seguida, são apresentados algoritmos usados na implementação de funções trigonométricas como seno, cosseno tangente e arco-tangente. Finalmente, são apresentadas as conclusões do capítulo.

3.1 Trabalhos Correlatos à Implementação de Operadores Aritméticos de Ponto Flutuante em Hardware

Os primeiros trabalhos em aritmética de ponto flutuante foram projetados para FPGAs de granularidade fina, tendo como objetivo a implementação de operadores de soma/subtração e multiplicação. Porém, dado o reduzido número de elementos lógicos nas FPGAs disponíveis na

época, estas implementações apresentaram um elevado custo em área. Estes primeiros resultados mostraram que implementar aritmética de precisão simples, seguindo o padrão IEEE-754 era viável, porém pouco prático para a tecnologia de FPGAs disponíveis na época [19, 20, 21].

FPGAs modernos têm uma grande quantidade de elementos lógicos, que permitem embarcar complexas operações e novos algoritmos [23]. Recentemente, operadores aritméticos em ponto flutuante têm sido implementados tanto para precisão simples (32 $bit-width$), como para precisão dupla (64 $bit-width$) [24, 25, 26]. Contudo, muitas das aplicações em engenharia requerem larguras de palavra diferentes, que dificilmente são encontradas em dispositivos de arquitetura fixa. De esta maneira, uma biblioteca aritmética em ponto flutuante parametrizável pela largura de palavra (bit-width) poderia ser implementada em FPGAs aproveitando sua flexibilidade. Por exemplo, uma destas aplicações é encontrar uma representação que garanta a estabilidade de um controlador digital, tendo em conta parâmetros do projeto como consumo de elementos lógicos e o desempenho do controlador [14].

As referências [5, 43] apresentam uma expansão em series de Taylor para calcular divisão e raiz quadrada. Em [43] é realizada uma comparação entre expansão em series de Taylor contra o processador Intel Pentium 4 e o AMD K7. Os processadores Intel Pentium 4 e AMD K7 usam algoritmos SRT e Goldschmidt, respectivamente, nas suas implementações de divisão e raiz quadrada em ponto flutuante. A referência [5] descreve uma implementação parametrizável baseada em series de Taylor. Neste trabalho são mostrados resultados de consumo de recursos para diferentes larguras de palavra, além de explorar capacidades de paralelismo usando estas implementações em processamento digital de imagens.

Implementações hardware de diferentes arquiteturas para operadores de divisão e raiz quadrada foram implementados em [10, 44]. Em [10] são apresentadas implementações em hardware pipelined e iterativas do algoritmo SRT. A referência [45] apresenta uma análise de diferentes técnicas tais como algoritmos diretos, algoritmos *non-restoring*, SRT e aproximações baseadas em Newton-Raphson e CORDIC. As referências [27, 24] apresentam bibliotecas parametrizáveis, onde são usados os algoritmos Radix-4 e Radix-2 SRT na implementação dos operadores de divisão e raiz quadrada, respectivamente.

Por outro lado, implementações de operadores aritméticos para o cálculo de funções trigonométricas tais como seno, cosseno tangente e arco-tangente têm sido amplamente estudadas. Muitas aplicações usando aritmética de ponto fixo são baseadas no algoritmo CORDIC [8, 46, 47, 48]

Implementações do algoritmo CORDIC em ponto flutuante são apresentadas em [49]. Nesse trabalho uma implementação de um processador CORDIC usando precisão dupla foi apresentado. Um modelo hibrido utilizando uma redução de argumento e ângulos de rotação foram desenvolvidos tentando reduzir o consumo de hardware. Em [50] foi apresentado um HOTBM para calcular funções aritméticas em ponto flutuante usando FPGA.

Pode-se observar que os trabalhos prévios mostram que é possível implementar operadores aritméticos em ponto flutuante sobre FPGAs. Porém, o projetista enfrenta-se a um grande desafio tendo que lidar com o compromisso entre consumo de elementos lógicos, latência e precisão nos cálculos, cujos requisitos variam de acordo com a aplicação.

3.2 Algoritmos de operadores aritméticos básicos em ponto flutuante

3.2.1 Soma/Subtração

O algoritmo mostrado na figura 3.1 implementa o operador de soma/subtração em ponto flutuante.

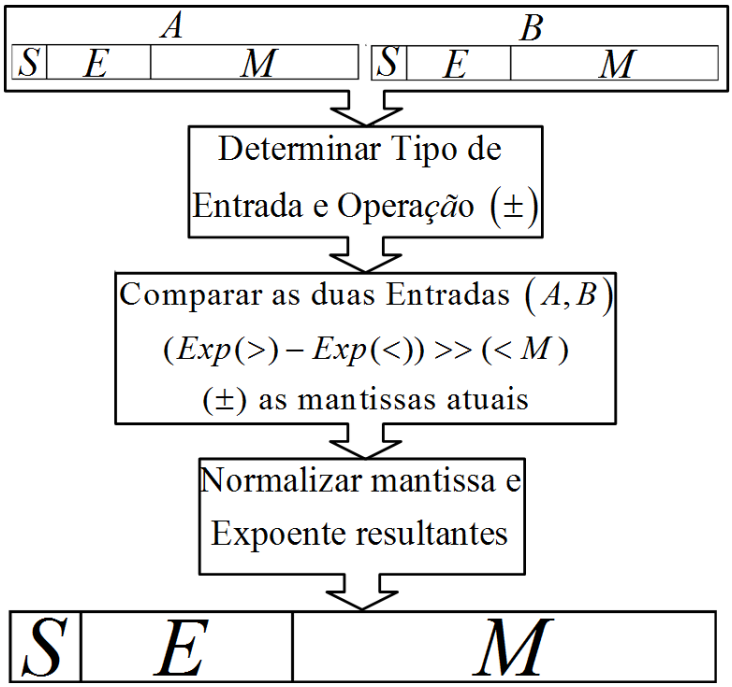


Figura 3.1: Passos seguidos na implementação de soma/subtração em ponto flutuante.

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 diferencia entre o expoente maior e menor. Posteriormente, é determinado o tipo de operação a ser executada (±). Subseqüentemente, expoente e a mantissa atuais são normalizadas. O algoritmo finaliza com a concatenação do sinal, expoente e mantissa resultantes.

3.2.2 Multiplicação

Os passos seguidos para implementar o operador de multiplicação são mostrados na figura 3.2.

Este algoritmo primeiro valida os dados de entrada, depois separa em registradores específicos o sinal, expoente e mantissa de cada operando, adicionando o bit implícito em cada mantissa. Posteriormente, em paralelo são executadas as seguintes operações: (a) determinar o produto dos sinais, (b) adicionar os expoentes subtraindo o *bias* e (c) multiplicar mantissas. Finalmente, se o MSB (*Most Significant Bit*) da mantissa resultante da multiplicação for 1 não é necessário fazer normalização. Em caso contrario, a mantissa resultante é deslocada a esquerda até encontrar o primeiro 1. Por cada operação de deslocamento, o expoente resultante é decrementado em 1.

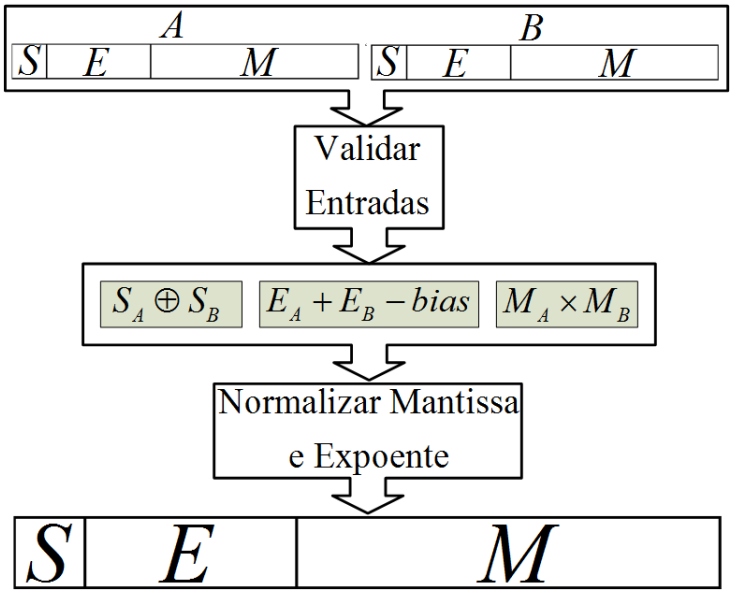


Figura 3.2: Passos seguidos na implementação da multiplicação em ponto flutuante.

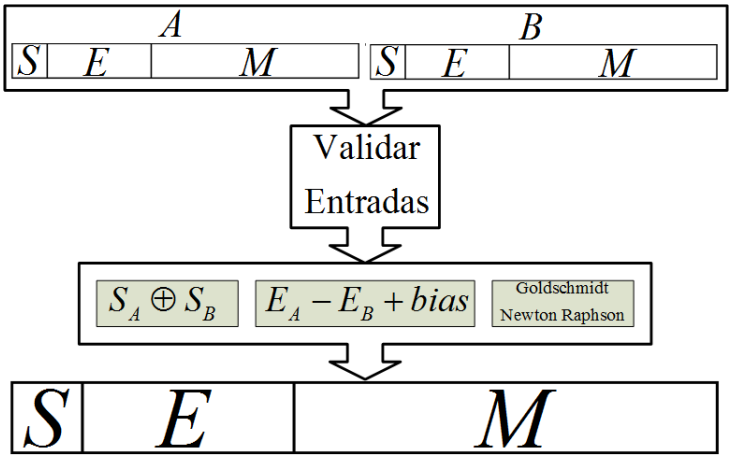


Figura 3.3: Passos seguidos na implementação da divisão em ponto flutuante

Uma vez, realizado o processo de normalização são concatenados o sinal, o expoente e a mantissa, finalmente resultantes.

3.2.3 Divisão

Na literatura existem diferentes algoritmos usados na implementação do operador aritmético de divisão, em esta seção é mostrado um algoritmo geral para o cálculo deste operador (ver figura 3.3) [35], para finalmente descrever os algoritmos Goldschmidt e Newton - Raphson implementados no desenvolvimento deste trabalho.

Sejam A e B dois números reais representados no padrão IEEE-754, onde 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 3.1, (c) determinar o produto dos sinais e calcular a mantissa resultantes usando um dos algoritmos que posteriormente serão descritos, e finalmente, (d) concatenar o sinal, expoente e mantissa resultantes.

$$E_A - E_B + bias \tag{3.1}$$

Algoritmo Goldschmidt para divisão

Este algoritmo tem como parâmetros de entrada as mantissas normalizadas do dividendo e do divisor, que implicitamente satisfazem que $M_N \geq 1$ e que $M_D \leq 2$. O algoritmo Goldschmidt calcula o cociente, partindo de uma semente aproximada ao valor $1/M_D$, refinado através de sucessivas multiplicações da semente vezes o numerador [51]. As implementações feitas neste trabalho foram baseadas no algoritmo apresentado em [52], cujos passos são os seguintes:

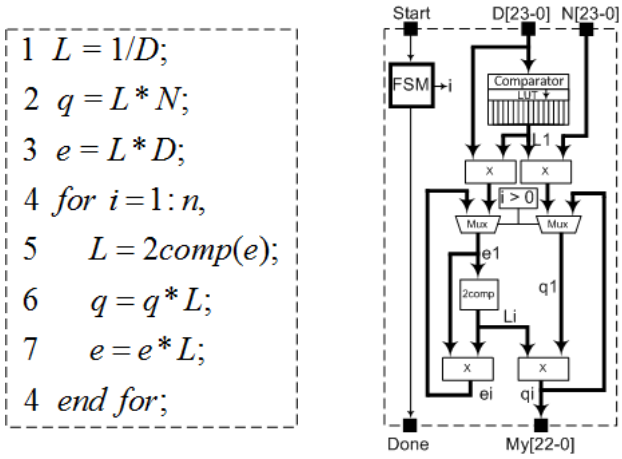
1. Mover o ponto do numerador (M_N) e do denominador (M_D) de forma tal que $M_N \geq 1$ e que $M_D \leq 2$.
2. Procurar em uma *look-up table* a aproximação inicial $1/M_D$ e nomear esta como L_1.
3. Fazer à primeira aproximação igual a $q_1 = L_1 \times M_N$ e calcular $e_1 = L_1 \times D$.
4. Iniciar iterações fazendo $L_2 = -e_1$.
5. Calcular $e_2 = e_1 \times L_2$ e $q_2 = q_1 \times L_2$.
6. Calcular $L_3 = -e_2$ similarmente ao passo 4, e continuar o processo com sucessivas iterações.

Depois de cada iteração do algoritmo e_i aproxima-se a 1, como produto das sucessivas multiplicações do denominador vezes $1/M_D$. Em q_i é armazenado um valor muito aproximado ao cociente verdadeiro.

Na figura 3.4 é apresentada a implementação deste algoritmo tanto em hardware como em software. Na arquitetura do hardware as aproximações iniciais a $1/M_D$ são armazenadas em uma *look-up table*. As sucessivas iterações para o refinamento da aproximação inicial são controladas por uma máquina de estados finitos (FSM - *Finite State Machine*), evitando maior consumo de recursos. As operações de multiplicação foram feitas usando os multiplicadores embarcados na FPGA [35].

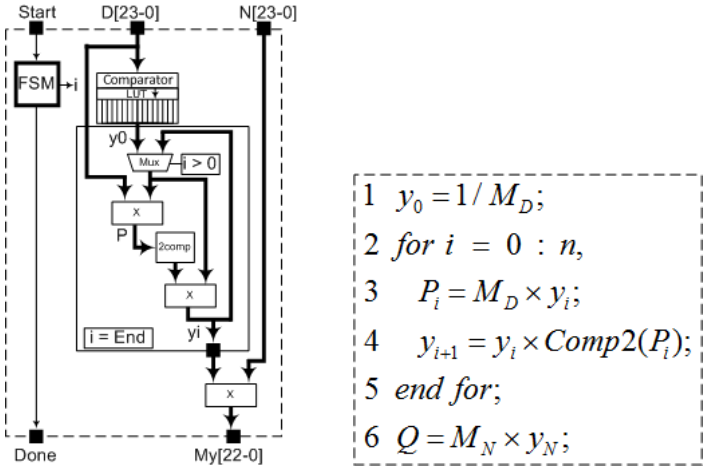
Algoritmo Newton Raphson para divisão

Da mesma forma que o algoritmo Goldschmidt, os operandos de entrada devem cumprir as mesmas condições ($M_N \geq 1$, $M_D \leq 2$), e o algoritmo inicia com uma aproximação inicial $y_0 \approx 1/D$. Depois as equações 3.2 e 3.3 devem ser executadas iterativamente.



(a) software (b) hardware

Figura 3.4: Implementação do algoritmo Goldschmidt para divisão



(a) hardware (b) software

Figura 3.5: Implementação do algoritmo Newton - Raphson para divisão

$$p = D \times y_i \tag{3.2}$$

$$y_{i+1} = yi \times (2 - p) \tag{3.3}$$

Depois da i^{th} iteração o produto da multiplicação de $N \times y_{i+1}$ produz uma aproximação a N/D. A principal diferença dos algoritmos Goldschmidt e Newton-Raphson radica em que este último, iterativamente refina a aproximação inicial, e só no final multiplica esta aproximação por o valor do numerador obtendo assim o resultado do cociente.

Na figura 3.5 é apresentada a implementação hardware e software deste algoritmo. Na arquitetura hardware as aproximações inicias a $1/M_D$ são armazenadas em uma *look-up table* e contínuos refinamentos de y_0 são realizados. Este algoritmo só calcula o cociente na iteração final, em quanto que, no algoritmo Goldschmidt em cada iteração é obtida uma nova aproximação ao cociente. Nesta arquitetura as operações de multiplicação são executadas usando os multiplicadores internos disponíveis no FPGA.

Comparando as arquiteturas dos algoritmos Goldschmidt e Newton-Raphson (figuras 3.4 e 3.5) pode-se observar que este último é mais simples de implementar em hardware. Principalmente pode-se observa que na arquitetura baseada em Newton-Raphson os multiplicadores são usados sequencialemente e precissase trabalhar com uma variável a menos em cada iteração, o que poderia levar a um menor consumo de recursos.

3.2.4 Raiz Quadrada

Os operadores de raiz quadrada e divisão em ponto flutuante são menos freqüentes em implementações desenvolvidas para FPGAs. Contudo, hoje encontramos diferentes algoritmos desenvolvidos e implementados nestes dispositivos. Da mesma maneira que na divisão, a implementação da raiz quadrada é dividida em operações independentes do expoente e mantissa. Neste documento são descritos os algoritmos Goldschmidt e Newton-Raphson, usados no desenvolvimento deste projeto.

O algoritmo geral usado na implementação do operador aritmético de raiz quadrada em ponto flutuante é baseado nos seguintes passos [35]:

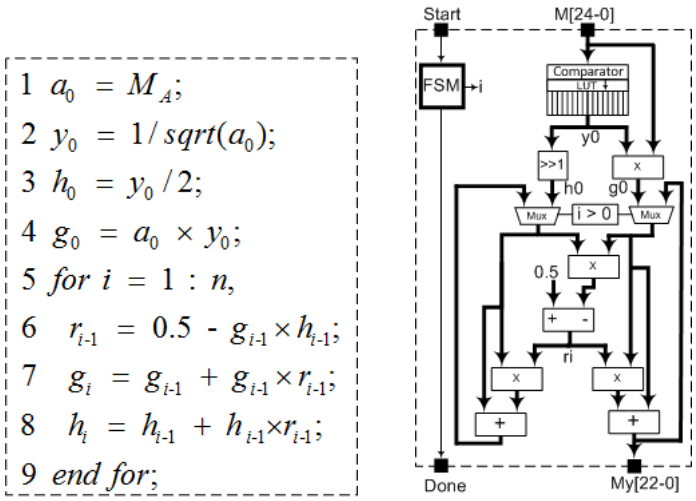
1. Seja A um número real representado no padrão IEEE-754 do qual deseja-se obter a raiz quadrada.
2. Detectar entradas negativas, zero e dados inválidos. Separar o sinal, o expoente e a mantissa adicionando o bit implícito nesta. Finalmente, se o expoente é par multiplicar a mantissa por 2.
3. Calcular a mantissa resultante usando um dos algoritmos descritos a continuação (Goldschmidt ou Newton-Raphson para raiz quadrada), e em paralelo avaliar o resultado do expoente usando a equação 3.4.
4. Finalmente, remover o bit implícito da mantissa e concatenar o sinal, expoente e mantissa resultantes.

$$Exp_r = \frac{Exp(X) + bias}{2} \tag{3.4}$$

3.2.4.1 Algoritmo Goldschmidt para raiz quadrada

Para uma mantissa de entrada M_A, este algoritmo calcula a $\sqrt{M_A}$, partindo de uma aproximação inicial igual a $1/\sqrt{M_A}$, e através de um processo iterativo a mantissa resultante é aproximada ao valor $\sqrt{M_A}$. Este trabalho tomou como referência o algoritmo proposto em [53], no qual apresentam-se os seguintes passos:

- Fazer $a_0 = M_A$, em uma variável y_0 armazenar uma boa aproximação a $1/\sqrt{a_0}$, dividir esta aproximação por 2 e armazenar o resultado em h_0 ($h_0 = y_0/2$). Finalmente, fazer um processo iterativo calculando as equações 3.5a -3.5c



(a) software (b) hardware

Figura 3.6: Implementação do algoritmo Goldschmidt para raiz quadrada

$$r_i = 0.5 - g_{i-1}h_{i-1} \tag{3.5a}$$

$$g_i = g_{i-1} + g_{i-1}r_{i-1} \tag{3.5b}$$

$$h_i = h_{i-1} + h_{i-1}r_{i-1} \tag{3.5c}$$

Em cada iteração é obtido um valor mais próximo a $\sqrt{M_A}$ o qual é armazenado na variável g. Na variável h é armazenado um valor próximo a $1/2\sqrt{M_A}$. O algoritmo apresentado por [53], e aqui descrito, permite o cálculo em paralelo das equações 3.5b e 3.5c, além de evitar sucessivas divisões por 2.

Na figura 3.6 é mostrada uma implementação hardware e software do algoritmo Goldschmidt usado no cálculo da raiz quadrada. De acordo com o algoritmo geral os valores de entrada abrangem a faixa $[1, 4)$ ($M_A \in [1, 4)$). Na arquitetura hardware, as sementes inicias ($1/\sqrt{M_A}$) são armazenadas em *look-up tables*, e tudo o processo é controlado por uma FSM compartilhando multiplicadores e somadores e reduzindo assim o consumo de recursos [35].

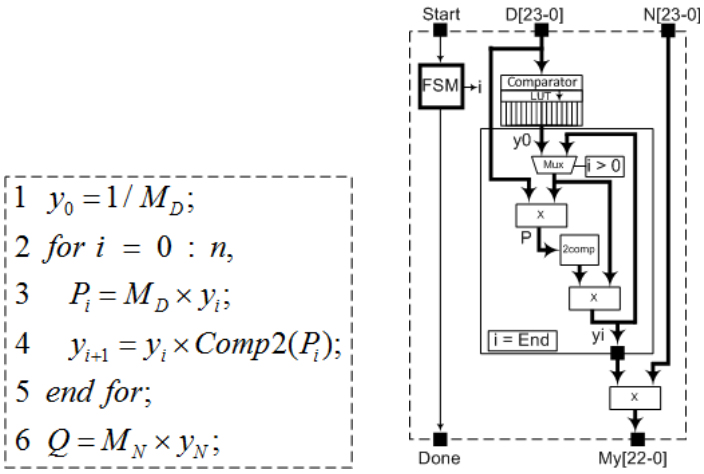
3.2.4.2 Algoritmo Newton-Raphson para raiz quadrada

Este algoritmo permite calcular a raiz quadrada iniciando a partir de uma aproximação inicial igual a $y_0 \approx 1/\sqrt{b}$ ($b = M_A$) e refinando esta iterativamente a partir do cálculo da equação 3.6.

$$y_{i+1} = 0.5 \times y_i \times (3 - b \times y_i^2) \tag{3.6}$$

Depois da i^{th} iteração, na variável y_{i+1} é acumulado um valor mais próximo a $1/\sqrt{M_A}$. Finalmente, uma aproximação a $\sqrt{M_A}$ é obtida multiplicando $y_{i+1} \times M_A$.

Na figura 3.7 é mostrada uma implementação hardware e software do algoritmo Newton-Raphson usado no cálculo da raiz quadrada. Da mesma forma ao algoritmo Goldschmidt para



(a) software (b) hardware

Figura 3.7: Implementação do algoritmo Newton-Raphson para raiz quadrada

raiz quadrada, nesta abordagem o valor de entrada abrange a faixa $[1, 4)$ ($M_A \in [1, 4)$). Na arquitetura do hardware, as sementes inicias ($1/\sqrt{M}$) são armazenadas em *look-up tables*, e tudo o processo é controlado por uma FSM compartindo multiplicadores e somadores e reduzindo assim o consumo de recursos.

3.3 Algoritmos para implementação de operadores para funções transcendentais

3.3.1 CORDIC

CORDIC (Coordinate Rotation Digital Computer) é um método para calcular funções elementares e transcendentais usando componentes de hardware mínimos, como deslocamentos lógicos e comparadores [54].

O algoritmo CORDIC executa uma rotação no plano. Graficamente, isto significa transformar um vetor (x_i, y_i) em um novo vetor (x_j, y_j), como mostrado na figura 3.8.

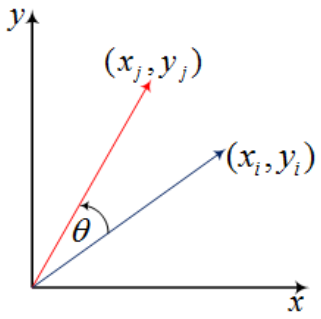


Figura 3.8: Rotação do vetor (x_i, y_i) por um ángulo θ

Usando a notação matricial, uma rotação no plano de um vetor (x_i, y_i) é definida pela equação 3.7.

$$\begin{bmatrix} X_j \\ Y_j \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \times \begin{bmatrix} X_i \\ Y_i \end{bmatrix} \tag{3.7}$$

Este algoritmo propõe executar a rotação θ em múltiplos passos usando um processo iterativo descrito em [28], no qual a matriz de rotação é redefinida segundo a equação 3.8.

$$\begin{bmatrix} X_{n+1} \\ Y_{n+1} \end{bmatrix} = \begin{bmatrix} \cos\theta_n & -\sin\theta_n \\ \sin\theta_n & \cos\theta_n \end{bmatrix} \times \begin{bmatrix} X_n \\ Y_n \end{bmatrix} \tag{3.8}$$

Na equação 3.8 é eliminado da matriz o termo $\cos(\theta_n)$, onde a matriz de rotação fica como mostrado na equação 3.9.

$$\begin{bmatrix} X_{n+1} \\ Y_{n+1} \end{bmatrix} = \cos\theta_n \times \begin{bmatrix} 1 & -\tan\theta_n \\ \tan\theta_n & 1 \end{bmatrix} \times \begin{bmatrix} X_n \\ Y_n \end{bmatrix} \tag{3.9}$$

Posteriormente, operações de multiplicação podem ser eliminadas, selecionando incrementos do ângulo (θ), de tal forma que a tangente do incremento seja uma potência de 2. Multiplicações e divisões por 2, facilmente podem ser implementadas usando operadores lógicos de deslocamento (a esquerda para multiplicar e a direita para dividir).

O ângulo de cada incremento é definido pela equação 3.10.

$$\theta_n = \arctan\left(\frac{1}{2^n}\right) \tag{3.10}$$

onde,

$$\sum_{n=0}^{\infty} S_n\theta_n = \theta \tag{3.11a}$$

$$S_n = \{-1; +1\} \tag{3.11b}$$

substituindo a equação 3.11a na equação 3.10 obtem-se a equação 3.12.

$$\tan\theta_n = S_n 2^{-n} \tag{3.12}$$

Finalmente, substituindo 3.12, em 3.9, obtêm-se equação 3.13.

$$\begin{bmatrix} X_{n+1} \\ Y_{n+1} \end{bmatrix} = \cos\theta_n \times \begin{bmatrix} 1 & -S_n 2^{-n} \\ -S_n 2^{-n} & 1 \end{bmatrix} \times \begin{bmatrix} X_n \\ Y_n \end{bmatrix} \tag{3.13}$$

Desta forma, o algoritmo foi reduzido em operações simples de deslocamento e soma. Adicionalmente, o coeficiente $\cos\theta_n$ pode ser eliminado seguindo o procedimento matemático descrito a continuação:

1. Reescreva o coeficiente como mostrado na equação 3.14.

$$\cos\theta_n = \cos\left(\arctan\left(\frac{1}{2^n}\right)\right) \tag{3.14}$$

2. Fazendo o cálculo da equação 3.14 para todos os valores de n e multiplicando os resultados se obtém um valor como descrito na equação 3.15.

$$K = \frac{1}{P} = \prod_{n=0}^{\infty} \cos\left(\arctan\left(\frac{1}{2^n}\right)\right) \approx 0.607253 \tag{3.15}$$

onde K é um parâmetro constante para todos os valores que tome o ângulo de rotação. Outro parâmetro comumente usado é P. Este parâmetro é igual a $P = 1/K \approx 1,64676$.

O parâmetro K somente é tido em conta no final do procedimento. Portanto, poder-se-ia reescrever a equação 3.13 segundo a equação 3.16.

$$\begin{bmatrix} X_{n+1} \\ Y_{n+1} \end{bmatrix} = \begin{bmatrix} 1 & -S_n 2^{-n} \\ -S_n 2^{-n} & 1 \end{bmatrix} \times \begin{bmatrix} X_n \\ Y_n \end{bmatrix} \tag{3.16}$$

Neste ponto, uma nova variável denominada Z é introduzida. Onde Z representa a parte do ângulo que ainda falta por rotar (ver equação 3.17).

$$Z_{n+1} = \theta - \sum_{i=0}^{n} \theta_i \tag{3.17}$$

Para cada passo da rotação a variável S_n é calculada de acordo com o sinal de Z_n (ver equação 3.18).

$$S_n = \begin{cases} -1 & if\ Z_n < 0 \\ +1 & if\ Z_n \geq 0 \end{cases} \tag{3.18}$$

Combinando as equações 3.11a e 3.18, obtêm-se um sistema que reduz a parte não-rotado do ângulo θ a zero.

$$Z_{n+1} = Z_n - \arctan(1/2^n) \tag{3.19}$$

O valor do $\arctan(1/2^n)$ é previamente calculado e armazenado em uma *look-up table* (LUT).

Este algoritmo leva Z a zero, e a partir de este procedimento pode-se calcular a equação 3.20.

$$\left[P\left(X_i\cos\left(Z_i\right) - Y_i\sin\left(Z_i\right)\right),\ P\left(Y_i\cos\left(Z_i\right) + X_i\sin\left(Z_i\right)\right),\ 0\right] \tag{3.20}$$

Nesta configuração as variáveis são inicializadas como mostrado nas equações 3.21a - 3.21c

$$X_i = \frac{1}{P} = K \approx 0.60725 \tag{3.21a}$$

$$Y_i = 0 \tag{3.21b}$$

$$Z_i = \theta \tag{3.21c}$$

obtendo como resultado o vetor rotado mostrado na equação 3.22.

$$[X_j, Y_j, Z_j] = [\cos\theta,\ \sin\theta,\ 0] \tag{3.22}$$

Outra configuração deste algoritmo é obtida tentando levar Y a zero. Neste caso, o algoritmo CORDIC pode ser inicializado de duas maneiras:

1. Inicializando os parâmetros em: $X_i = X$, $Y_i = Y$, $Z_i = 0$ pode-se calcular a equação 3.23.
2. Inicializando os parâmetros em: $X_i = 1$, $Y_i = a$, $Z_i = 0$, pode-se obter a equação 3.24.

$$[X_j, Y_j, Z_j] = \left[P\sqrt{X_i^2 + Y_i^2},\ 0,\ \arctan\left(\frac{Y_i}{X_i}\right)\right] \tag{3.23}$$

$$[X_j, Y_j, Z_j] = \left[P\sqrt{1 + a^2},\ 0,\ \arctan\left(a\right)\right] \tag{3.24}$$

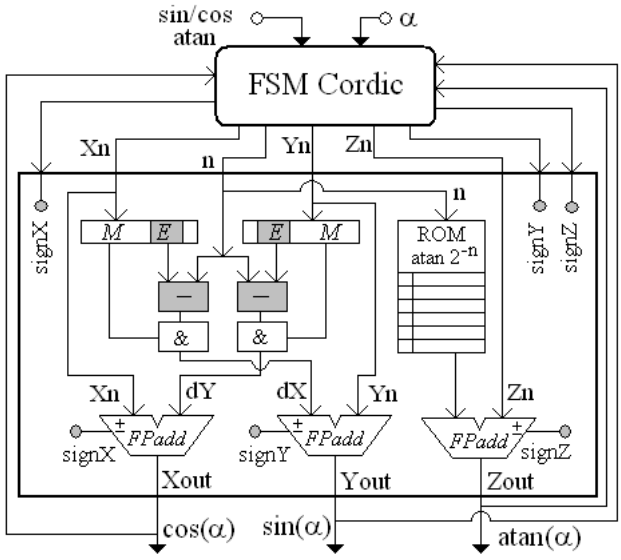
Em resumo, a partir de pequenas variações nos parâmetros do algoritmo CORDIC é possível calcular as seguintes funções:

- Rotação de vetores (polar a rectangular).
- Seno e cosseno.
- Seno e cosseno hiperbólico.
- Arcotangente.
- Arcotangente hiperbólica.
- Raiz quadrada.

Na figura 3.9, é apresentada uma implementação software do algoritmo CORDIC para o cálculo das funções seno e cosseno. Tanto na implementação hardware como software, os valores $\arctan(1/2^i)$ são pré-calculados e armazenados em uma *look-up table*. A arquitetura hardware apresentada permite calcular seno, cosseno e arco-tangente, com uma mudança simples dos parâmetros do algoritmo controlada pela FSM. Esta arquitetura é baseada em (a) três unidades de ponto flutuante de soma/subtração ($FPadd$), (b) uma função de busca em ROM para recuperar os valores pré calcula-os de $atan(1/2^i)$ e (c) dois somadores em pronto fixo de tamanho ($E_w - bits$) para calcular $2^{-i}Y_i$ e $2^{-i}X_i$ [55].

```
for n = 0 : n,
  if (Z(n) >= 0) then
    X(n+1) = X(n) - (Y(n) / 2^n);
    Y(n+1) = Y(n) + (X(n) / 2^n);
    Z(n+1) = Z(n) - atan(1 / 2^n);
  else
    X(n+1) = X(n) + (Y(n) / 2^n);
    Y(n+1) = Y(n) - (X(n) / 2^n);
    Z(n+1) = Z(n) + atan(1 / 2^n);
  end if;
end for;
```

(a) software



(b) hardware

Figura 3.9: Implementação do algoritmo CORDIC.

3.3.2 Expansão em Séries de Taylor

A série de Taylor de uma função infinitamente diferençável real ou complexa $f(x)$ centrada em a, pode ser representada como mostrado na equação 3.25.

$$T(x) = \sum_{n=0}^{\infty} \frac{f^n(a)}{n!}(x-a)^n \tag{3.25}$$

Esta série representa uma função como uma soma de termos polinomiais, calculados a partir da sua derivada no ponto a. Portanto, a partir da equação 3.25 é possível representar em soma de polinômios diferentes funções entre as quais tem-se a função exponencial e um logaritmo natural, funções trigonométricas, funções hiperbólicas entre outras.

No desenvolvimento deste trabalho é tido como objetivo a implementação da cinemática direta em hardware de um robô manipulador de configuração esférica e para isso implementou-se as funções seno, cosseno e arco-tangente. A continuação a expansão em series de Taylor destas funções as quais são apresentadas segundo as equações 3.26a - 3.26c.

$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!}x^{(2n+1)} \qquad \forall x \tag{3.26a}$$

$$\cos(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n!}x^{2n} \qquad \forall x \tag{3.26b}$$

$$\arctan(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}x^{(2n+1)} \qquad \rightarrow |x| < 1 \tag{3.26c}$$

Uma arquitetura em hardware que implementa as séries de Taylor para o cálculo de seno, cosseno e arco-tangente é mostrada na figura 3.10. Esta arquitetura foi projetada usando um mul-

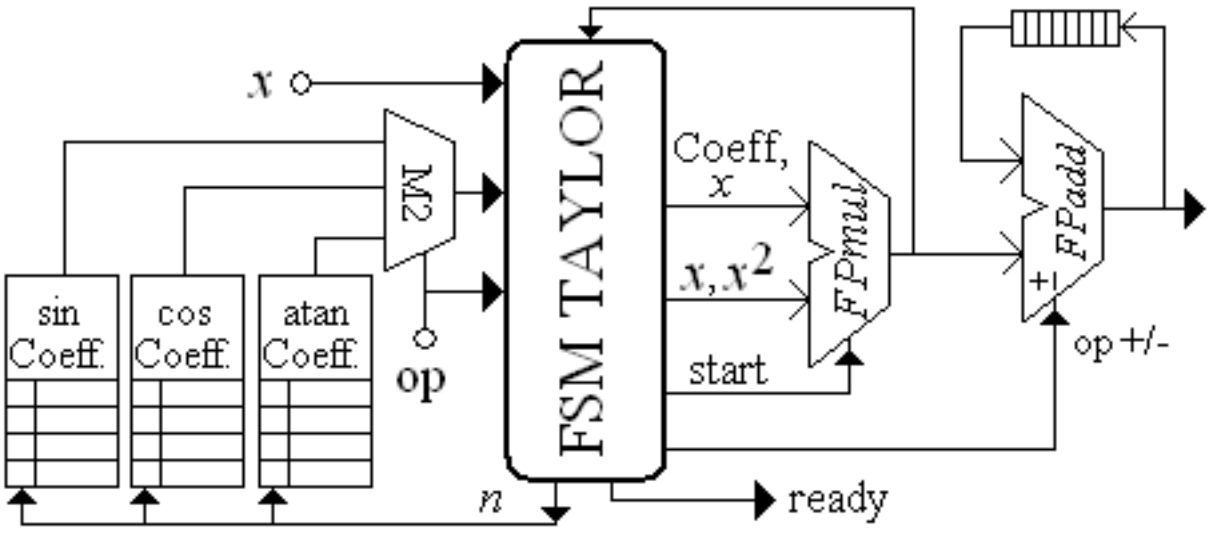


Figura 3.10: Implementação das funções seno, cosseno e arco-tangente baseado em series de Taylor.

tiplicador e um somador em ponto flutuante, $FPmult$ e $FPadd$ respectivamente. Adicionalmente, foram usadas três memórias ROM para o armazenamento de valores pré-calculados dos parâmetros $1/n!$ e $1/n$ (vide figura 3.10).

Na arquitetura o parâmetro *op* permite escolher a função a ser calculada (seno, cosseno ou arco-tangente). A FSM sincroniza as unidades $FPmul$ e $FPadd$ e alterna o sinal $op + /-$. Na primeira iteração, o valor x^2 é calculado e armazenado em um registrador e realimentado para a FSM. Subseqüentemente, são carregados os coeficiente pré-calculados no intuito de calcular os termos $x^{2n+1}/(2n+1)!$ paro o seno, $x^{2n+1}/(2n+1)$ para arco-tangente, e $\frac{x^{2n}}{2n!}$ para o cosseno. Finalmente, depois de n iterações, o sinal *ready* indica uma saída válida [55].

3.4 Conclusões do Capítulo

Neste capítulo foram apresentados os algoritmos usados na implementação de operadores aritméticos de ponto flutuante e as arquiteturas propostas para sua implementação em VHDL. Os operadores de soma/subtração e multiplicação são algoritmos mais "simplesťť, os quais não requerem de processos iterativos para produzir o resultado e constituem operadores elementares que poderiam ser usados na implementação de outros operadores mais complexos.

Dois algoritmos foram introduzidos no intuito de implementar em VHDL os operadores aritméticos de divisão e raiz quadrada a saber, Goldschmidt e Newton-Raphson. Estes algoritmos baseiam-se em processos iterativos de refinamento de aproximações iniciais e operam somente no cálculo da mantissa do resultado. A principal diferença destes algoritmos radica na forma de se operar os resultados parciais. Em quanto em Goldschmidt em cada iteração é obtido uma nova aproximação ao resultado, em Newton-Raphson somente no final das iterações é obtida ama aproximação ao resultado esperado.

Na implementação de operadores aritméticos de funções elementares foi descrito o algoritmo CORDIC, amplamente usado em implementação de operadores em ponto fixo. Diferentes funções

podem ser calculadas partindo deste algoritmo. No desenvolvimento deste trabalho somente é tomada em conta a implementação de funções seno, cosseno e arco-tangente. Na arquitetura apresentada se propõe uma implementação do algoritmo CORDIC em ponto flutuante para o qual três unidades de soma/subtração são usadas em paralelo.

De igual forma, se introduziu neste trabalho uma arquitetura baseada nas séries de Taylor. Esta arquitetura internamente contém uma unidade de soma/subtração e uma unidade de multiplicação em ponto flutuante.

Com as diferentes arquiteturas aqui apresentadas pretende-se fazer um estúdio de compromisso entre parâmetros tais como consumo de área, latência e precisão nos cálculos, e desta forma se ter um critério para a eleição do algoritmo que possua o melhor comportamento de acordo com a aplicação. Pode ser visto que as arquiteturas propostas exploram o paralelismo implícito dos algoritmos, prevendo como isto ganhos de desempenho, como serão discutidos no capítulo 5.

Finalmente, implementou-se uma biblioteca completa de operadores aritméticos de ponto flutuante em FPGA que permita ser usada no cálculo da cinemática direta de manipuladores, e em outras aplicações que precisem de altas velocidade de processamento.

Capítulo 4

Obtenção da Cinemática Direta do Manipulador e sua Implementação em Hardware

Atualmente, há uma alta demanda de aplicações robóticas que operam em altas velocidades, nas quais os cálculos de algoritmos de controle e posicionamento devem ser realizados na ordem de milissegundos (ms) ou micro-segundos (μs). Em geral, estes algoritmos requerem da execução em tempo real de operações repetitivas e complexas que abrangem o cálculo de funções transcendentais e operações aritméticas idealmente calculadas em ponto flutuante. Estes cálculos repetitivos e complexos, em muitas ocasiões ultrapassam as capacidades dos processadores de propósitos gerais (GPPs).

O comportamento seqüencial dos GPPs apresenta limitação em aplicações que requerem altas velocidades de processamento, isto está dentro do contexto do modelo de von Neumman [2]. Entretanto, as FPGAs ultimamente estão sendo usadas em aplicações tais como o processamento digital de sinais, processamento de imagens, encriptação e desencriptação, decodificação de protocolos de comunicação, robótica entre outros, devido a suas potencialidades de processamento em paralelo.

A cinemática direta é uma formulação matemática que freqüentemente precisa de um elevado número de operações aritméticas e trigonométricas. Para propósitos de robótica estas operações devem ser calculadas com alta precisão. Desta forma, a adoção de aritmética de ponto flutuante deve ser uma característica obrigatória neste tipo de aplicações. Neste contexto, uma implementação em FPGA da cinemática direta totalmente em hardware, usando aritmética de ponto flutuante, se converte em um desafio muito interessante no intuito de superar restrições de execução da tarefa em tempo real.

No início deste capítulo são descritos trabalhos correlatos que implementam a cinemática de manipuladores em FPGA. Em seguida, é apresentado o modelo matemático para obtenção da cinemática direta do robô manipulador de configuração esférica com cinco graus de liberdade. Finalmente, é apresentada uma arquitetura para calcular a cinemática direta totalmente em hardware usando um FPGA VIRTEX II PRO XC2VP30.

4.1 Trabalhos Correlatos à Implementação da Cinemática Direta em Hardware

Na literatura são encontrados poucos trabalhos abrangendo implementação de cinemática direta e inversa em FPGA. Neste sentido, encontram-se com maior freqüência trabalhos apresentando arquiteturas hardware na implementação de laços de controle para servo-motores usados em manipuladores robóticos, e implementações software para o cálculo da cinemática direta e inversa [30, 32, 31, 33].

Nas referencias [30] e [31] são apresentadas arquiteturas de hardware para controlar um robô manipulador SCARA. Entretanto, os complexos cálculos envolvendo operações aritméticas são executados em um GPP e deixando para a FPGA os algoritmos básicos de controle dos servos usados no robô.

Em [32] e [33] é apresentada uma arquitetura de co-projeto de hardware/software (*hardware/software co-design*) no intuito de controlar um robô manipulador. A arquitetura apresentada usa um processador NIOS II embarcado na FPGA sobre o qual é implementada a cinemática inversa do robô.

Em [56] e [57] apresenta-se uma arquitetura baseada em DSP e FPGA. Em [56] esta arquitetura é usada para controlar um robô de juntas flexíveis (*flexible joint robot*), um DSP de ponto flutuante é usado na implementação. Em [57] apresenta-se uma arquitetura baseada em lógica nebulosa para o controle de um robô manipulador de dois graus de liberdade.

Kung et al. [4] apresentam uma implementação hardware para calcular a cinemática inversa e controlar os servo-motores usados no robô, deixando para a FPGA o cálculo de complexas operações como funções transcendentais, assim como explorar as capacidades de paralelismo na implementação da cinemática inversa.

Em resumo, trabalhos prévios descrevem o co-projeto de hardware/software na implementação da cinemática direta e inversa. Porém, a nenhum destes trabalhos executam a aritmética de ponto flutuante diretamente em hardware. Como um caso particular, o trabalho [56] considera o cálculo de operações em ponto flutuante usando um Processador Digital de Sinais (DSP). Outros usam GPPs para executar este tipo de operações matemáticas.

Neste trabalho mostra-se uma implementação para calcular a cinemática direta de um robô manipulador de configuração esférica. A implementação em hardware, considera o uso da biblioteca aritmética de ponto flutuante parametrizável pelo tamanho da palavra descrito no capítulo 3.

4.2 Modelo cinemático Direto de um Robô Manipulador de Configuração Esférica

Como caso de estúdio neste trabalho considera-se um robô manipulador de configuração esférica com 5 graus de liberdade. Este robô tem uma junta rotacional situada na base dando origem ao movimento de *Roll*, seguida por outra junta rotacional no eixo y (*Pitch*). Finalmente, tem-se uma

junta prismática seguida de duas juntas rotacionais de origens coincidentes situada na mão do manipulador. As três primeiras juntas são responsáveis pela localização, e as duas últimas pela orientação da mão do robô. Na figura 4.1 apresenta-se a topologia do robô.

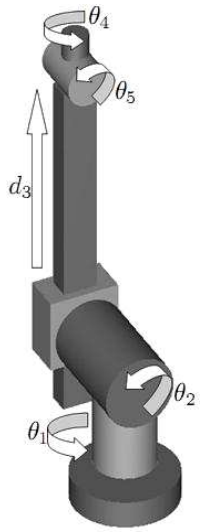


Figura 4.1: Robô manipulador de 5 gruas de liberdade (objeto de estudo)

Seguindo o procedimento descrito no capítulo 2, para obter o modelo cinemático direto, o primeiro passo é levar o robô até a posição zero. Neste caso, assume-se a posição zero como mostrado na figura 4.1.

Depois é atribuído um marco de referência para cada enlace, como mostrado na figura 4.2. A partir do marco de coordenadas atribuido a cada enlace obtêm-se as relações entre as variáveis das juntas e dos elos e, finalmente, se obtem a tabela 4.1 com os parâmetros de Denavit e Hartenberg [13].

Tabela 4.1: Parâmetros de Denavit-Hartenberg paro o robô caso de estudo

Junta Variável	Ângulo θ_n	Deslocamento d_n	longitude l_n	Torção α_n
θ_1	θ_1	$d_1(106mm)$	0	90
θ_2	$\theta_2 + 90$	$d_2(130mm)$	$l_2(0mm)$	-90
d_3	0	d_3	0	0
θ_4	θ_4	0	0	90
θ_5	θ_5	$d_5(0mm)$	0	0

A partir destes parâmetros são obtidas as matrizes que relacionam os marcos coordenados consecutivos. Para se obter estas matrizes e a matriz de transformação geral, foi construído um programa em MATLAB que permite obter automaticamente estas matrizes, partindo dos parâmetros de Denavit e Hartenberg (D-H).

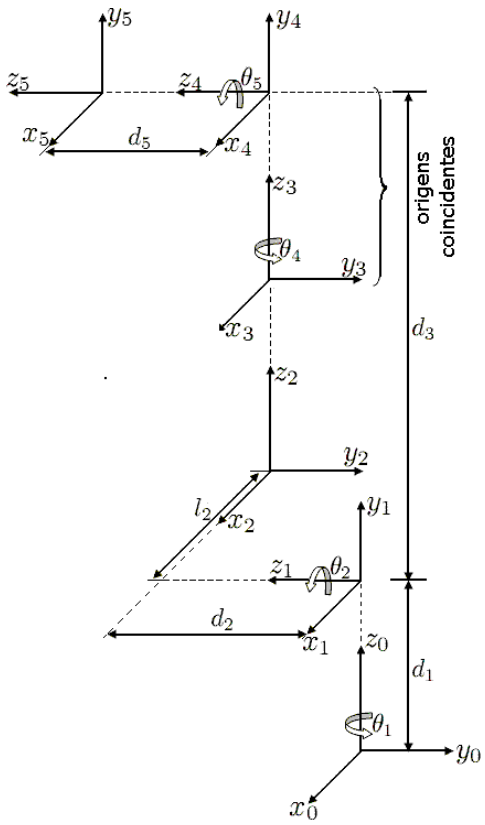


Figura 4.2: Atribuição de um marco de referencia para cada enlace.

4.3 Verificação da metodologia de cálculo da cinemática direta (o caso do robô PUMA)

A continuação se usará como exemplo de teste do programa desenvolvido o caso do clássico robô manipulador PUMA [13], para o qual são obtidos os parâmetros de Denavit e Hartenberg mostrados na tabela 4.2.

Tabela 4.2: Parâmetros D-H, para o robô manipulador PUMA

Junta Variável	Ângulo θ_n	Deslocamento d_n	Translação l_n	Torção α_n
θ_1	θ_1	d_1	0	90
θ_2	θ_2	d_2	l_2	0
θ_3	θ_3	0	0	-90
θ_4	θ_4	d_4	0	90
θ_5	θ_5	0	0	-90
θ_6	θ_6	d_6	0	0

A partir dos parâmetros da tabela 4.2, foi construído um *script* em MATLAB como mostrado na figura 4.3.

No *script* são declaradas variáveis simbólicas para os parâmetros d_1, d_2, d_4, d_6 e l_2. No caso de se conhecer a medida exata destes parâmetros, estes podem ser diretamente substituídos por

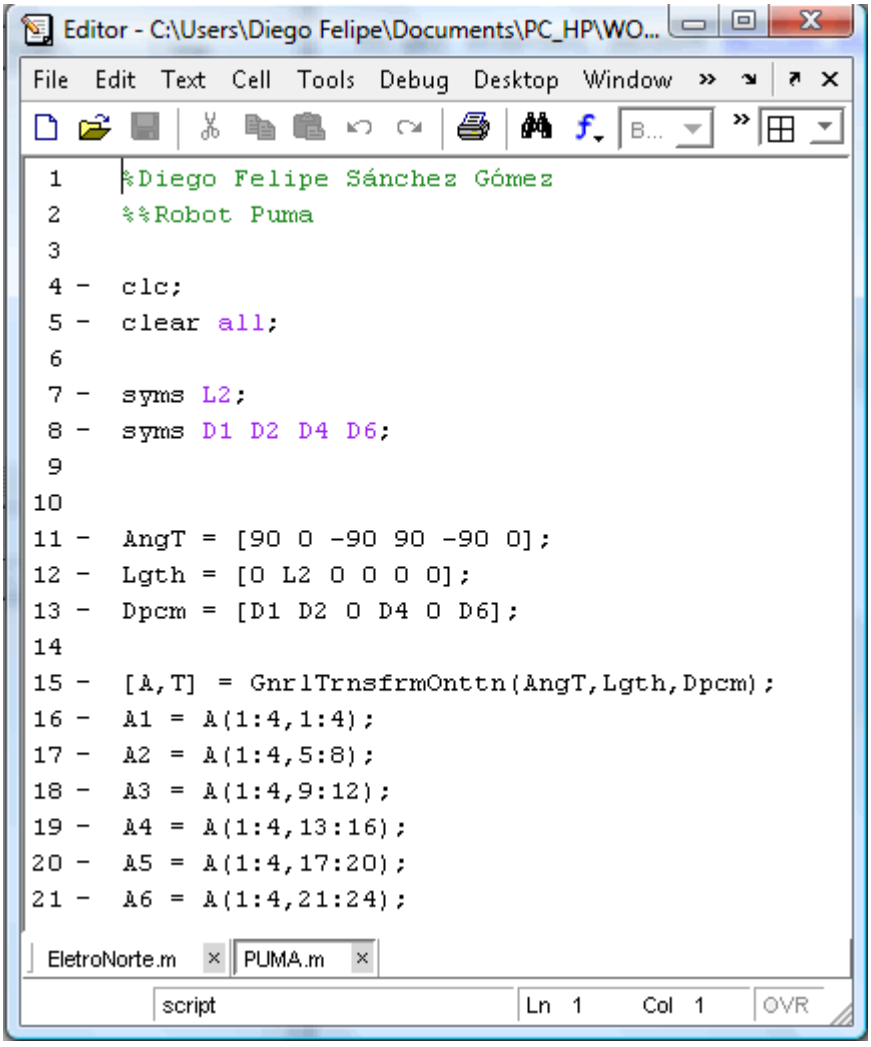


Figura 4.3: *Script* de Parâmetros D-H para robô manipulador PUMA

seus valores numéricos. Nas variáveis $AngT$, $Lgth$ e $Dpcm$ são armazenados os valores do ângulo de torção α_n, translação longitudinal l_n e deslocamento d_n, respetivamente. Estes parâmetros são passados à função previamente construída $GnrlTrnsfrmOnttn$ que retorna as matrizes de relação de marcos coordenados consecutivos e a matriz de transformação geral (vide figura 4.4).

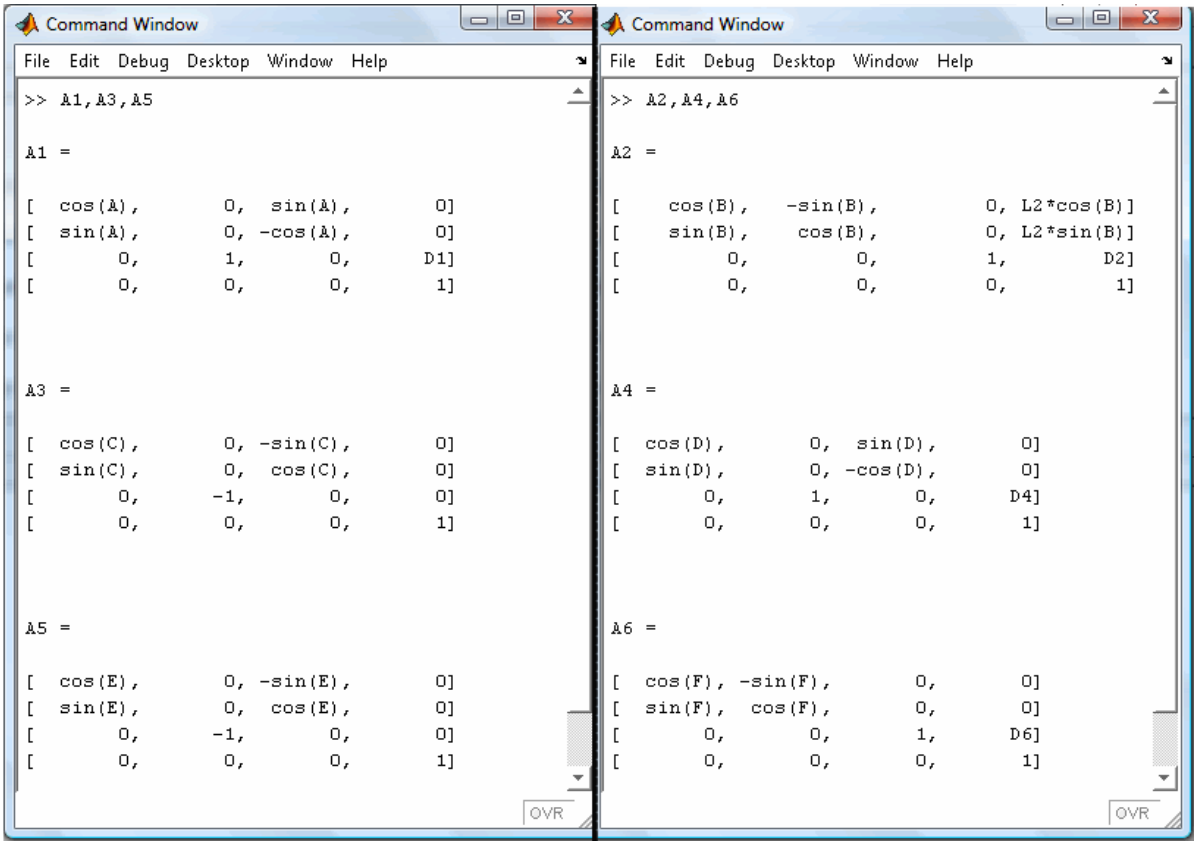


Figura 4.4: Matrizes obtidas para relação de marcos coordenados consecutivos do robô PUMA.

A matriz de transformação geral é obtida multiplicando estas seis matrizes. O programa descrito em MATLAB faz este procedimento automaticamente, obtendo-se as equações 4.1 a 4.12.

$$X_x = (((C_AC_BC_C - C_AS_BS_C)C_D - S_AS_D)C_E + (-C_AC_BS_C - C_AS_BC_C)S_E)C_F + (-(C_AC_BC_C - C_AS_BS_C)S_D - S_AC_D)S_F \quad (4.1)$$

$$X_y = (((S_AC_BC_C - S_AS_BS_C)C_D + C_AS_D)C_E + (-S_AC_BS_C - S_AS_BC_C)S_E)C_F + (-(S_AC_BC_C - S_AS_BS_C)S_D + C_AC_D)S_F \quad (4.2)$$

$$X_z = ((S_BC_C + C_BS_C)C_DC_E + (-S_BS_C + C_BC_C)S_E)C_F - (S_BC_C + C_BS_C)S_DS_F \quad (4.3)$$

$$Y_x = -(((C_AC_BC_C - C_AS_BS_C)C_D - S_AS_D)C_E + (-C_AC_BS_C - C_AS_BC_C)S_E)S_F + (-(C_AC_BC_C - C_AS_BS_C)S_D + C_AC_D)C_F \quad (4.4)$$

$$Y_y = -(((S_AC_BC_C - S_AS_BS_C)C_D + C_AS_D)C_E + (-S_AC_BS_C - S_AS_BC_C)S_E)S_F + (-(S_AC_BC_C - S_AS_BS_C)S_D + C_AC_D)C_F \quad (4.5)$$

$$Y_z = ((S_BC_C + C_BS_C)C_DC_E + (-S_BS_C + C_BC_C)S_E)S_F - (S_BC_C + C_BS_C)S_DC_F \quad (4.6)$$

$$Z_x = -((C_AC_BC_C - C_AS_BS_C)C_D - S_AS_D)S_E + (-C_AC_BS_C - C_AS_BC_C)C_E \quad (4.7)$$

$$Z_y = -((S_AC_BC_C - S_AS_BS_C)C_D - C_AS_D)S_E + (-S_AC_BS_C - S_AS_BC_C)C_E \quad (4.8)$$

$$Z_z = -(S_BC_C + C_BS_C)C_DS_E + (-S_BS_C + C_BC_C)C_E \quad (4.9)$$

$$P_x = (-((C_AC_BC_C - C_AS_BS_C)C_D - S_AS_D)S_E + (-C_AC_BS_C - C_AS_BC_C)C_E)d_6 + (-C_AC_BS_C - C_AS_BC_C)d_4 + C_AC_Bl_2 + S_Ad_2 \quad (4.10)$$

$$P_y = (-((S_AC_BC_C - S_AS_BS_C)C_D + C_AS_D)S_E + (-S_AC_BS_C - S_AS_BC_C)C_E)d_6 + (-S_AC_BS_C - S_AS_BC_C)d_4 + S_AC_Bl_2 + C_Ad_2 \quad (4.11)$$

$$P_z = (-(S_BC_C + C_BS_C)C_DS_E + (-S_BS_C + C_BC_C)C_E)d_6 + (-S_BS_C - C_BC_C)D_4 + S_Bl_2 + d_1 \quad (4.12)$$

Nas anteriores equações θ_i foi substituído por os valores consecutivos A, B, C, D, E e F, e foi usada a notação $C_x = cos(x)$, no intuito de simplificar as equações. Esta ferramenta implementada em MATLAB facilita o processo de obtenção da matriz de transformação homogênea. Além disso, permite o trabalho com variáveis simbólicas para futuros ajustes destes parâmetros.

4.4 Uso da metodologia proposta para o robô objeto do estudo

Para o robô de configuração esférica (caso de estudo), foi definido o *script* mostrado na figura 4.5. Este se diferencia do *script* apresentado para o robô PUMA pela inclusão da variável $AngV$. Esta variável é usada para definir um valor de *offset* entre a posição zero ideal e a posição zero real. Na figura 4.1 é mostrada a posição zero seguindo o procedimento descrito no capítulo 2. Já na figura 4.6, é mostrada a posição zero na implementação real do robô com uma rotação de 90 graus na segunda junta. Além do anterior, a figura 4.6 foi construída com os valores dos parâmetros reais do robô a ser construído.

Para calcular a matriz de transformação geral, e usada a função $GnrlTrnsfrmInPrsmtc$. Esta função tem como parâmetros de entrada os vetores com os valores de ângulos de offset, ângulos de torção, translação longitudinal, deslocamento e, finalmente, a posição da junta prismática. A

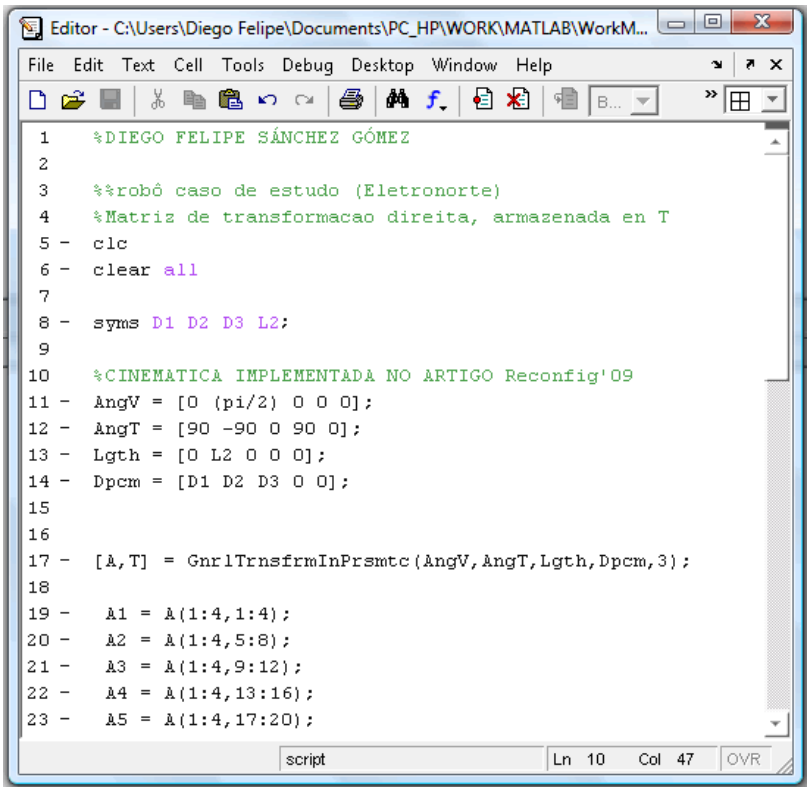


Figura 4.5: *Script* de parâmetros D-H para robô manipulador caso de estudo

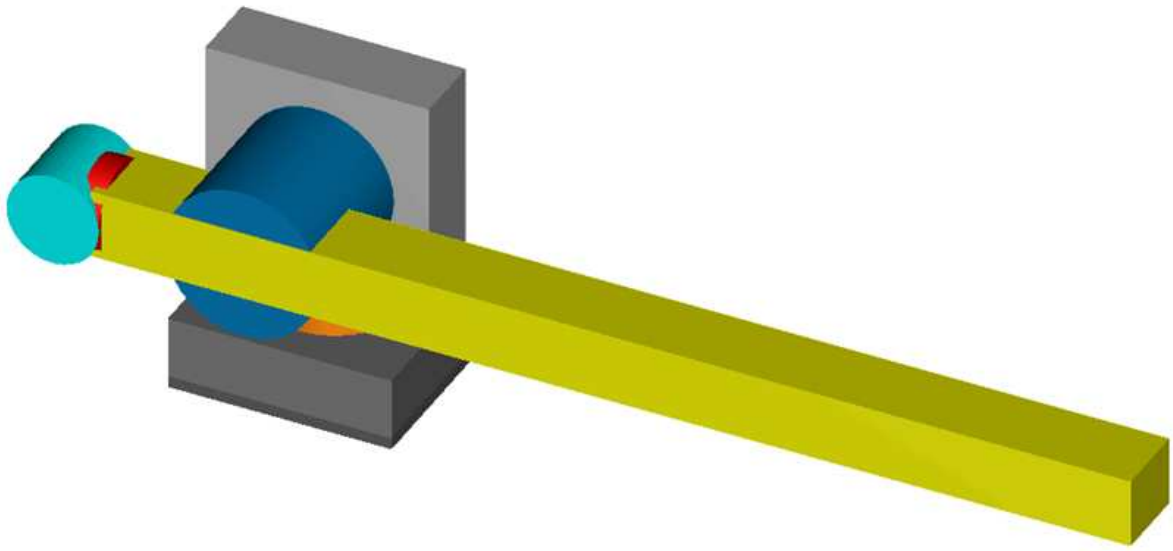


Figura 4.6: Posição zero real do robô.

função retorna as matrizes de relação de marcos de coordenadas consecutivas assim como a matriz de transformação geral (vide figura 4.7).

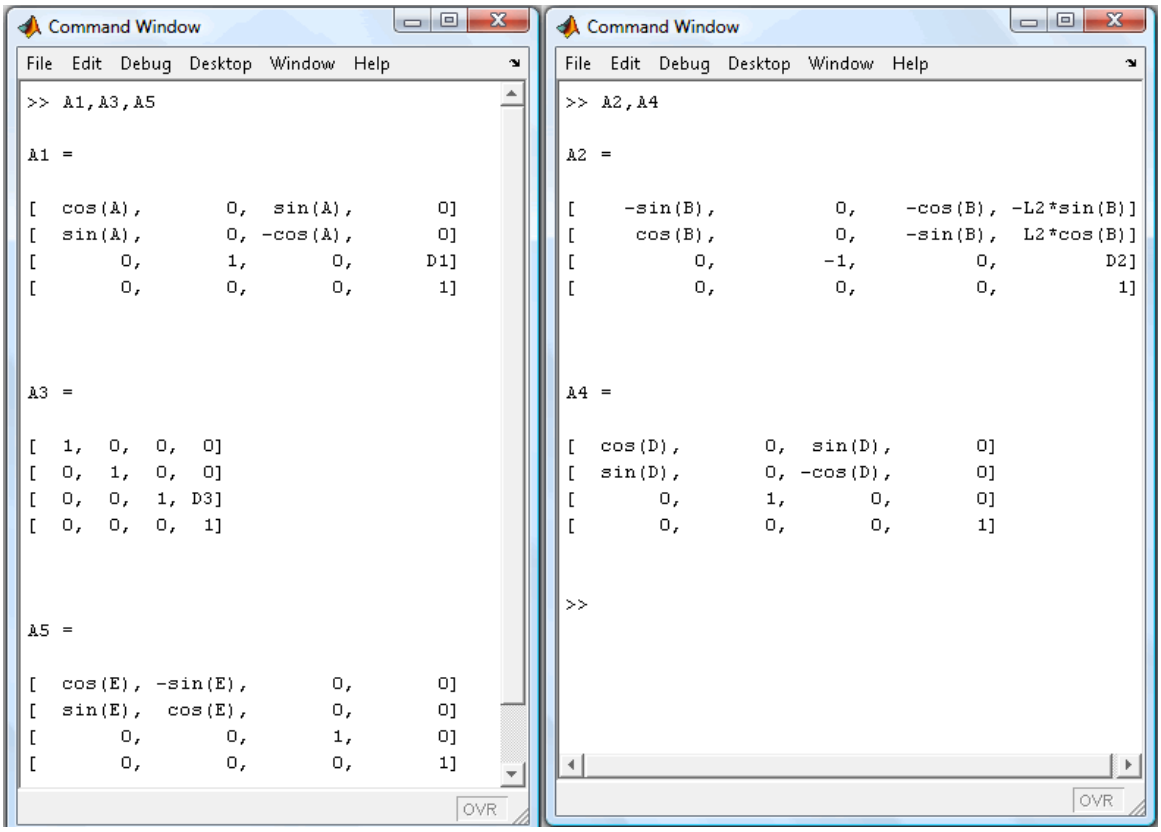


Figura 4.7: Matrizes obtidas para relação de marcos de coordenadas consecutivos do robô esférico

A matriz de transformação geral é obtida multiplicando as cinco matrizes. O programa descrito em MATLAB faz este procedimento automaticamente, obtendo as equações 4.13 a 4.24 .

$$x_x = (-C_1S_2C_4 - S_1S_4)C_5 - C_1C_2S_5 \tag{4.13}$$
$$x_y = (-S_1S_2C_4 + C_1S_4)C_5 - S_1S_2S_5 \tag{4.14}$$
$$x_z = C_2C_4C_5 - S_2S_5 \tag{4.15}$$
$$y_x = -(-C_1S_2C_4 - S_1S_4)S_5 - C_1C_2C_5 \tag{4.16}$$
$$y_y = -(-S_1S_2C_4 + C_1S_4)S_5 - S_1C_2C_5 \tag{4.17}$$
$$y_z = -C_2C_4S_5 - S_2C_5 \tag{4.18}$$
$$z_x = -C_1S_2S_4 + S_1C_4 \tag{4.19}$$
$$z_y = -S_1S_2S_4 - C_1C_4 \tag{4.20}$$
$$z_z = C_2S_4 \tag{4.21}$$
$$p_x = -C_1C_2d_3 - C_1S_2l_2 + S_1d_2 \tag{4.22}$$
$$p_y = -S_1C_2d_3 - S_1S_2l_2 - C_1d_2 \tag{4.23}$$
$$p_z = -S_2d_3 + l_2C_2 + d_1 \tag{4.24}$$

Ainda que o parâmetro l_2 é zero, o mesmo foi descrito junto com as outras variáveis simbólicas

(visando futuras calibrações e ajustes de parâmetros no robô).

Usando a ferramenta de simulação de robôs manipuladores Workspace 5.0 [1], foi construída uma simulação do robô. A figura 4.8 representa o modelo obtido usando esta ferramenta.

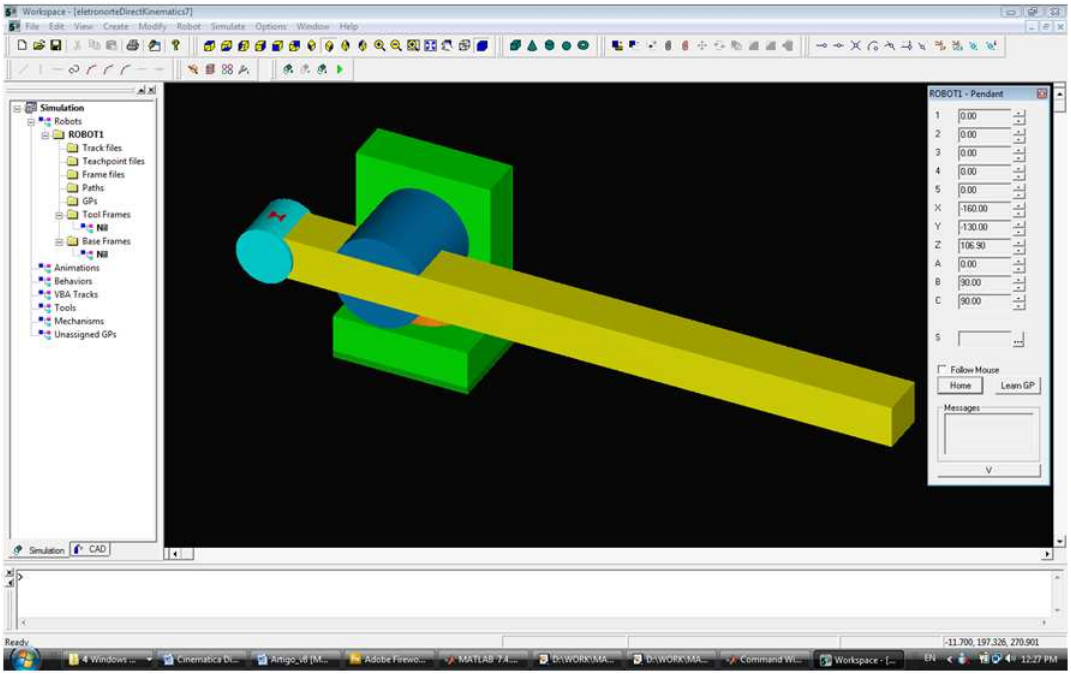


Figura 4.8: Entorno de trabalho do Workspace.

4.5 Implementação em FPGA da cinemática direta

A implementação em FPGA da cinemática direta do robô manipulador de configuração esférica, foi feita usando planejamento por restrições de tempo (TC - *Time-Constrained Scheduling*) e planejamento por restrições de espaço [58]. Na figura 4.9 é mostrada a arquitetura proposta. A arquitetura foi descrita em VHDL é controlada máquinas de estados finitos (FSM - *Finite State Machine*) usando a ferramenta de desenvolvimento Xilinx ISE 10.1.

Na figura 4.9 pode-se observar que os parâmetros da matriz são calculados da seguinte forma:

- No passo 3: X_z e Y_z.
- No passo 4: o parâmetro P_z.
- No passo 5: Z_z e P_x.
- No passo 6: P_y.
- No passo 7: Z_x e Z_y.
- No passo 10: X_x e Y_x.
- No passo 11: são calculados X_y e Y_y.

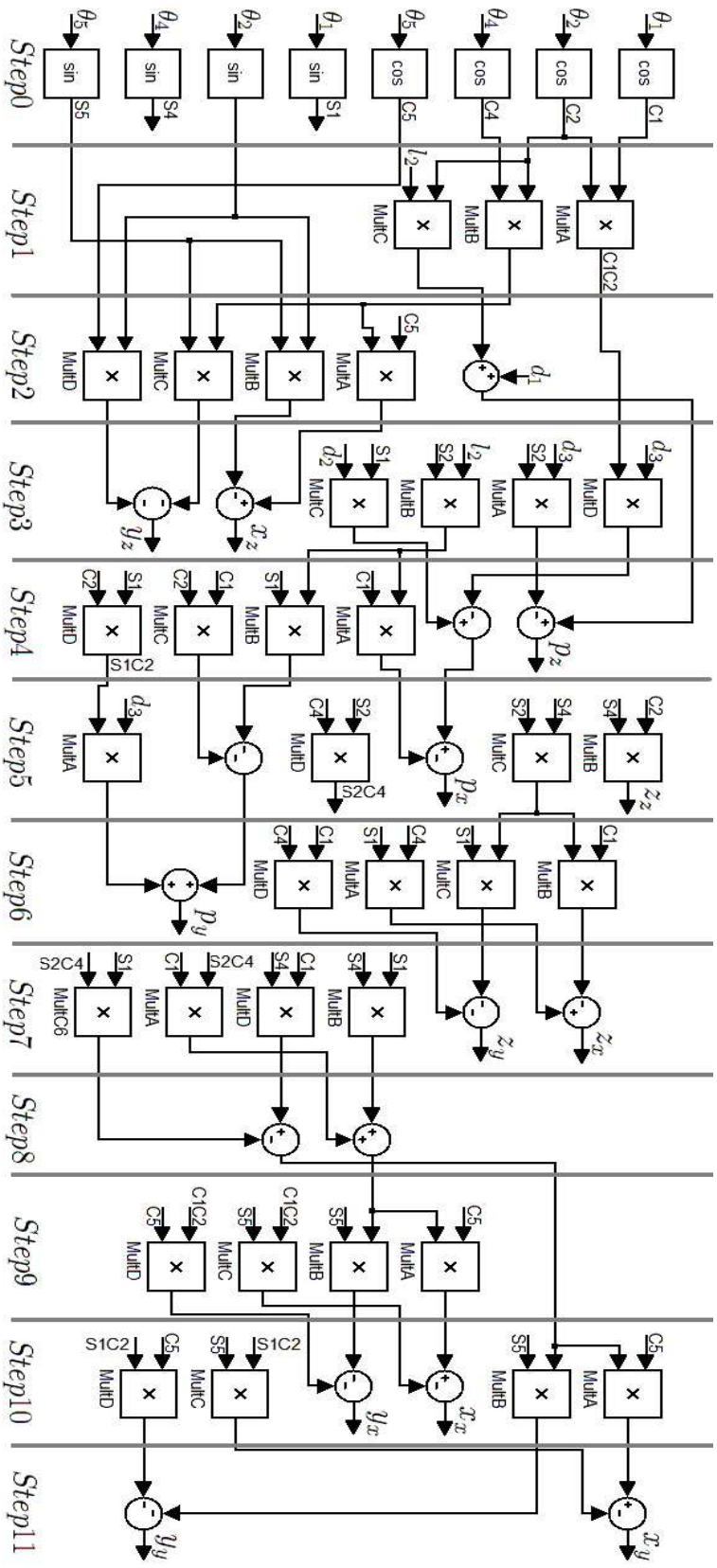


Figura 4.9: Arquitetura hardware para o cálculo da cinemática direta controlado por FSM.

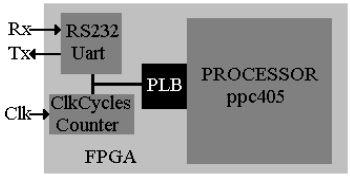


Figura 4.10: Arquitetura dentro da FPGA usando um processador Hard

```
43    xil_printf("Computation direct Kinematics in Software V.1\r\n");
44    xil_printf("Engineer:\r\n");
45    xil_printf("Diego Felipe Sanchez Gomez\r\n");
46
47    CLOCKCYCLESCOUNTER_mWriteSlaveReg0(clockcyclescounter, 0, 2); //reset
48    Cntr1 = CLOCKCYCLESCOUNTER_mReadSlaveReg1(clockcyclescounter,0);
49    xil_printf("Duration is %d\r\n", Cntr1);
50    CLOCKCYCLESCOUNTER_mWriteSlaveReg0(clockcyclescounter, 0, 1); //Enable
51
52    Xx = ( -cos(A)*sin(B)*cos(D) - sin(A)*sin(D) )*cos(E) - cos(A)*cos(B)*sin(E);
53    Xy = ( -sin(A)*sin(B)*cos(D) + cos(A)*sin(D) )*cos(E) - sin(A)*cos(B)*sin(E);
54    Xz = cos(B)*cos(D)*cos(E) - sin(B)*sin(E);
55
56    Yx = -( -cos(A)*sin(B)*cos(D) - sin(A)*sin(D) )*sin(E) - cos(A)*cos(B)*cos(E);
57    Yy = -( -sin(A)*sin(B)*cos(D) + cos(A)*sin(D) )*sin(E) - sin(A)*cos(B)*cos(E);
58    Yz = -cos(B)*cos(D)*sin(E) - sin(B)*cos(E);
59
60
61    Zx = -cos(A)*sin(B)*sin(D) + sin(A)*cos(D);
62    Zy = -sin(A)*sin(B)*sin(D) - cos(A)*cos(D);
63    Zz = cos(B)*sin(D);
64
65    Px = -cos(A)*cos(B)*D3 - cos(A)*L2*sin(B) + sin(A)*D2;
66    Py = -sin(A)*cos(B)*D3 - sin(A)*L2*sin(B) - cos(A)*D2;
67    Pz = -sin(B)*D3 + L2*cos(B) + D1;
68
69    CLOCKCYCLESCOUNTER_mWriteSlaveReg0(clockcyclescounter, 0, 0); //stop
70
71    Cntr13 = CLOCKCYCLESCOUNTER_mReadSlaveReg1(clockcyclescounter,0);
72
73    xil_printf("Duration is %d\r\n", Cntr13);
74
75    NXx = *((unsigned int *)(&Xx)); //flot2dec
76    NYx = *((unsigned int *)(&Yx)); //flot2dec
77    NZx = *((unsigned int *)(&Zx)); //flot2dec
78    NPx = *((unsigned int *)(&Px)); //flot2dec
```

[Platform Studio] | System Assembly View | Block Diagram | SoftDirKinematics.c*

Figura 4.11: Algoritmo descrito em C para o cálculo da cinemática direta no processador PowerPC.

Pode ser observado que a arquitetura proposta inclui o problema de definir o escalonamento de operações matemáticas no tempo (problema de *scheduling*), que pode variar dependendo do compromisso de otimizar o desempenho do circuito (*timming*) ou a área gasta na implementação do mesmo [58]. Neste trabalho se tiveram em conta as características da FPGA alvo do projeto para se ajustar o escalonamento de tal maneira que o circuito coubesse no dispositivo.

No intuito de comparar o desempenho da arquitetura proposta com uma implementação em software, foi escrito um programa em C que é executado sobre um processador *Hard* PowerPC usando como ferramenta de desenvolvimento a Xilinx Platform Studio. O citado programa implementa a cinemática direta do robô em software (dentro do processador embarcado). A arquitetura proposta para calcular a cinemática direta usando o processador *Hard* da FPGA é mostrada na figura 4.10.

Na arquitetura apresentada na figura 4.10, foram adicionados dois periféricos hardware ao barramento PLB do processador PowerPC: um periférico para contar o número de ciclos de relógio que demora calcular a cinemática direta, e outro periférico para realizar a comunicação com o computador via porta serial. A contagem do contador é habilitada e desabilitada por meio de instruções especificas de software e enviadas o periférico hardware através do barramento PLB. Esta metodologia permitiu comparar as duas implementações (software e hardware) nas mesmas condições: (a) a mesma FPGA e (b) a mesma freqüência de operação. O anterior permite ter uma comparação mais justa das duas metodologias de implementação, para o cálculo da cinemática direta.

Na figura 4.11 é mostrado um trecho do programa escrito em C. Deve ser salientado que na arquitetura do processador usado está sendo utilizada a unidade de ponto flutuante padrão do dispositivo. Na linha 47 é enviado o comando de reset para o periférico de contagem de ciclos. Na linha 48, o periférico é lido e no PC é comprovado que este inicie em zero (linha 49). Na linha 50 é habilitada a contagem de pulsos, e nas linhas posteriores às equações de cinemática direta são calculadas.

Uma vez que todos os parâmetros da matriz de transformação homogênea têm sido calculados, o contador de eventos de relógio é parado (linha 69), o número de pulsos contados é armazenado em uma variável (linha 71), para finalmente ser enviado pela porta serial ao computador (linha 73).

4.6 Conclusões do Capítulo

Neste capítulo justificou-se os motivos a implementação da cinemática direta de robôs manipuladores em hardware e o porque de usar aritmética de ponto flutuante. Depois foi feita uma revisão de trabalhos correlatos, onde foi percebido que na maioria de trabalhos prévios é usada uma abordagem de co-projeto de hardware/software. Nestes trabalhos, a implementação de complexas operações aritméticas é implementada usando o processador de propósito geral (GPP). Nas poucas arquiteturas de cálculo da cinemática de manipuladores em hardware não é reportado o uso de operadores aritméticos de ponto flutuante.

[1] http://www.workspace5.com/

Posteriormente, foi apresentado o modelo cinemático direto do robô caso de estudo, obtido partindo da atribuição de um marco de coordenadas para cada junta e os parâmetros de Denavit e Hartenberg. Adicionalmente, foi descrito o procedimento para a obtenção da matriz de transformação, usando um programa em MATLAB desenvolvido no contexto deste trabalho. Este programa permitiu calcular, automaticamente, a matriz de transformação homogênea partindo de um *script* com os parâmetros de Denavit e Hartenberg.

Finalmente, foi apresentada uma arquitetura totalmente em hardware usando aritmética de ponto flutuante para o cálculo da cinemática direta de um robô manipulador de configuração esférica. A arquitetura foi desenvolvida de forma a ser o mais rápida possível, levando em conta as restrições de recursos disponíveis na FPGA. Esta arquitetura executa em paralelo o cálculo de 8 operações de funções transcendentais (seno e cosseno), 4 operações de multiplicação e duas operações de soma/subtração. Adicionalmente, é apresentada uma arquitetura que usa um processador *hard* PowerPC dentro da FPGA, com o objetivo de fazer futuras comparações de tempo de processamento em hardware e em software.

Capítulo 5

Resultados

Os operadores de soma, subtração, multiplicação, divisão, raiz quadrada, seno, cosseno e arco-tangente, foram propostos e implementados em este trabalho, no intuito de construir uma biblioteca de operadores em ponto flutuante parametrizável pelo tamanho da palavra. A biblioteca construída pode ser usada em diferentes aplicações cientificas e de engenharia onde são necessários cálculos com alta precisão.

Neste capítulo são apresentados, os resultados de síntese lógica dos operadores implementados, tendo uma estimativa do consumo de recursos. Adicionalmente, são apresentados resultados de simulação, onde é mostrada a precisão com que um resultado é calculado, assim como o tempo gasto para calcular o mesmo. Para medir a precisão nos cálculos são comparados os resultados obtidos com o MATLAB com os resultados obtidos da simulação dos operadores implementados em VHDL, calculando o erro quadrático médio (MSE - *Mean Square Error*).

Seguidamente, são mostrados os resultados de sínteses e simulação da implementação da cinemática direta, assim como, uma análise da precisão com que são executados os cálculos.

Finalmente, é feita uma comparação do tempo de processamento do cálculo da cinemática direta das implementações em hardware (ver figura 4.9) e em software. Neste último caso, duas abordagens são apresentadas: (a) uma baseada em um processador *hard* PowerPC executada em uma FPGA e (b) outra em um computador usando o MATLAB.

5.1 Implementação dos Operadores em VHDL

Nesta seção apresentam-se com maior profundidade os operadores de soma, subtração, multiplicação, divisão, raiz quadrada, seno, cosseno e arco-tangente, baseados nas arquiteturas propostas no capítulo 3.

5.1.1 Unidade de Soma/Subtração

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*, *op_a*, *op_b* e *start_i*, esta última porta serve para dar início à operação desejada. Como portas de saída têm-se: *addsub_out* e *ready_as*. Esta última porta serve para indicar a finalização da operação. Na figura 5.1 é mostrada a declaração da entidade deste operador.

```
32  entity addsubfsm is
33      port (reset      :  in std_logic;
34            clk        :  in std_logic;
35            op         :  in std_logic;      -- 0:suma, 1:resta
36            op_a       :  in std_logic_vector(FP_WIDTH-1 downto 0);
37            op_b       :  in std_logic_vector(FP_WIDTH-1 downto 0);
38            start_i    :  in std_logic;
39            addsub_out : out std_logic_vector(FP_WIDTH-1 downto 0);
40            ready_as   : out std_logic);
41  end addsubfsm;
```

Figura 5.1: Declaração em VHDL da entidade da unidade de soma subtração

A construção de esta unidade foi obtida usando máquinas de estados finitos (*Finite State Machine* - FSM), contendo os estados de *waiting*, *add*, *sub* e *postnorm*. No estado de *waiting*, o circuito fica em 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). Seguidamente, é feita a operação lógica $op\_a(S)XOR(opXORop\_b(S))$. O resultado desta operação define o estado seguinte *add* ou *sub*. Na figura 5.2 é mostrado um diagrama da máquina de estados construída. Na figura 5.3 é mostrado parte do código em VHDL para implementar a soma.

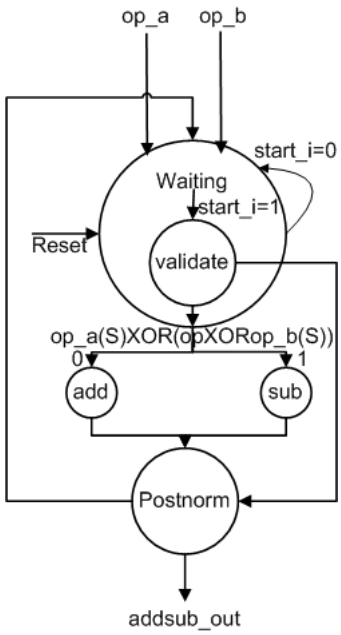


Figura 5.2: Diagrama da máquina de estados do operador soma/subtração.

Tanto no estado de *add* como de *sub*, internamente são feitas operações de comparação para se obter o maior dos operandos. Uma vez identificado o operando maior, na variável *Alinear* é calculado o número de vezes que a mantissa menor será deslocada a direita. Finalmente, a mantissa maior é somada com a mantissa menor previamente deslocada.

Para finalizar, no estado de *postnorm* são concatenados em um registro o sinal, o expoente

```
129  when add =>
130     CpiaSigno := op_a(FP_WIDTH-1);
131     IF op_a = op_b THEN
132        ExptMayor(EXP_WIDTH-1 downto 0) := op_a(FP_WIDTH-1 downto FRAC_WIDTH) + '1';
133        Alinear   := 1;
134        SumMntsas := '0' & op_b(FRAC_WIDTH-1 downto 0);
135     ELSIF op_a(FP_WIDTH-1 downto FRAC_WIDTH) = op_b(FP_WIDTH-1 downto FRAC_WIDTH) THEN
136        ExptMayor(EXP_WIDTH-1 downto 0) := op_a(FP_WIDTH-1 downto FRAC_WIDTH);
137        Alinear   := 0;
138        MntsMenor := op_b(FRAC_WIDTH-1 downto 0);
139        MntsMayor := '0' & op_a(FRAC_WIDTH-1 downto 0);
140        SumMntsas := '0' & MntsMenor + MntsMayor;
141     ELSIF op_a(FP_WIDTH-1 downto FRAC_WIDTH) > op_b(FP_WIDTH-1 downto FRAC_WIDTH) THEN
142        ExptMayor(EXP_WIDTH-1 downto 0) := op_a(FP_WIDTH-1 downto FRAC_WIDTH);
143        AuxExpon  := (op_a(FP_WIDTH-1 downto FRAC_WIDTH) - (op_b(FP_WIDTH-1 downto FRAC_WIDTH)));
144        Alinear   := CONV_INTEGER(AuxExpon);
145        CpiaMntsa := '1' & op_b(FRAC_WIDTH-1 downto 0);
146        DesplazarD(Alinear,CpiaMntsa,MntsMenor);
147        MntsMayor := '0' & op_a(FRAC_WIDTH-1 downto 0);
148        SumMntsas := '0' & MntsMenor + MntsMayor;
149     ELSE
150        ExptMayor(EXP_WIDTH-1 downto 0) := op_b(FP_WIDTH-1 downto FRAC_WIDTH);
151        AuxExpon  := (op_b(FP_WIDTH-1 downto FRAC_WIDTH) - (op_a(FP_WIDTH-1 downto FRAC_WIDTH)));
152        Alinear   := CONV_INTEGER(AuxExpon);
153        CpiaMntsa := '1' & op_a(FRAC_WIDTH-1 downto 0);
154        MntsMayor := '0' & op_b(FRAC_WIDTH-1 downto 0);
155        DesplazarD(Alinear,CpiaMntsa,MntsMenor);
156        SumMntsas := '0' & MntsMenor + MntsMayor;
157     END IF;
```

Figura 5.3: Parte do código em VHDL para implementar o operador soma.

e a mantissa resultante, e é ativada a porto de saída $ready\_as$. Note-se que o código está totalmente parametrizado, usando para este fim as constantes FP_WITDH, $FRAC\_WITDH$ e EXP_WIDTH armazenadas em um *package* (arquivo VHDL para definir tipos, constantes e funções). Neste caso os parâmetros armazenados no *package* podem ser facilmente modificados. Estas modificações são válidas para todos os arquivos do projeto em VHDL. Os parâmetros $FRAC\_WITDH$ e EXP_WIDTH representam o tamanho da mantissa e o tamanho do expoente, respectivamente. O parâmetro FP_WITDH é igual a $FRAC\_WITDH+EXP\_WIDTH$.

5.1.2 A unidade do Operador Multiplicação

Esta unidade permite calcular a multiplicação de dois números em ponto flutuante. Na figura 5.4 é mostrada a declaração da entidade deste operador. Nesta unidade temos como portas de entrada o *reset*, o *clock* para sincronizar a mesma, os dois operandos de entrada op_a e op_b e a porta que da início à realização dos cálculos $start\_i$. Como portas de saída têm-se o resultado da multiplicação (mul_out), assim como o sinal de indicação do final da operação ($ready\_mul$).

```
27  entity multiplicador_fp is
28      port (reset    :  in std_logic;
29            clk      :  in std_logic;
30            op_a     :  in std_logic_vector(FP_WIDTH-1 downto 0);
31            op_b     :  in std_logic_vector(FP_WIDTH-1 downto 0);
32            start_i  :  in std_logic;
33            mul_out  : out std_logic_vector(FP_WIDTH-1 downto 0);
34            ready_mul: out std_logic);
35  end multiplicador_fp;
```

Figura 5.4: Declaração da entidade do operador multiplicação

Esta unidade é descrita em VHDL usando uma FSM, tendo como estados *waiting*, *multiplier* e *postnorm*. Na figura 5.5 é mostrado o diagrama de estados desta unidade. No estado de *waiting*,

o circuito fica em espera até que a entrada *start_i* seja ativada. Neste momento, as entradas são validadas de igual forma a como foi descrito na unidade de soma/subtração.

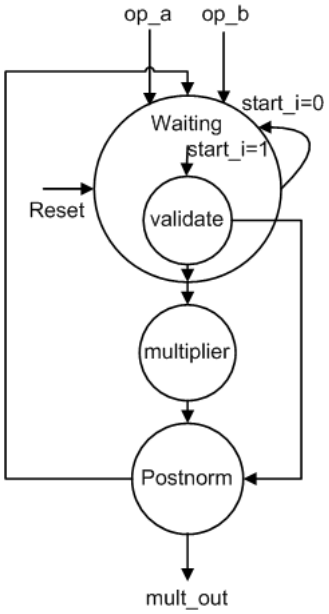


Figura 5.5: Diagrama de estados para a unidade de multiplicação

No estado *multiplier*, 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 (ver figura 5.6). No estado de *postnorm* são concatenados o sinal, o expoente e a mantissa resultante e, por último, o sinal de *ready_mul* é ativado em 1.

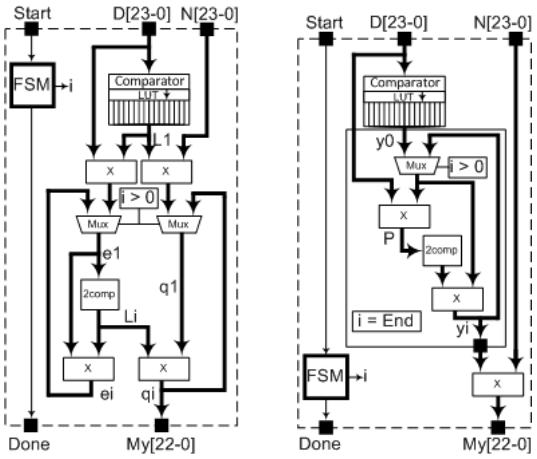
```
71   when multiplier =>
72      s_add_exp := ('0' & op_a(FP_WIDTH-1 downto FRAC_WIDTH)) + ('0' & op_b(FP_WIDTH-1 downto FRAC_WIDTH));
73      s_add_exp := s_add_exp - Bias;
74      s_mul_man := ('1' & op_a(FRAC_WIDTH-1 downto 0)) * ('1' & op_b(FRAC_WIDTH-1 downto 0));
75
76      if s_mul_man((FRAC_WIDTH*2)+1) = '1' then
77         s_add_exp := s_add_exp + '1';
78         s_mantisa := s_mul_man(FRAC_WIDTH*2 downto FRAC_WIDTH+1);
79      else
80         s_mantisa := s_mul_man((FRAC_WIDTH*2)-1 downto FRAC_WIDTH);
81      end if;
```

Figura 5.6: Estado *multiplier* da unidade de multiplicação

Comparando a dificuldade da arquitetura de soma/subtração com a unidade de multiplicação, encontra-se que esta última é mais simples de implementar, uma vez que a unidade de soma precisa de um ciclo *for* para fazer uma busca iterativa na operação de normalização. Além disso, na unidade de soma/subtração é necessário fazer operações de comparação para determinar o operando maior; entanto que, estas funções não são necessárias na operação de multiplicação.

5.1.3 A unidade do Operador Divisão

No capítulo 3 foi apresentada a arquitetura geral para o cálculo da divisão, e foram apresentados duas arquiteturas para se obter a mantissa resultante, uma baseada no algoritmo Goldschmidt e a outra no algoritmo Newton Raphson. Na figura 5.7, novamente são mostradas a arquiteturas



(a) Goldschmidt (b) Newton Raphson

Figura 5.7: Arquiteturas usadas no cálculo do cociente da mantissa

Goldschmidt e Newton-Raphson, respectivamente.

O cálculo do sinal e do expoente foi obtido similarmente a como foi feito na multiplicação. Em sguida trata-se com mais detalhe o cálculo da mantissa usando as arquiteturas da figura 5.7.

As duas arquiteturas apresentadas são controladas por uma FSM. A obtenção da mantissa do cociente resultante é calculada partindo de uma aproximação inicial a $1/D$ (onde D representa o denominador). Estas aproximações, em hardware são armazenadas usando uma memória ROM. Porém, encontra-se um problema no intuito de construir uma biblioteca parametrizável, já que, dependendo da largura da palavra que o usuário deseja, as sementes (aproximações) também mudam. De esta forma, o usuário deveria mudar o código toda vez que se deseja mudar o tamanho da palavra.

Este problema foi resolvido usando um gerador de código descrito em MATLAB, no qual são introduzidos os parâmetros de número de posições da LUT ($SLUT$), assim como o tamanho da mantissa desejada ($FMnt$). Este programa gera automaticamente o código VHDL da unidade de divisão. Na figura 5.8, mostra-se o gerador de código VHDL em MATLAB, para o operador total de Divisão (onde é gerado o código VHDL calculando tanto o sinal, o expoente e a mantissa) usando o algoritmo Goldschmidt.

Uma vez configurados o tamanho da LUT e mantissa desejados, é chamado o programa $FPDivisionVHD$ no qual, usando funções do MATLAB, um arquivo de saída com extensão vhd é construído, compatível com ferramentas da Xilinx e da Altera.

Para criar a LUT, a faixa de valores que pode tomar a mantissa ($1 > \mathrm{M} > 2$) é dividido em $SLUT$ partes iguais (esto significa que a LUT armazena $SLUT$ palavras). Depois, é gerada uma memória ROM de tamanho $SLUT$ contendo valores binários de tamanho $FMnt - 1$, já que, $\mathrm{ROM}(i)(FMnt)$ sempre é igual a 1 (portanto não e necessário armazenar esse bit). Na figura 5.9, é mostrada a memória ROM gerada.

A escolha da semente é feita concorrentemente usando comparadores. Na figura 5.10 mostra-se a implementação dos comparadores obtidos usando a ferramenta de geração de código, assim como

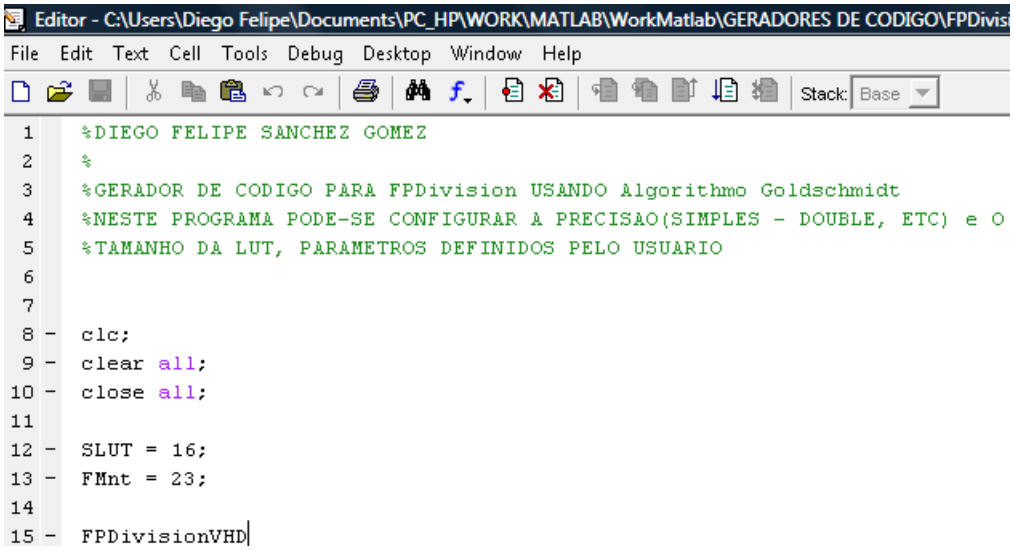


Figura 5.8: Gerador de código VHDL em MATLAB da unidade do operador divisão

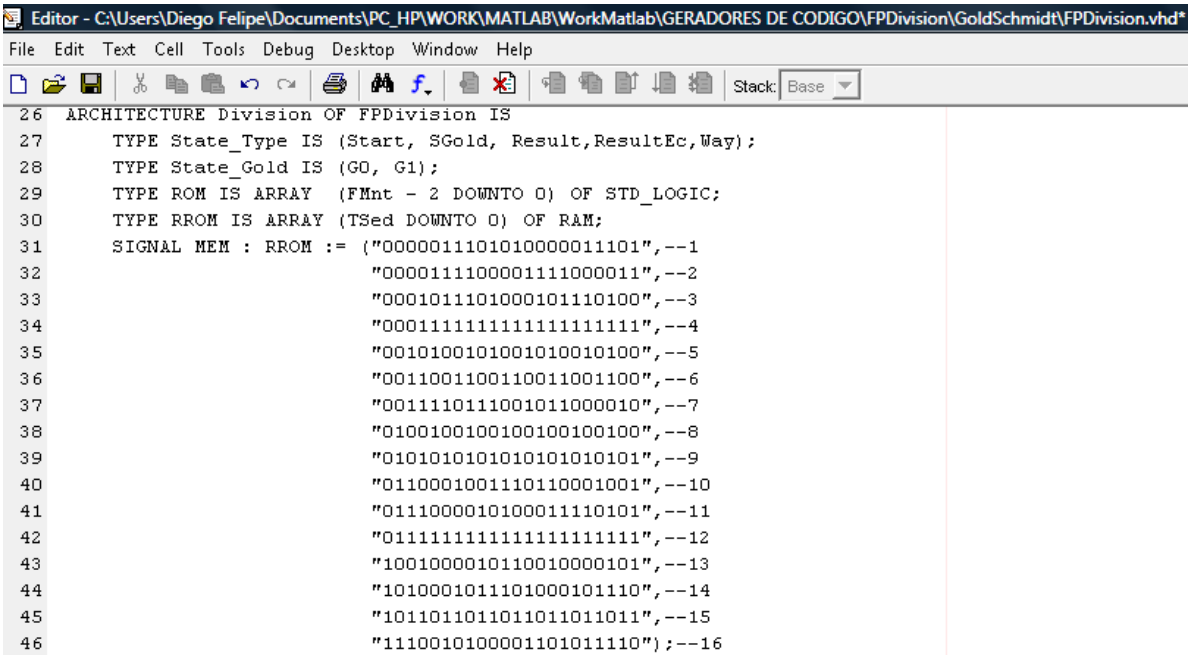
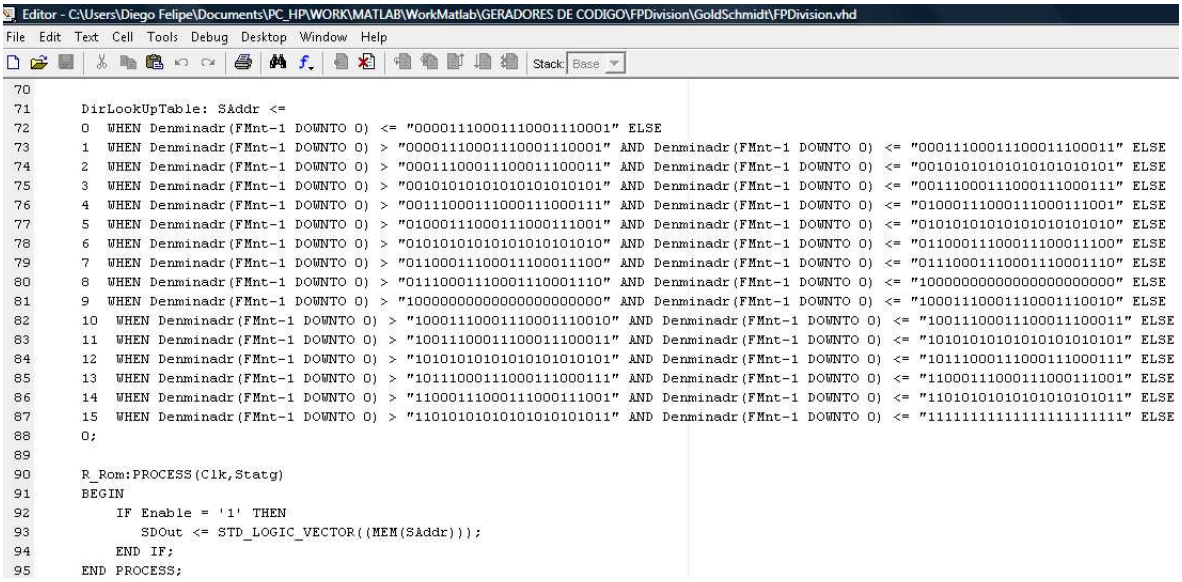


Figura 5.9: Memória ROM gerada

o processo de acesso à memória para extrair a semente.



```
Editor - C:\Users\Diego Felipe\Documents\PC_HP\WORK\MATLAB\WorkMatlab\GERADORES DE CODIGO\FPDivision\GoldSchmidt\FPDivision.vhd
File  Edit  Text  Cell  Tools  Debug  Desktop  Window  Help
Stack: Base
70
71      DirLookUpTable: SAddr <=
72      0  WHEN Denminadr(FMnt-1 DOWNTO 0) <= "00001110001110001110001" ELSE
73      1  WHEN Denminadr(FMnt-1 DOWNTO 0) > "00001110001110001110001" AND Denminadr(FMnt-1 DOWNTO 0) <= "00011100011100011100011" ELSE
74      2  WHEN Denminadr(FMnt-1 DOWNTO 0) > "00011100011100011100011" AND Denminadr(FMnt-1 DOWNTO 0) <= "00101010101010101010101" ELSE
75      3  WHEN Denminadr(FMnt-1 DOWNTO 0) > "00101010101010101010101" AND Denminadr(FMnt-1 DOWNTO 0) <= "00111000111000111000111" ELSE
76      4  WHEN Denminadr(FMnt-1 DOWNTO 0) > "00111000111000111000111" AND Denminadr(FMnt-1 DOWNTO 0) <= "01000111000111000111001" ELSE
77      5  WHEN Denminadr(FMnt-1 DOWNTO 0) > "01000111000111000111001" AND Denminadr(FMnt-1 DOWNTO 0) <= "01010101010101010101010" ELSE
78      6  WHEN Denminadr(FMnt-1 DOWNTO 0) > "01010101010101010101010" AND Denminadr(FMnt-1 DOWNTO 0) <= "01100011100011100011100" ELSE
79      7  WHEN Denminadr(FMnt-1 DOWNTO 0) > "01100011100011100011100" AND Denminadr(FMnt-1 DOWNTO 0) <= "01110001110001110001110" ELSE
80      8  WHEN Denminadr(FMnt-1 DOWNTO 0) > "01110001110001110001110" AND Denminadr(FMnt-1 DOWNTO 0) <= "10000000000000000000000" ELSE
81      9  WHEN Denminadr(FMnt-1 DOWNTO 0) > "10000000000000000000000" AND Denminadr(FMnt-1 DOWNTO 0) <= "10001110001110001110010" ELSE
82      10  WHEN Denminadr(FMnt-1 DOWNTO 0) > "10001110001110001110010" AND Denminadr(FMnt-1 DOWNTO 0) <= "10011100011100011100011" ELSE
83      11  WHEN Denminadr(FMnt-1 DOWNTO 0) > "10011100011100011100011" AND Denminadr(FMnt-1 DOWNTO 0) <= "10101010101010101010101" ELSE
84      12  WHEN Denminadr(FMnt-1 DOWNTO 0) > "10101010101010101010101" AND Denminadr(FMnt-1 DOWNTO 0) <= "10111000111000111000111" ELSE
85      13  WHEN Denminadr(FMnt-1 DOWNTO 0) > "10111000111000111000111" AND Denminadr(FMnt-1 DOWNTO 0) <= "11000111000111000111001" ELSE
86      14  WHEN Denminadr(FMnt-1 DOWNTO 0) > "11000111000111000111001" AND Denminadr(FMnt-1 DOWNTO 0) <= "11010101010101010101011" ELSE
87      15  WHEN Denminadr(FMnt-1 DOWNTO 0) > "11010101010101010101011" AND Denminadr(FMnt-1 DOWNTO 0) <= "11111111111111111111111" ELSE
88      0;
89
90      R_Rom:PROCESS(Clk,Statg)
91      BEGIN
92          IF Enable = '1' THEN
93             SDOut <= STD_LOGIC_VECTOR((MEM(SAddr)));
94          END IF;
95      END PROCESS;
```

Figura 5.10: Comparadores e processo de acesso a memória para extrair a semente em VHDL.

Um gerador de código similar foi criado para a implementação da unidade de divisão usando o algoritmo Newton-Raphsom. Na figura 5.11 e 5.12 mostra-se trechos do código gerado em VHDL para o cálculo da mantissa usando os algoritmos Goldschmidt e Newtom-Raphson, respetivamente. Estas implementações estão baseadas em máquinas destados finitos (FSMs).

```
127     PROCESS(Clk,StatG)
128         VARIABLE A1 : STD_LOGIC_VECTOR(FMnt*2 + 1 DOWNTO 0) := (OTHERS => '0');
129         VARIABLE A2 : STD_LOGIC_VECTOR(FMnt*2 + 1 DOWNTO 0) := (OTHERS => '0');
130         VARIABLE La : STD_LOGIC_VECTOR(FMnt DOWNTO 0) := (OTHERS => '0');
131         VARIABLE VQa : STD_LOGIC_VECTOR(FMnt DOWNTO 0) := (OTHERS => '0');
132         VARIABLE VEa : STD_LOGIC_VECTOR(FMnt DOWNTO 0) := (OTHERS => '0');
133     BEGIN
134         A1  := (OTHERS => '0');
135         VQa := (OTHERS => '0');
136         CASE StatG IS
137             WHEN GO =>
138                 A1 := ("01" & SDOut) * MntNumer;
139                 A2 := ("01" & SDOut) * MntDnmnd;
140                 VQa := A1(FMnt*2 DOWNTO FMnt);
141                 VEa := A2(FMnt*2 DOWNTO FMnt);
142                 La := NOT(VEa) + 1;
143             WHEN G1 =>
144                 A1 := Qa*Ls;
145                 A2 := Ea*Ls;
146                 VQa := A1(FMnt*2 DOWNTO FMnt);
147                 VEa := A2(FMnt*2 DOWNTO FMnt);
148                 La := NOT(VEa) + 1;
149         END CASE;
150        Qa <= VQa;
151         Ls <= La;
152         Ea <= VEa;
153     END PROCESS;
```

Figura 5.11: Processo em VHDL para o cálculo da mantissa usando o algoritmo Goldschmidt

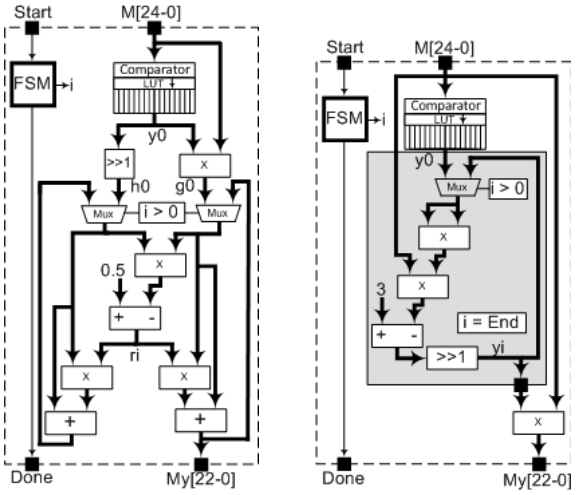
No processo mostrado na figura 5.12, o valor refinado do cociente $1/D$ é armazenado na variável E_a. Finalmente, em outro processo este valor é multiplicado pelo numerador, obtendo assim o valor do cociente.

```
121     OPer: PROCESS(Clk,StatG)
122         VARIABLE A1 : STD_LOGIC_VECTOR(FMnt*2 + 1 DOWNTO 0) := (OTHERS => '0');
123         VARIABLE A2 : STD_LOGIC_VECTOR(FMnt*2 + 1 DOWNTO 0) := (OTHERS => '0');
124         VARIABLE P  : STD_LOGIC_VECTOR(FMnt DOWNTO 0) := (OTHERS => '0');
125         VARIABLE Vax : STD_LOGIC_VECTOR(FMnt DOWNTO 0)   := (OTHERS => '0');
126         VARIABLE VEa : STD_LOGIC_VECTOR(FMnt DOWNTO 0)   := (OTHERS => '0');
127     BEGIN
128         P   := (OTHERS => '0');
129         A1  := (OTHERS => '0');
130         A2  := (OTHERS => '0');
131         Vax:= (OTHERS => '0');
132         VEa := (OTHERS => '0');
133         CASE StatG IS
134             WHEN G0 =>
135                 P   := (OTHERS => '0');
136                 A1 := (OTHERS => '0');
137                 A2 := (OTHERS => '0');
138                 Ea <= (OTHERS => '0');
139                 Vax:= (OTHERS => '0');
140                 VEa := ("01" & SDOut);
141             WHEN G1 =>
142                 A2 := MntDnmnd*Ea;
143                 P  := A2(FMnt*2 DOWNTO FMnt);
144                 Vax := NOT(P) + 1;
145                 A1   := Ea*Vax;
146                 VEa := A1(FMnt*2 DOWNTO FMnt);
147         END CASE;
148         Ea <= VEa;
149     END PROCESS;
```

Figura 5.12: Processo em VHDL para o cálculo da mantissa usando o algoritmo Newton-Raphson

5.1.4 Raiz Quadrada

No capítulo 3 foi apresentada a arquitetura geral para o cálculo da raiz quadrada. Neste caso, foram apresentadas duas arquiteturas para se obter a mantissa resultante, uma baseada no algoritmo Goldschmidt e a outra no algoritmo Newton-Raphson. Na figura 5.13, novamente são mostradas a arquiteturas Goldschmidt e Newton-Raphson, respectivamente.



(a) Goldschmidt (b) Newton Raphson

Figura 5.13: Arquiteturas usadas no cálculo da mantissa na operação de raiz quadrada

Similarmente à divisão, o cálculo da raiz quadrada foi obtido usando um gerador de código e, de esta forma, foi eliminado o problema de mudar a LUT toda vez que o tamanho da palavra for alterado. Finalmente, a implementação das arquiteturas propostas para o cálculo da mantissa na

```
120             CASE StatG IS
121                 WHEN G0 =>
122                     h   := (OTHERS => '0');
123                     r   := (OTHERS => '0');
124                     Ra  := (OTHERS => '0');
125                     Ga  <= (OTHERS => '0');
126                     Ha  <= (OTHERS => '0');
127                     LeerLkUp <= '1';
128                     g   := ReadLKY0 * Mantissa;
129                     Hs  := ('0' & ReadLKY0(FMnt DOWNTO 1));
130                     Gs  := g(FMnt*2 DOWNTO FMnt);
131                 WHEN G1 =>
132                     LeerLkUp <= '0';
133                     r   := Ga*Ha;
134                     r   := "00100000000000000000000000000000000000000000000000" - r; --0.5 - r
135                     IF r(FMnt*2 + 1) = '1' THEN
136                         r := NOT(r) + '1';
137                         Ra  := r(FMnt*2 DOWNTO FMnt);
138                         g(FMnt*2 + 1 DOWNTO 0)  := Ga*Ra;
139                         Gs  := g(FMnt*2 DOWNTO FMnt);
140                         Gs  := Ga - Gs;
141                         h(FMnt*2 + 1 DOWNTO 0)  := Ha*Ra;
142                         Hs  := h(FMnt*2 DOWNTO FMnt);
143                         Hs  := Ha - Hs;
144                     ELSE
145                         Ra  := r(FMnt*2 DOWNTO FMnt);
146                         g(FMnt*2 + 1 DOWNTO 0)  := Ga*Ra;
147                         Gs  := g(FMnt*2 DOWNTO FMnt);
148                         Gs  := Ga + Gs;
149                         h(FMnt*2 + 1 DOWNTO 0)  := Ha*Ra;
150                         Hs  := h(FMnt*2 DOWNTO FMnt);
151                         Hs  := Ha + Hs;
152                     END IF;
153             END CASE;
```

Figura 5.14: Processo em VHDL, para o cálculo da mantissa usando o algoritmo Goldschmidt

operação de raiz quadrada em VHDL é mostrada nas figuras 5.14 e 5.15.

5.2 Cálculo de Funções Transcendentais

Na implementação de funções transcendentais foi usada tanto a expansão em séries de Taylor como o algoritmo CORDIC. Estas duas implementações tomam vantagem das unidades de soma/subtração e multiplicação previamente construídas. Na arquitetura para o CORDIC foram usadas 3 unidades de soma/subtração em paralelo. Já na arquitetura para séries de Taylor foi usada uma unidade de soma/subtração, assim como uma unidade de multiplicação.

Usando estes algoritmos foram implementadas as funções seno, cosseno e arco-tangente. Como foi descrito no capítulo 3 a arquitetura CORDIC calcula concorrentemente (e em paralelo) as funções seno e cosseno. Para a arquitetura, usando expansão em séries de Taylor, foi proposta uma arquitetura generalizada na qual é escolhida a função a ser executada usando a porta *op*. Porém, de acordo com a arquitetura proposta para o cálculo da cinemática direta, as funções foram implementadas independentemente, e desta forma obtendo-se o paralelismo nos cálculos. Isso permite obter em paralelo as funções seno, cosseno e arco-tangente (ver figura 4.9).

No intuito de automatizar o processo de mudar o tamanho da palavra, também foi escrito em MATLAB um gerador de código para a obtenção das funções seno, cosseno e arco-tangente usando

```
104     PROCESS(Clk,StatG)
105         VARIABLE ax1: STD_LOGIC_VECTOR(FMnt*2 + 2 DOWNTO 0) := (OTHERS => '0');
106         VARIABLE Ys : STD_LOGIC_VECTOR(FMnt DOWNTO 0) := (OTHERS => '0');
107         VARIABLE AX2: STD_LOGIC_VECTOR(FMnt DOWNTO 0) := (OTHERS => '0');
108     BEGIN
109             ax1 := (OTHERS => '0');
110             Ys      := (OTHERS => '0');
111             AX2 := (OTHERS => '0');
112             LeerLkUp <= '0';
113             CASE StatG IS
114                 WHEN GO =>
115                     ax1 := (OTHERS => '0');
116                     Ys      := (OTHERS => '0');
117                     AX2 := (OTHERS => '0');
118                     LeerLkUp <= '1';
119                     Ys      := ReadLKY0;
120                 WHEN G1 =>
121                     LeerLkUp <= '0';
122                     ax1(FMnt*2 + 1 DOWNTO 0)    := Y*Y;
123                     Ax2 := ax1(FMnt*2 DOWNTO FMnt);                 --(yi^2)
124                     ax1 := Ax2 * Mantissa;                          --(R*(yi^2))
125                     ax1(FMnt*2 + 1 DOWNTO 0)    := "110000000000000000000000000000000000000000000000000" - ax1(FMnt*2 + 1 DOWNTO 0); --(3 - R*(yi^2))
126                     Ax2 := ax1(FMnt*2+1 DOWNTO FMnt+1);                 --(3 - R*(yi^2))
127                     ax1(FMnt*2 + 1 DOWNTO 0)    := Ax2*Y;           --yi*(3 - R*(yi^2))
128                     Ys      := '0' & ax1(FMnt*2-1 DOWNTO FMnt);     --0.5*yi*(3 - R*(yi^2))
129             END CASE;
130         Y <= Ys;
131     END PROCESS;
```

Figura 5.15: Processo em VHDL para o cálculo da mantissa usando o algoritmo Newton-Raphson

ambos algoritmos CORDIC e expansão em séries de Taylor.

Na figura 5.16, é mostrado o *script* em MATLAB do programa gerador da função arco-tangente usando o algoritmo CORDIC. Em este código, na variável EW é armazenado o valor correspondente ao tamanho do expoente, na variável $FMnt$ o tamanho da mantissa e na variável N o tamanho da LUT.

Este gerador entrega como saída os arquivos $FPatanCordic.vhd$ e $uRotation.vhd$. No primeiro arquivo, é implementada uma FSM encarregada de controlar cada micro-rotação, e no segundo arquivo é descrito em VHDL o procedimento realizado nas micro-rotações, baseado principalmente no controle dos três somadores em paralelo. Na figura 5.17, são mostradas as ligações aos três componentes de soma usados nas micro-rotações.

5.3 Síntese e Simulação Funcional dos Operadores

Os operadores aritméticos foram sintetizados utilizando a ferramenta ISE 10.1 e FPGAs da família Virtex II, fornecidos pela empresa Xilinx[1]. Todos os operadores são baseados no padrão IEEE 754 suportando exceções, e parametrizáveis pela largura de bits. Além disso, estes operadores foram simulados usando a ferramenta ModelSim 6.3g[2]. Finalmente, um ambiente de simulação para validar o comportamento das unidades foi desenvolvido em MATLAB[3]. Este programa cria vetores de teste em notação binária do correspondente número em ponto flutuante para diferentes larguras de bits. Estes vetores são passados às unidades correspondentes no ModelSim e, finalmente, os resultados obtidos são analisados usando um decodificador de binário para notação em ponto flutuante em MATLAB. Neste caso, os resultados das operações em MATLAB usando aritmética de ponto flutuante de 64 bits foram usados como estimador estatístico.

Os resultados de síntese em diferentes precisões das unidades de soma/subtração e multiplicação

[1] `http://www.xilinx.com/`

[2] `http://www.model.com/`

[3] `http://www.mathworks.com/`

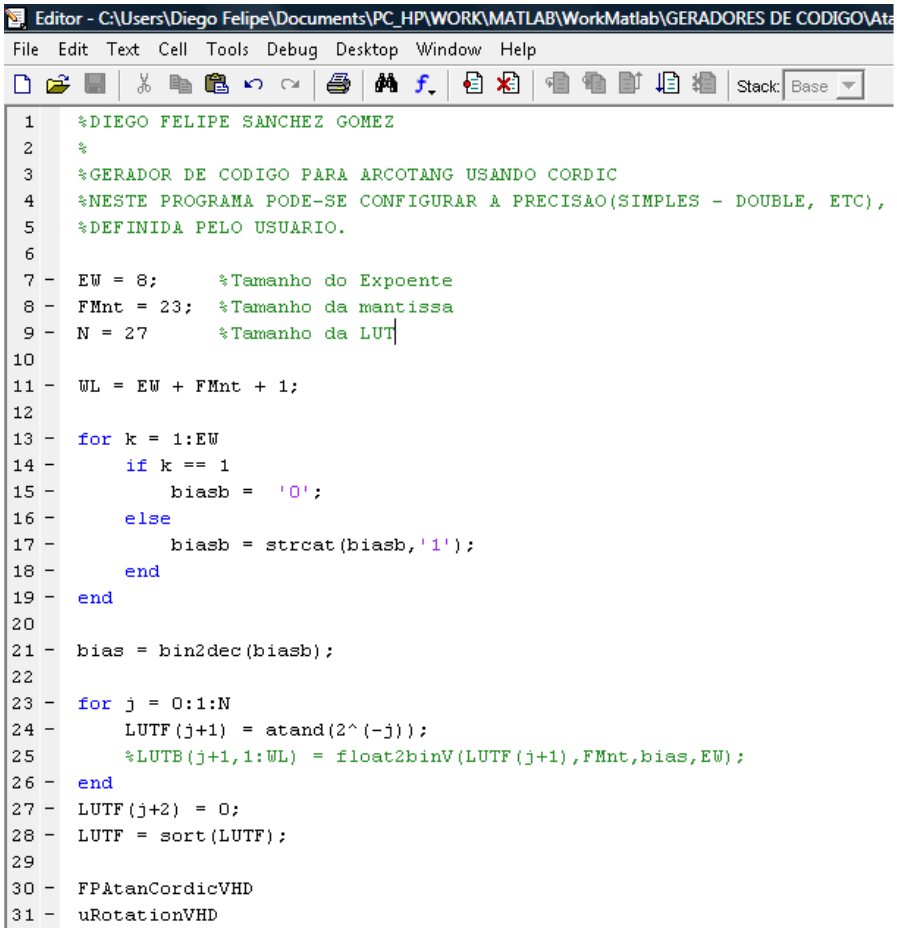


Figura 5.16: Gerador de código da função arco-tangente usando o algoritmo CORDIC

```
 96      X_addsub : FPAddSubFSM
 97      PORT MAP(clk    => clk,
 98               Reset  => Reset,
 99               Enable => Enable_u,
100               op     => ASgnX, -- 0:suma, 1:resta
101               op_a   => Xin,
102               op_b   => Yaux,
103               addsub_out => Xrslt,
104               ready_as   => XRdy);
105
106      ASgnY <= ('1' XOR SgnY);
107
108      Y_addsub : FPAddSubFSM
109      PORT MAP(clk    => clk,
110               Reset  => Reset,
111               Enable=> Enable_u,
112               op => ASgnY, -- 0:suma, 1:resta
113               op_a   => Yin,
114               op_b   => Xaux,
115               addsub_out => Yrslt,
116               ready_as   => YRdy);
117
118      ASgnZ <= ASgnY;
119      opAux <= STD_LOGIC_VECTOR(MEM(Iter));
120
121      Z_addsub : FPAddSubFSM
122      PORT MAP(clk    => clk,
123               Reset  => Reset,
124               Enable=> Enable_u,
125               op => ASgnZ, -- 0:suma, 1:resta
126               op_a   => Zin,
127               op_b   => opAux,
128               addsub_out => Zrslt,
129               ready_as   => ZRdy);
```

Figura 5.17: Ligações das tres unidades de soma usadas na implementação das micro-rotações.

são mostrados na tabela 5.1. Como era de esperar, na medida em que o tamanho da palavra (*bit-width*) aumenta apresenta-se um maior consumo de recursos e uma queda na freqüência de operação.

A unidade de soma/subtração tem um maior consumo de recursos do que a unidade de multiplicação. Porém, a unidade de soma/subtração não consume multiplicadores de ponto fixo disponíveis na FPGA (máximo 144 para uma Virtex2 XCV6000).

Tabela 5.1: Resultados de síntese unidades de soma/subtração e multiplicação. Virtex2 XCV6000

Bit-width	FP-Multiplier Core				FP-Add/sub Core		
(Man-Exp)	Slices 33792	LUTs 67584	Freq MHz	Multiplicadores 18×18	Slices 33792	LUTs 67584	Freq MHz
16 (5,10)	47	83	281	1	277	504	204
24 (6,17)	84	152	240	1	559	1006	172
32 (8,23)	121	223	231	4	821	1472	167
48 (9,38)	245	461	283	9	1509	2693	164
64 (11,52)	404	780	280	15	2843	5029	161

Na tabela 5.2, mostram-se os resultados de síntese da unidade de divisão utilizando os algoritmos Goldschmidt e Newton-Raphson, com uma LUT com 8 sementes de aproximações inicias (8 palavras). Pode-se observar, que a unidade baseada no algoritmo Newton-Raphson apresenta um menor consumo de recursos, tanto em área como na utilização de multiplicadores embarcados na FPGA.

Tabela 5.2: Resultados de síntese das unidades de divisão com LUT 8. Virtex2 XCV6000

Bit-width	FP Div Goldschmidt				FP Div Newton-Raphson			
(Ex,Mant)	Slices 33792	LUTs 67584	Mult 18×18	Freq MHz	Slices 33792	LUTs 67584	Mult 18×18	Freq MHz
16(5, 10)	124	219	4	312	104	185	3	399
24(6,17)	224	413	4	335	211	393	3	430
32(8,23)	328	615	16	338	288	539	12	427
48(9,38)	740	1402	36	336	616	1165	27	427
64(11,52)	1216	2343	54	324	1002	1926	45	430

Na tabela 5.3 são mostrados os resultados de síntese da unidade de raiz quadrada usando os algoritmos Goldschmidt e Newton-Raphson. Os resultados foram obtidos para uma LUT com 16 aproximações inicias. Observa-se que tanto as unidade de divisão como na de raiz quadrada baseadas no algoritmo Newton-Raphson apresentam um menor consumo de recursos que as unidades baseadas no algoritmo Goldschmidt. Além disto, as implementações baseadas neste algoritmo utilizam menos multiplicadores. O consumo de multiplicadores internos da FPGA é um parâmetro que o projetista deve minimizar, já que estes encontram-se em quantidades limitadas, e poderia restringir o número de unidades que podem ser usadas em paralelo.

Na tabela 5.4, mostram-se os resultados de síntese das unidades de cálculo das funções transcendentais seno, cosseno e arco-tangente usando tanto o algoritmo CORDIC como a expansão em

Tabela 5.3: Resultados de síntese das unidades de raiz quadrada com LUT 16. Virtex2 XCV6000

Bit-width	FP Sqrt Goldschmidt				FP Sqrt Newton-Raphson			
(Ex,Mant)	Slices 33792	LUTs 67584	Mult 18×18	Freq MHz	Slices 33792	LUTs 67584	Mult 18×18	Freq MHz
16(5, 10)	125	230	6	415	83	150	4	423
24(6,17)	304	568	7	423	191	345	6	423
32(8,23)	419	789	24	423	276	512	16	423
48(9,38)	1001	1908	54	423	648	1230	36	423
64(11,52)	1893	3646	90	420	1278	2454	60	420

séries de Taylor. Os operadores foram divididos em duas arquiteturas: (a) uma para o cálculo da funções seno e cosseno (FPCordic sin / cos e FPTaylor sin / cos) e (b) outra para o cálculo da função arco-tangente (FPCordic atan e FPTaylor atan). Esta divisão foi feita devido ao fato do algoritmo CORDIC calcular ao mesmo tempo as funções seno e cosseno.

Os resultados de síntese (ver tabela 5.4) mostram um menor consumo de recursos das unidades baseadas na expansão em séries de Taylor. As unidades baseadas no algoritmo CORDIC consumem quase o dobro de recursos do FPGA que a unidade baseada em séries de Taylor. Porém, as unidades baseadas no algoritmo CORDIC não usam multiplicadores embarcados no FPGA. Outro ponto a favor do algoritmo CORDIC para o cálculo das funções seno e cosseno é que estas são calculadas em paralelo usando os mesmos recursos. No caso do cálculo destas funções usando expansão em séries de Taylor só pode ser calculada uma função de cada vez. Na seguinte seção são mostrados os resultados de simulação tendo em conta parâmetros como tempo de processamento e precisão nos cálculos, que permitirão avaliar o desempenho das implementações.

5.3.1 Resultados de Simulação

Na tabela 5.5, são mostrados os resultados do erro quadrático médio (MSE), das unidade de soma/subtração e multiplicação, para diferentes larguras de palavra usando 100 valores randômicos na faixa de valores entre -1000 a 1000. Como era de esperar, entre mais bits sejam usados na representação do número se obtêm uma maior precisão nos cálculos.

As unidades de soma/subtração e multiplicação tardam um ciclo de relógio entre o evento de inicializar a operação ($start\_i = 1$) e obter o resultado ($ready = 1$).

Na tabela 5.6 é mostrado o MSE das unidades de divisão e raiz quadrada em aritmética de precisão simples, mudando a tamanho da *look-up table* (aproximações inicias). Como era de se esperar, o consumo de recursos aumenta proporcionalmente ao tamanho da *look-up table*. Porém, *look-up table* maiores não garantem maior precisão nos cálculos. Este comportamento é identificado tanto nas implementações baseadas no algoritmo Goldschmidt como no algoritmo Newton-Raphson.

No caso da divisão, o menor valor de MSE é obtido para uma LUT de 8 posições usando o algoritmo Goldschmidt e 16 posições usando o algoritmo Newton-Raphson. Ainda que o algoritmo baseado no algoritmo Newton-Raphson precisa de uma LUT maior para obter a mesma ordem de magnitude no MSE, esta implementação continua sendo mais econômica enquanto ao consumo de

Tabela 5.4: Resultados de síntese funções transcendentais. Virtex2 XCV6000

Bit-width (Exp, Man)	Função Implementada	Slices 33792	LUTs 67584	Multiplicadores 18×18	Freq MHz
24(6,17)	FPCordic sin/cos	1670	3183	0	35
	FPCordic atan	1479	2835	0	40
	FPTaylor sin/cos	845	1546	3	82
	FPTaylor atan	808	1485	3	33
28(7,20)	FPCordic sin/cos	2621	5024	0	34
	FPCordic atan	2014	3831	0	40
	FPTaylor sin/cos	1282	2382	12	82
	FPTaylor atan	1123	2085	12	32
32(8,23)	FPCordic sin/cos	3269	6299	0	30
	FPCordic atan	2266	4327	0	37
	FPTaylor sin/cos	1469	2748	12	53
	FPTaylor atan	1416	2646	12	32
43(11,31)	FPCordic sin/cos	7736	9078	0	30
	FPCordic atan	3261	6276	0	36
	FPTaylor sin/cos	2143	4052	12	40
	FPTaylor atan	1894	3584	12	32
64(11,31)	FPCordic sin/cos	9229	17659	0	28
	FPCordic atan	9214	17712	0	30
	FPTaylor sin/cos	3731	7056	27	39
	FPTaylor atan	3689	6989	27	27

Tabela 5.5: MSE das unidades de soma/subtração e multiplicação para diferentes larguras de bits

Bit-width (Exp, Man)	MSE Add/sub	MSE Mult
16(5, 10)	2,24E-03	2,21E-02
24(6,17)	2,27E-07	9,53E-04
32(8,23)	2,76E-11	1,53E-07
48(9,38)	1,31E-15	3,96E-12
64(11,52)	1,26E-17	5,87E-16

Tabela 5.6: MSE das unidades de divisão e raiz quadrada mudando o tamanho da LUT

Operação	Tamanho da LUT	Goldschmidt			Newton-Raphson		
		Slices	LUTs	MSE	Slices	LUTs	MSE
Div	4	303	568	3,63E-11	263	492	3,41E-11
	8	328	615	3,67E-13	288	539	1,19E-11
	16	403	753	3,71E-12	362	677	3,82E-13
	32	559	1033	3,87E-12	519	957	3,81E-13
	64	865	1587	1,57E-12	865	1587	1,57E-12
Sqrt	8	396	745	2,14E-11	253	468	1,81E-11
	16	419	789	8,62E-12	276	512	8,48E-12
	32	458	860	7,45E-12	315	583	6,71E-12
	64	520	969	7,24E-12	377	692	6.25E-12
	128	616	1143	1,24E-11	473	866	5.93E-12

recursos (devido ao menor consumo de multiplicadores embarcados).

No caso da raiz quadrada, um valor aceitável do MSE é obtido para uma LUT de 16 posições tanto para a implementação baseada no algoritmo Goldschmidt como para a implementação baseada no algoritmo Newton-Raphson. Além disso, esta última implementação apresenta um menor consumo de recursos do FPGA.

Na tabela 5.7 é mostrado o MSE mudando o número de iterações das unidades de divisão e raiz quadrada para LUTs de 8 palavras no caso da divisão (usando o algoritmo Goldschmidt) e uma LUT de 16 posições nos outros casos. Além do anterior, nesta tabela é mostrado o número de ciclos de relógio necessários para produzir um resultado dependendo do número de iterações. Pode-se observar que a menor ordem de magnitude em todos os casos é obtido para 3 iterações. Depois da terceira iteração o MSE aumenta lentamente devido ao erro propagado da aproximação inicial (erros propagados por arredondamento) [53].

Tabela 5.7: MSE das unidades de divisão e raiz quadrada mudando o número de iterações

itereções	Ciclos de Relógio	Goldschmidt		Newton-Raphson	
		Div	Sqrt	Div	Sqrt
3	3	3,67E-13	8,62E-12	3,82E-13	8,48E-12
5	4	7,71E-12	2,70E-11	3,72E-13	6,96E-12
7	5	3,46E-11	3,63E-11	3,72E-13	6,31E-12
9	6	8,11E-11	3,63E-11	3,72E-13	6,30E-12
11	7	1,47E-10	3,63E-11	3,72E-13	6,96E-12

Na tabela 5.8 é feito um resumo dos melhores resultados obtidos usando uma representação em aritmética de precisão simples das operações de divisão e raiz quadrada usando os algoritmos Goldschmidt e Newton-Raphson. Fazendo um análise das variáveis: (a) consumo de recursos do FPGA, (b) vazão (Freqüência/(Ciclos de relógio)) e (c) precisão, encontra-se que a principal vantagem das implementações baseada no algoritmo Newton-Raphson é o baixo consumo de multiplicadores embarcados no FPGA.

Tabela 5.8: Comparação dos algoritmos Goldschmidt e Newton-Raphson

Algoritmo		Goldschmidt		Newton-Raphson	
Operação		Div	Sqrt	Div	Sqrt
Consumo de Recursos	Slices	328	419	362	276
	LUTs	615	789	677	512
	Mult 18×18	16	24	12	16
Vazão [Mresults/s]	(Freqüência/ Latência)	112.67	141.00	142.33	141.00
MSE		3,67E-13	8,62E-12	3,82E-13	8,48E-12

Na tabela 5.9 é mostrado o MSE obtido para formato em aritmética de precisão simples (8,23) mudando o número de iterações (Micro-Rotações/Potências) dos operadores seno, cosseno e arco-tangente baseadas no algoritmo CORDIC e expansão em séries de Taylor. Além disto, é mostrado o número de ciclos de relógio necessários para obter o resultado (latência), dependendo do número de iterações.

Tabela 5.9: MSE das funções transcendentais mudando o número de iterações

Algoritmo	Micro-Rotações Potências	MSE sin $[-\frac{\pi}{2};\frac{\pi}{2}]$	MSE cos $[-\frac{\pi}{2};\frac{\pi}{2}]$	MSE atan $[-100;100]$	Latência
FPCORDIC	10	1,85E-07	1,90E-07	3,93E-04	34
	11	3,54E-08	3,51E-08	2,58E-04	37
	12	7,90E-09	8,51E-09	7,37E-05	40
	13	1,71E-09	2,37E-09	1,54E-05	43
	14	6,68E-10	5,74E-10	3,58E-06	46
	15	1,50E-10	1,64E-10	7,93E-07	49
	16	3,63E-11	4,44E-11	2,06E-07	52
	17	1,03E-11	1,16E-11	6,26E-08	55
	18	2,60E-12	2,49E-12	1,70E-08	58
	19	6,51E-13	7,02E-13	3,73E-09	61
	20	1,97E-13	1,99E-13	1,24E-09	64
FPTaylor	2	1,59E-06	3,52E-05	0,00137	11
	3	1,56E-09	5,58E-08	0,00136	15
	4	7,01E-13	3,62E-11	0,00136	19
	5	3,13E-15	1,15E-14	0,00136	23
	6	2,13E-14	5,91E-15	0,00136	27

Pode-se observar (tabela 5.9) que as funções seno e cosseno baseadas em expansão em séries de Taylor, mostram um melhor comportamento tendo em conta latência e MSE. Além do anterior, a unidade FPTaylor precisa de poucas iterações para obter um MSE com uma ordem de magnitude baixa. O menor MSE (na ordem de E-15) para implementação baseada na expansão em séries de Taylor é obtido em 23 ciclos de relógio para o seno e 27 ciclos de relógio para o cosseno. Por outro lado, o menor MSE (na ordem de E-13) para a implementação baseada em expansão em series de Taylor é obtida em 64 ciclos de relógio.

O MSE da unidade FPTaylor para calcular o seno e cosseno (ver tabela 5.9) apresenta uma rápida convergência para 5 e 6 iterações. Note-se que o MSE para função seno com 5 iterações (grau do polinômio igual a 11 x^{11}) é menor que o MSE para 6 iterações (grau do polinômio igual a 13 x^{13})). Este comportamento é conhecido como fenômeno de Runge's no qual é provado que o erro da interpolação aumenta quando o grau do polinômio aumenta.

Para o operador arco-tangente encontra-se um melhor comportamento do MSE da unidade baseada no algoritmo CORDIC (MSE de E-9 para 20 micro-rotações). Na figura 5.18 mostra-se uma comparação entre os resultados obtidos para a função arco-tangente baseada na expansão em series de Taylor e os resultados obtidos para esta função com o MATLAB. Pode-se observar uma aproximação satisfatória para valores de entrada entre $[-0,8$ a $0,8]$, obtendo um MSE na ordem de 10^{-19}. Porém, para valores de entrada entre $[\pm 0,8$ ate $\pm 1]$, a arquitetura FPTaylor produz erros na ordem de 10^{-3}. Valores de entrada maiores que 1 produzem divergências do MSE, devido a que as potências do polinômio crescem mais rapidamente que seus respectivos coeficientes (ver equação 3.26c).

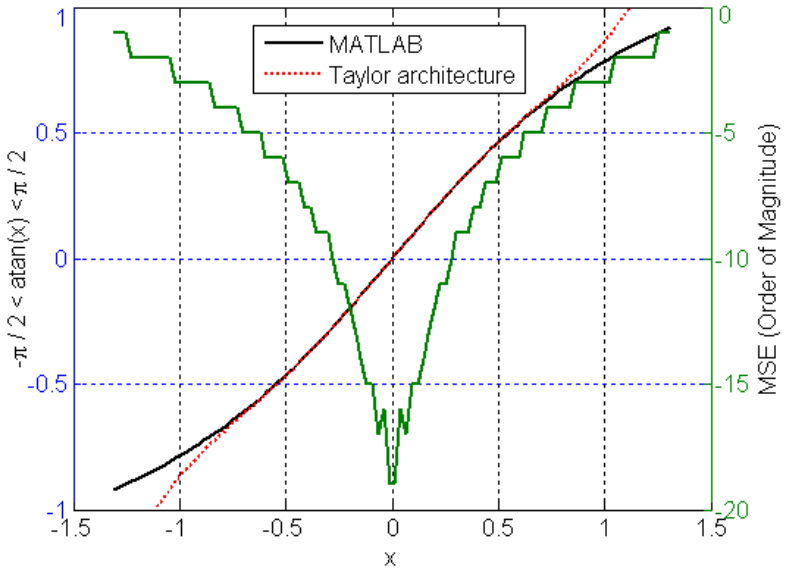


Figura 5.18: MSE da unidade FPTaylor para a função arco-tangente

Na tabela 5.10 são resumidos os melhores resultados obtidos usando uma representação em aritmética de precisão simples dos operadores seno, cosseno e arco-tangente usando os algoritmos CORDIC (20 micro-rotações) e expansão em series de Taylor (5 iterações).

Fazendo um análise das variáveis: (a) consumo de recursos do FPGA, (b) vazão (Freqüência/(Ciclos de relógio)) e (c) precisão, encontra-se que um melhor desempenho das unidades seno e cosseno é para aquelas baseadas em expansão em séries de Taylor. Está arquitetura apresenta um baixa latência e em conseqüência uma alta vazão em comparação com a unidade baseada no algoritmo CORDIC.

Para o cálculo da função arco-tangente, encontra-se um melhor desempenho usando o algoritmo CORDIC, ainda que este algoritmo tem uma baixa vazão, o CORDIC apresenta uma melhor precisão do que a mesma implementação baseada em séries de Taylor.

Tabela 5.10: Comparção dos algoritmos CORDIC e expansão en séries de Taylor

Algoritmo		CORDIC		Taylor	
Operação		sin/cos	atan	sin/cos	atan
Consumo de Recursos	Slices	3269	2266	1469	1416
	LUTs	6299	4327	2748	2646
	Mult 18×18	0	0	12	12
Vazão [Mresults/s]	(Freqüência/ Latência)	0,47	0,58	1,96	1,18
MSE		1,97E-13	1,24E-09	3,13E-15	0,00136

5.4 Resultados de Síntese e Simulação da Implementação da Cinemática Direta

A arquitetura proposta (ver figura 4.9), foi implementada usando os operadores aritméticos em ponto flutuante desenvolvidos no contexto deste trabalho. Esta arquitetura requer de 8 unidades para calcular funções transcendentais, 4 para calcular a função seno e outras 4 para calcular a função cosseno. Adicionalmente, são usados 4 multiplicadores e duas unidades de soma/subtração. Para o cálculo das funções transcendentais seno e cosseno foram escolhidas as unidades baseadas em séries de Taylor (menor tempo de processamento do que as unidades baseadas em CORDIC), com 5 iterações (polinômio de ordem 10 e 11 para o cosseno e seno, respectivamente).

No intuito de se obter o paralelismo no cálculo das funções seno e cosseno, estas foram implementadas como unidades independentes.

Na tabela 5.11 são mostrados os resultados de síntese da arquitetura apresentada em 4.9 para calcular a cinemática direta do robô manipulador, caso de estudo. Estes resultados foram obtidos para diferentes larguras de palavra (expoente, mantissa). Como esperado, operações executadas com larguras de palavra maiores são mais custosas em termos de área. Porém, a freqüência de operação do circuito é maior para larguras de palavra menores.

Tabela 5.11: Resultados de síntese da arquitetura para o cálculo da cinemática direta. Virtex2 XC2VP30

Bit-width (Exp,Man)	Slices 13696	LUTs 27392	Mult18x18 136	Freq MHz
32(8,23)	11112	20282	48	231.107
43(11,31)	15634	28401	48	201.694
64(11,52)	35477	63421	180	193.456

Na tabela 5.11 pode ser observado que as representações com larguras de palavra 43 e 64 excedem o máximo de recursos disponíveis na FPGA Virtex 2 XC2VP30.

Para calcular o MSE nos cálculos da cinemática direta foram usados 100 valores de entrada, na faixa de -90^o até 90^o para os ângulos θ_1, θ_4 e θ_5. A faixa de valores de teste para o ângulo θ_2 , é de -35^o to 20^o. Finalmente, para a junta prismática D_3 são passados os valores na faixa de 160

mm to 760 mm. Os vetores de teste para cada ângulo foram criados usando a mesma ferramenta desenvolvida em MATLAB usada na criação dos vetores de teste dos operadores aritméticos anteriormente descritos. Estes vetores são passados a uma simulação funcional descrita no ModelSim. Os resultados entregados pela simulação são comparados com os resultados obtidos em MATLAB para, finalmente, se obter o MSE.

Na tabela 5.12 são mostrados os resultados do MSE em diferentes larguras de bits. Pode-se observar que a ordem de magnitude do MSE não tem mudanças significativas. Simulações usando tamanhos de palavra menores que 32 bits não apresentam resultados satisfatórios, devido à saturação nos operadores de ponto flutuante.

Tabela 5.12: MSE em diferente tamanhos de palavra da arquitetura para o cálculo da cinemática direta

Bit-width	axis	x	y	z	p
32(8,23)	x	3.61E-12	1.44E-11	1.75E-14	5.42E-06
	y	9.08E-13	3.55E-12	1.06E-14	1.33E-06
	z	1.44E-11	3.63E-12	4.46E-12	2.07E-10
43(11,31)	x	3.63E-12	1.45E-11	1.62E-14	5.44E-06
	y	8.88E-13	3.58E-12	3.28E-16	1.34E-06
	z	1.45E-11	3.64E-12	4.45E-12	7.07E-11
64(11,52)	x	3.63E-12	1.45E-11	1.62E-14	5.44E-06
	y	8.88E-13	3.58E-12	3.29E-16	1.34E-06
	z	1.45E-11	3.64E-12	4.45E-12	7.07E-11

A implementação em software usando o processador PowerPC(usando a arquitetura mostrada na figura 4.10) foi validada com um programa desenvolvido em MATLAB. Este programa usa a porta serial para enviar as posições das juntas e recebe os valores de (posição, orientação e tempo de processamento em ciclos de relógio). Esta simulação tem por objetivo comparar o tempo de processamento entre a implementação em software (usando o processador *hard*) e a implementação totalmente em hardware.

Na tabela 5.13 é mostrado o número de ciclos de relógio necessários para calcular cada variável (x_x), e o passo (*step*) no qual as memas são obtidas. O total da cinemática direta é calculada em 67 ciclos de relógio (0,67 μs). A mesma implementação em software no processador PowerPC usando aritmética de precisão simples demora 1,61036 ms. Pode-se observar que a arquitetura de hardware é na ordem de 2400 vezes mais rápida do que a implementação em software (no processador PowerPC). Este resultado mostra um aumento considerável na velocidade de processamento do cálculo da cinemática direta usando a arquitetura de mapeamento proposta.

Na tabela 5.14 é mostrado o tempo de processamento que levou o cálculo da cinemática direta usando três abordagens: (a) a arquitetura proposta totalmente em hardware, (b) uma implementação em software usando um processador *hard* PowerPC e (c) uma implementação em MATLAB. A implementação no PowerPC e a arquitetura em hardware proposta trabalham com um relógio de 100 MHz. A implementação em MATLAB é executada usando um processador AMD Atlhon dual-core de 2.1 GHz, 3 GB de RAM e o sistema operacional Windows Vista.

Tabela 5.13: Latência e passo no qual é calculada cada variável

Passo	Variável	Ciclos de Relógio
3	x_z, y_z	43
4	p_z	46
5	z_z, p_x	49
6	p_y	52
7	z_x, z_y	55
10	x_x, y_x	64
11	x_y, y_y	67

Tabela 5.14: Latência das implementações hardware e software (PowerPC e MATLAB))

Hardware	PowerPC	MATLAB
0.67μs	1.61036ms	13μs

Como pode-se observar a implementação hardware executa todos os cálculos em $0,67\mu$s. Esta abordagem é aproximadamente 19 vezes mais rápida que a implementação em MATLAB e ao redor de 2400 vezes mais rápida que a implementação no PowerPC.

5.5 Conclusões do Capítulo

Neste capítulo foi descrita a implementação de cada operador, em aritmética de ponto flutuante, em VHDL. Também foram apresentados geradores de código para os operadores divisão, raiz quadrada (usando Goldschmidt como Newton-Raphson) e funções transcendentais seno, cosseno e arco-tangente (usando CORDIC e expansão em séries de Taylor). Estes geradores, descritos em MATLAB, permitem obter automaticamente o código VHDL da função a ser implementada para qualquer tamanho de palavra que o usuário deseje.

Por outro lado, foram apresentados os resultados de síntese e simulação de cada operador, para diferentes larguras de palavra. Estes resultados foram usados para fazer uma análise tendo em conta parâmetros do projeto como: (a) consumo de recursos no FPGA, (b) vazão (freqüência/latência) e (c) precisão. Esta análise permitiu determinar qual dos algoritmos Newton-Raphson ou Goldschmidt (no caso da divisão e raiz quadrada) e CORDIC ou expansão em séries de Taylor (no caso das funções transcendentais) apresenta um melhor desempenho.

Adicionalmente, foram mostrados os resultados de síntese e simulação da arquitetura proposta para o cálculo totalmente em hardware da cinemática direta de um robô manipulador de configuração esférica com cinco graus de liberdade. Esta arquitetura foi construída usando os operadores em aritmética de ponto flutuante desenvolvidos no contexto de este trabalho, no qual para o cálculo das funções transcendentais foram usados os operadores baseados em expansão em séries de Taylor.

A arquitetura para o cálculo da cinemática direta descrita totalmente em hardware, apresentou

uma alta velocidade de processamento se comparada com a implementação em software (baseada em um processador *hard* PowerPC executado em um FPGA). Adicionalmente, a implementação em hardware apresentou um melhor desempenho do que uma implementação em software executada em um computador usando o MATLAB, ainda que o computador usa um relógio com uma freqüência muito maior.

Este trabalho deu origem a três publicações internacionais (referências [35, 55, 59]), em [35], foram publicados os operadores de soma/subtração, multiplicação divisão e raiz quadrada, mostrando os resultados de síntese e simulação. Em este trabalho foi usado só o algoritmo Goldschmidt para o cálculo dos operadores de divisão e raiz quadrada.

No trabalho [55] são mostrados os operadores para o cálculo de funções transcendentais em ponto flutuante usando os algoritmos CORDIC e expansão em séries de Taylor. Em esta publicação adicionalmente são mostrados os resultados de síntese e simulação e uma comparação de desempenho das duas arquiteturas (CORDIC e Taylor).

Na referência [59] é mostrado a arquitetura para o calculo da cinemática direta em hardware. Adicionalmente, é feito um comparativo entre o tempo de processamento que leva o cálculo da cinemática direta em hardware e em software (PowerPC e MATLAB). Alem disso, são mostrados os resultados de síntese e simulação para o cálculo da cinemática direta em diferentes larguras de bits.

Capítulo 6

Conclusões e Sugestões para Trabalhos Futuros

6.1 Considerações Gerais

Esta dissertação apresenta uma descrição em VHDL dos operadores aritméticos, em ponto flutuante (soma/subtração, multiplicação divisão, raiz quadrada) e funções transcendentais (seno, cosseno e arco-tangente), assim como a utilização dos mesmos no cálculo da cinemática direta de um robô manipulador de cinco graus de liberdade. Todos os operadores são parametrizáveis pelo tamanho da palavra (tanto expoente como mantissa), sendo possível trabalhar nos formatos de aritmética de precisão simples e dupla como em formatos customizados pelo usuário. Estas características dão ao projetista a flexibilidade de explorar um formato em ponto flutuante com o tamanho de palavra otimizado para uma aplicação especifica.

Os operadores de soma/subtração e multiplicação apresentam uma latência de 1 ciclo de relógio para obter o resultado, com um MSE satisfatório na ordem de magnitude de E-11 e E-7. Os resultados de síntese mostram que a unidade de soma/subtração apresenta um maior consumo de recursos do que a unidade de multiplicação.

Na implementação dos operadores raiz quadrada e divisão foram consideradas duas abordagens baseadas nos algoritmos Goldschmidt e Newton-Raphson. Estes algoritmos fazem uso dos multiplicadores embarcados na FPGA e de memórias para armazenar as aproximações inicias. Com estas arquiteturas são obtidas altas vazões (*throughput*). Usando a arquitetura Goldschmidt para o cálculo da divisão são obtidos 112,67 mega resultados por segundo, e para raiz quadrada 141 mega resultados por segundo. Para a implementação usando o algoritmo Newton-Raphson são obtidos 142,33 e 141 mega resultados por segundo para divisão e raiz quadrada, respectivamente.

Fazendo uma análise entre o consumo de recursos, vazão e precisão para as unidades de divisão e raiz quadrada encontrou-se que a implementação baseada no algoritmo Newton-Raphson apresenta um melhor desempenho. Os resultados de síntese mostram que esta arquitetura tem como principal vantagem sobre a arquitetura baseada no algoritmo Goldschmidt o menor consumo de multiplicadores embarcados no FPGA. Os resultados de síntese mostram similar comportamento

para as duas implementações em termos da precisão nos cálculos.

Para o cálculo das funções transcendentais foram consideradas as abordagens baseadas nos algoritmos CORDIC e expansão em séries de Taylor. Estes algoritmos fazem uso das unidades de soma/subtração e multiplicação em ponto flutuante desenvolvidas no contexto deste projeto. A unidade baseada no algoritmo CORDIC usa três unidades de soma/subtração executadas em paralelo e uma LUT para armazenar os valores pré-calculados de $\arctan(2^n)$. Por outro lado, a unidade baseada em séries de Taylor usa uma unidade de soma/subtração, uma unidade de multiplicação e uma LUT para armazenar os valores pré-calculados dos coeficientes $\frac{1}{n!}$ e $\frac{1}{n}$.

Fazendo uma análise entre o consumo de recursos, vazão e precisão para o cálculo das funções transcendentais encontrou-se que a implementação baseada em Taylor apresenta um melhor desempenho para o cálculo das funções seno e cosseno, porém, para o cálculo do arco-tangente recomenda-se o uso da implementação baseada no algoritmo CORDIC.

Os resultados de síntese e simulação mostram que a arquitetura baseada em series de Taylor tem como principal vantagem um menor consumo de recursos e uma boa precisão (na ordem de E-15 para seno e cosseno em aritmética de precisão simples) obtida em poucos ciclos de relógio (6 iterações, 27 ciclos de relógio). Usando CORDIC para calcular estas funções tem-se um maior consumo de recursos e uma alta latência (20 micro-rotações, 64 ciclos de relógio) para chegar a uma precisão na mesma ordem de magnitude do que a implementação baseada em Taylor.

Para o cálculo da função arco-tangente a arquitetura baseada em Taylor não apresenta uma boa precisão, portanto, definitivamente não é recomendável fazer uso da mesma em futuras aplicações. Ainda que esta função baseada em CORDIC tenha uma alta latência, esta arquitetura apresenta uma boa precisão.

Os operadores como divisão, raiz quadrada e funções transcendentais nas quais são necessárias memórias para implementar as arquiteturas, apresentam maior dificuldade para serem parametrizados. Já que, toda vez que o usuário deseje modificar o tamanho da palavra, os valores armazenados na memória deveriam ser mudados. Este problema foi resolvido criando geradores de código descritos em MATLAB que permitem obter, automaticamente, o código em VHDL da função requerida, para qualquer tamanho da palavra que o usuário deseje de uma forma simples e rápida.

A biblioteca de operadores aritméticos em ponto flutuante descrita neste trabalho apresenta vantagens sobre bibliotecas concorrentes como a apresentada em [28] enquanto menor consumo de recursos, alta vazão, propriedade de ser parametrizável, a independência dos operadores e inclusão de funções transcendentais. Além disto, é feito em estudo da precisão nos cálculos da biblioteca desenvolvida.

Uma implementação totalmente em hardware para o cálculo da cinemática direta usando os operadores aritméticos de ponto flutuante desenvolvidos permitiu validar a funcionalidade dos operadores. Em muitas aplicações científicas e de engenharia que requerem de um elevado número de operações aritméticas, estas são implementadas em GPPs ou implementadas em FPGAs usando aritmética de ponto fixo. Em algumas aplicações é preferivel implementar as operações usando processadores $soft$ ou *hard* ou implementações diretamente em hardware, porém, usando aritmética de precisão simples. Com esta biblioteca pretende-se incentivar o uso das FPGAs em aplicações

que requerem de elevadas operações aritméticas executadas com alta precisão, explorando as capacidades de paralelismo destes dispositivos, tais como aplicações em robótica e mecatrônica em geral.

Na implementação da cinemática direta foram usadas 8 unidades para o cálculo das funções seno e cosseno (4 para o cosseno e 4 para o seno), 4 multiplicadores e 2 unidades de soma/subtração todas estas unidades funcionando em paralelo. Os resultados da síntese mostram que implementar este tipo de aplicações usando potencialidades de paralelismo é fatível para as FPGAs disponíveis.

Por outro lado, comparando a implementação totalmente em hardware com implementações em software, é obtida uma considerável aceleração para o cálculo da cinemática direta usando a implementação em hardware. Esta implementação é em volta de 2400 vezes mais rápida que a o algoritmo executado em um processador *hard* embarcado na FPGA PowerPC. Este processador usa uma freqüência de relógio de 100 MHz. Comparada com uma implementação executada num computador usando o MATLAB usando um processador AMD Atlhon dual-core de 2,1 GHz, 3 GB de RAM e o sistema operacional Windows Vista a implementação hardware é 19 vezes mais rápida.

Estas altas velocidades de processamento mostram a potencialidade tanto das FPGAs como dos operadores implementados. Esta biblioteca pode ser usada em uma grande quantidade de aplicações cientificas e de engenharia que possuem restrições de tempo de processamento e precisa-se de algoritmos executados com alta precisão.

Além das aplicações em robótica de manipuladores, esta biblioteca vem sendo usada por alunos de doutorado e mestrado em sistemas mecatrônicos em aplicações de otimização baseadas em técnicas bio-inspiradas, processamento digital de sinais e na implementação de algoritmos para o cálculo de matrizes inversas.

6.2 Sugestões para trabalhos futuros

As perspectivas de trabalhos futuros relacionadas a seguir, visam alcançar um nível plenamente funcional dos operadores da arquitetura proposta para o cálculo da cinemática direta. Melhorar as funcionalidades já existentes e acrescentar novos operadores, assim como, integrar a arquitetura para o cálculo da cinemática direta ao robô e implementar em hardware o cálculo da cinemática inversa e outros algoritmos nos quais seja importante ter altas velocidades de processamento.

Acrescentar funcionalidades na biblioteca de operadores

No intuito de melhorar a precisão nos cálculos, deve ser acrescentado, em cada operador, a possibilidade de trabalhar com números desnormailzados. Também, devem ser tidos em conta outros algoritmos que além de produzir uma boa precisão tenham uma rápida convergência.

Os operadores implementados são baseados em arquiteturas seqüenciais, o que os faz adequados por seu baixo consumo de recursos e alto desempenho, em aplicações onde têm-se uma baixa taxa de fluxo de dados. Para aplicações com um alto fluxo de dados (telecomunicações por exemplo)

é recomendável a criação dos operadores baseados em arquiteturas *pipeline*. Estas permitirão diminuir o consumo de recursos.

No intuito de aumentar a flexibilidade da biblioteca, pretende-se acrescentar novos operadores em ponto flutuante tais como comparadores, funções raiz quadrada inversa e o inverso da divisão, conversores de ponto fixo para ponto flutuante e vice-versa, funções exponenciais e logarítmicas, entre outras. Além disto, a criação de funcionalidades com maior complexidade como algoritmos de inversão de matrizes, transformadas rápidas de Fourier, entre outras.

Criação de interface com o usuário

No intuito de facilitar a criação dos operadores de ponto flutuante no tamanho de palavra desejado pelo usuário, inicialmente poderia-se fazer uma interface gráfica amigável, em uma ferramenta como C, na qual o usuário não tenha que adquirir outras ferramentas como o MATLAB.

Integrar a arquitetura hardware para o cálculo da cinemática direta ao robô

Finalmente, pretende-se integrar a arquitetura hardware para o cálculo da cinemática direta ao robô para o qual têm se desenvolvido arquiteturas em VHDL para o controle de pulso e direção dos motores de passo usados no robô. Porém, está faltando desenvolver os módulos para coletar os dados fornecidos pelos *enconders* fixados em cada junta.

Criação de uma ferramenta de otimização automática de sistemas de equações usando os operadores descritos

Na implementação de algoritmos que envolvem um elevado número de sistemas de equações, o projetista se enfrenta ao problema de otimização em tempo e consumo de recursos do FPGA. Esta tarefa dificilmente é otimizável de forma manual. A criação de uma ferramenta que possa automatizar a criação do sistema de equações segundo parâmetros definidos pelo usuário como: dispositivo FPGA a ser usado e tipo de otimização (consumo de recursos, tempo, consumo de energia, entre outros) permitiria liberar o projetista desta complexa tarefa.

A figura 6.1 mostra a arquitetura implementada para o cálculo da cinemática direta, com a restrição de utilização de apenas 4 multiplicadores e dois unidades soma/subtração operando em paralelo. Além disto, 8 unidades de cálculo de funções transcendentais. Esta arquitetura poderia ser melhorada segundo o critério de tempo, de modo a executar as operações no instante seguinte à disponibilização dos resultados resultando na figura 6.2.

Comparando as duas arquiteturas nota-se uma redução no número de passo (6 passos a menos), e como conseqüência isto levaria a uma redução no tempo de processamento. Porém, pode-se observar que esta arquitetura tem um maior consumo de recursos, utilizando até 15 multiplicadores, 8 unidades de soma/subtração e 8 unidades para o cálculo de funções transcendentais todas estas unidades operando em simultâneo.

Comparando as duas arquiteturas pode-se observar que outros parâmetros poderiam ser oti-

mizados, tais como consumo de energia, freqüência de operação, consumo de recursos do FPGA, entre outros. Desta forma, mostra-se a necessidade da criação de uma ferramenta deste tipo.

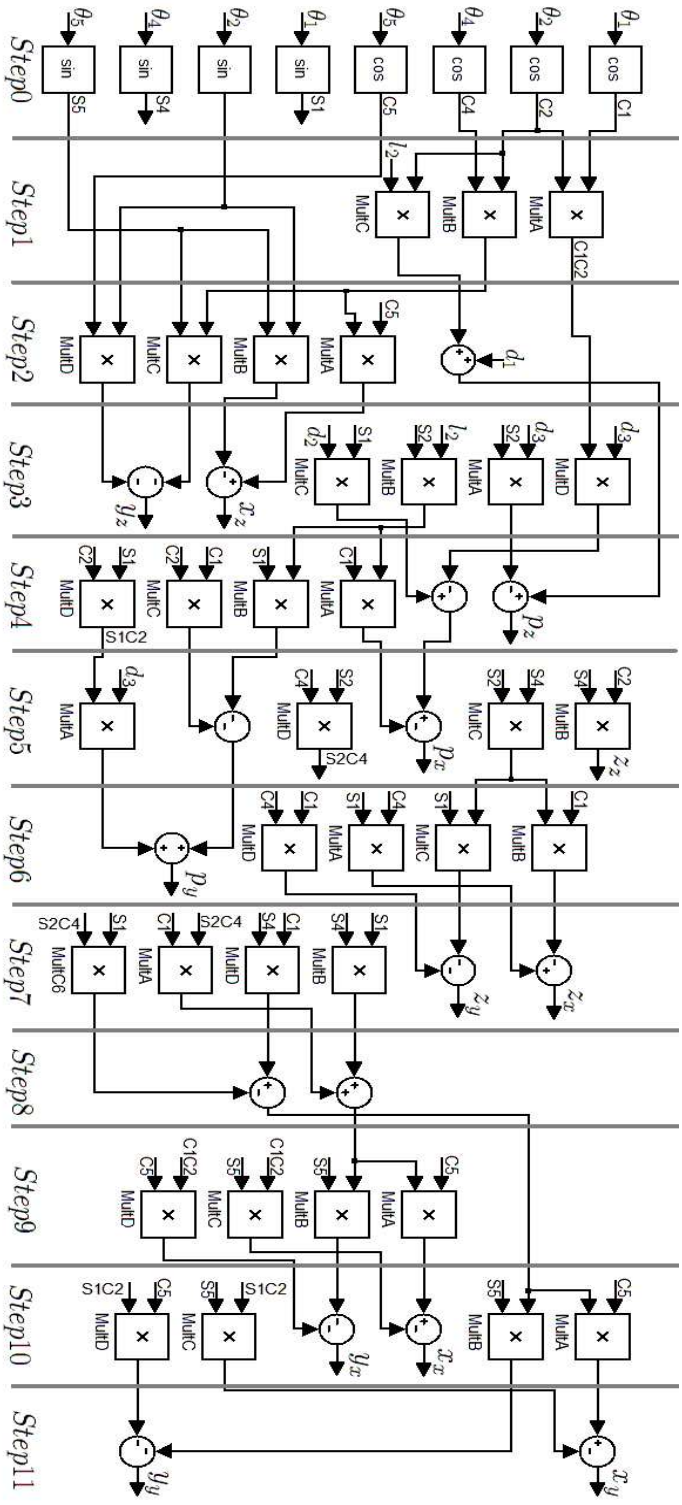


Figura 6.1: Arquitetura hardware usada no cálculo da cinemática direta.

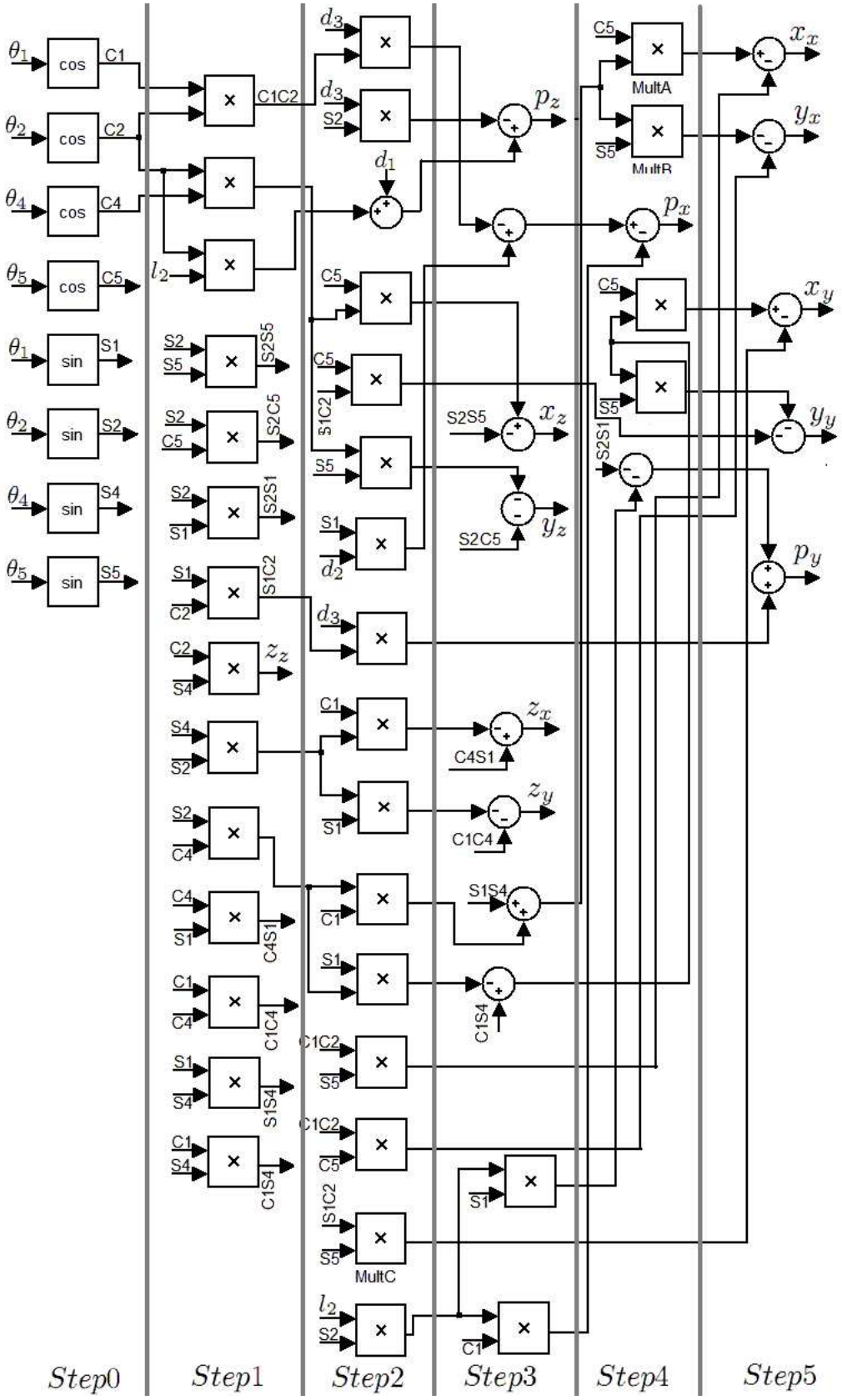


Figura 6.2: Proposta de arquitetura hardware otimizada em tempo para o cálculo da cinemática direta.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] ROSINGER, H.-P. *Connecting Customized IP to the MicroBlaze Soft Processor Using the Fast Simplex Link (FSL) Channel*. Disponível em *http://www.xilinx.com/support/documentation/application_ notes/xapp529.pdf*, 2004. Xilinx Corporation.

[2] HARTENSTEIN, R.; KAISERSLAUTERN, T. Basics of reconfigurable computing. In: HENKEL, J.; PARAMESWARAN, S. (Ed.). *Designing Embedded Processors*. Netherlands: Springer, 2007. p. 451–501.

[3] DUBEY, R. *Introduction to Embedded System Design Using Field Programmable Gate Arrays*. London: Springer-Verlag, 2009.

[4] KUNG, Y. et al. FPGA-implementation of inverse kinematics and servo controller for robot manipulator. In: IEEE. *Proceedings on Robotics and Biomimetics*. China, 2006. p. 1163–1168.

[5] WANG, X. *Variable Precision Floating-point Divide and Square Root for EfficientFPGA Implementation of Image and Signal Processing Algorithms*. Tese (Doutorado) — Northeastern University, 2007.

[6] HARTENSTEIN, R. The microprocessor is no longer general purpose: why futurereconfigurable platforms will win. In: IEEE. *Prooceedings Conference on Innovative Systems in Silicon*. USA, 1997. p. 2–12.

[7] ZOMAYA, A. Parallel processing for robot dynamics computations. *Parallel Computing*, v. 21, n. 4, p. 649–668, Nov. 1995.

[8] ANDRAKA, R. A survey of CORDIC algorithms for FPGA based computers. In: ACM. *Proceedings Symposium on Field Programmable Gate Arrays*. USA, 2000. p. 253 – 258.

[9] SHUANG-YAN, C. et al. Design and implementation of a 64/32-bit floating-point division, reciprocal, square root, and inverse square root unit. In: IEEE. *Prooceedings on Solid-State and Integrated Circuit Tech*. China, 2006. p. 1976–1979.

[10] WANG, X.; NELSON, B. Trade-offs of designing floating-point division and square root on virtex fpgas. In: IEEE. *Proceedings Symposium on Field Programmable Custom Computer*. USA, 2003. p. 195–203.

[11] WIRES, K.; SCHULTE, M. Reciprocal square root units with operand modification and multiplication. *Journal of VLSI Signal Processing Systems*, v. 42, n. 3, p. 257–272, Mar. 2006.

[12] BARRIENTOS, A. et al. *Fundamentos de Robótica*. España: McGraw-Hill, 2007.

[13] MCKERROW, P. *Introduction to Robotics*. Malásia: Addison-Wesley.

[14] SANCHEZ, D.; SARRIA, C. *ASPECTOS NUMÉRICOS EN LA IMPLEMENTACIÓN DE CONTROLADORES DIGITALES UTILIZANDO HARDWARE EN EL LAZO DE SIMULACIÓN*. Trabalho de Graduação. Universidad del Cauca, Colombia, 2005.

[15] RAMIREZ, P.; BARROS, E.; LIMA, S. *PROCESSAMENTO DIGITAL DE SINAIS - PROJETO E ANÁLISE DE SISTEMAS*. Brasil: Bookman, 2004.

[16] PIñEIRO, J. *Algorithms and Architectures for Elementary Function Computation*. Tese (Doutorado) — UNIVERSIDADE DE SANTIAGO DE COMPOSTELA, 2003.

[17] MEYER, U. *Digital Signal Processing with Field Programmable Gate Arrays*. Alemanha: Springer-Verlag, 2001.

[18] FRANTZ, G. *Comparing Fixed- and Floating-Point DSPs Does your design need a fixed- or floating-point DSP?* Disponível em *http://focus.ti.com/lit/wp/spry061/spry061.pdf*, 2004. Texas Instruments Incorporated.

[19] FAGIN, B.; RENARD, C. Field programmable gate arrays and floating point arithmetic. *Journal on VLSI Systems*, v. 2, n. 3, p. 365 – 367, Setembro 1994.

[20] LOUCA, L.; COOK, T.; JOHNSON, W. Implementation of IEEE single precision floating point addition and multiplication on FPGAs. In: IEEE. *Proceedings Symposium on FPGAs for Custom Computer Machines*. USA. p. 107 – 116.

[21] LIGON, W. et al. A re-evaluation of the practicality of floating-point operations on FPGAs. In: IEEE. *Proceedings Symposium on FPGAs for Custom Computer Machines*. USA, 1998. p. 206 – 215.

[22] SODERQUIST, P.; LEESER, M. Division and square root: choosing the right implementation. *Micro, IEEE*, USA, v. 17, n. 4, p. 56 – 66, julio/agosto 1997.

[23] BEAUCHAMP, M. et al. Embedded floating-point units in FPGAs. In: ACM. *Proceedings Symposium on Field Programmable Gate Arrays*. USA, 2006. p. 12–20.

[24] LEE, B.; BURGESS, N. Parameterizable floating-point operations on FPGA. In: IEEE. *Proceedings Conference. On Signals, Systems and Computers*. USA, 2002. p. 1064 – 1068.

[25] LIANG, J.; TESSIER, R.; MENCER, O. Floating point unit generation and evaluation for FPGAs. In: IEEE. *Proceedings Symposium on Field Programmable Custom Comp. Mach.* USA, 2003. p. 185–194.

[26] UNDERWOOD, K. FPGAs vs. CPUs: trends in peak floating-point performance. In: ACM. *Proceedings Symposium on Field Programmable Gate Arrays*. USA, 2004. p. 171 – 180.

[27] GOVINDU, G.; SCROFANO, R.; PRASANNA, V. A library of parameterizable floating-point cores for FPGAs and their application to scientific computing. In: IEEE. *Proceedings International Conference Eng of Reconfig. Syst. and Algorithms*. USA, 2005. p. 137–148.

[28] HERVEILLE, R. *Cordic Core Specification*. Disponível em *http://www.opencores.org/*, Dezembro 2001. OpenCores.

[29] LOGICORE, X. *Floating-Point Operator v3.0 Product Specification*. Disponível em *http://www.xilinx.com/support/documentation/ip_ documentation/floating_ point_ ds335.pdf*, Abril 2009. Xilinx Corporation.

[30] SHAO, X.; SUN, D.; MILLS, J. A new motion control hardware architecture with FPGA-based IC design for robotic manipulators. In: IEEE. *Proceedings on Robotics and Automation*. USA, 2006. p. 3520 – 3525.

[31] SHAO, X.; SUN, D.; MILLS, J. Development of an FPGA-based motion control ASIC for robotic manipulators. In: IEEE. *Proceedings on Intelligent Control and Automation*. China, 2006. p. 8221–8225.

[32] KUNG, Y.; SHU, G. Development of a FPGA-based motion control IC for robot arm. In: IEEE. *Proceedings on Industrial Technology*. China, 2005. p. 1397 – 1402.

[33] KUNG, Y.; SHU, G. Design and implementation of a control IC for vertical articulated robot arm using SOPC technology. In: IEEE. *Proceedings on Mechatronics*. Taiwan, 2005. p. 532–536.

[34] HUTCHINGS, B.; NELSON, B. Implementing applications with FPGAs. In: HAUCK, S.; DEHON, A. (Ed.). *Reconfigurable Computing*. USA: Elsevier Inc., 2008. p. 439–454.

[35] SANCHEZ, D. et al. Parameterizable floating-point library for arithmetic operations in FPGAs. In: ACM. *SBCCI 2009 22nd Symposium on Integrated Circuits and Systems Design*. Brasil, 2009. p. 253–258.

[36] ANSI/IEEE STD 754-1985. *IEEE standard for binary floating-point arithmetic*. USA. ANSI/IEEE.

[37] GOLDBERG, D. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, USA, v. 23, n. 1, p. 5 – 48, 1991.

[38] BELANOVIC, P. *Library of Parameterized Hardware Modules for Floating-Point Arithmetic With An Example Application*. Dissertação (Mestrado) — Northeastern University, 2002.

[39] SILVA, J. da. *JFloat: Uma Biblioteca de Ponto Flutuante para Linguagem Java com Suporte a Arredondamento Direcionado*. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Norte, 2007.

[40] ISO 8373:1994. *Manipulating industrial robots – Vocabulary*. USA, 1994. ISO.

[41] KOREM, Y. *Robotics for Engineers*. USA: McGraw-Hill, 1985.

[42] TATAS, K.; SIOZIOS, K.; SOUDRIS, D. A survey of existing fine-grain reconfigurable architectures and CAD tools. In: VASSILIADIS, S.; SOUDRIS, D. (Ed.). *Fine and Coarse-Grain Reconfigurable Computing*. Grécia: Springer, 2008. p. 3–88.

[43] KWON, T.; SONDEEN, J.; DRAPER, J. Floating-point division and square root implementation using a taylor-series expansion algorithm with reduced look-up tables. In: IEEE. *Proceedings Symposium Circuits and Systems*. USA, 2008. p. 954–957.

[44] LI, Y.; CHU, W. Implementation of single precision floating point square root on fpgas. In: IEEE. *Proceedings Symposium on Custom Computing Machines*. USA, 1997. p. 226–232.

[45] MONTUSCHI, P.; MEZZALAMA, M. Survey of square rooting algorithms. In: IEEE. *Proceedings Computers and Digital Techniques*. Reino Unido, 1990. p. 31 – 40.

[46] SAEZ, E. et al. FPGA implementation of a variable precision CORDIC processors. In: IEEE. *Proceedings Conference Design Circuits and Integrated Systems Computers*. USA, 1998. p. 253–258.

[47] VALLS, J.; KUHLMANN, M.; PARHI, K. Evaluation of CORDIC algorithms for FPGA design. *Journal of VLSI Signal Processing*, USA, v. 32, p. 207 – 222.

[48] LOGICORE. *COORDIC V 3.0*. Disponível em *http://www.xilinx.com/*, Abril 2005. Xilinx Corporation.

[49] ZHOU, J. et al. Dynamic configurable floating-point FFT pipelines and hybrid-mode CORDIC on FPGA. In: IEEE. *International Conference on Embedded Circuits and Systems*. China. p. 616–620.

[50] DETREY, J.; DINECHIN, F. Floating-point trigonometric functions for FPGAs. In: IEEE. *Proceedings Conference on Field Programmable Logic and Applications*. Holanda. p. 29–34.

[51] GOLDSCHMIDT, R. *Applications of Division by Convergence*. Dissertação (Mestrado) — M.I.T., 1964.

[52] KILTS, S. *Advanced FPGA Design*. New Jersey: John Wiley & Sons, Inc., 2007.

[53] MARKSTEIN, P. Software division and square root using goldschmidt's algorithms. In: UNIVERSITÄT TRIER. *Conference on Real Numbers and Computers*. Alemanha, 2004. p. 146 – 157.

[54] VOLDER, J. The CORDIC computing technique. In: *IRE-AIEE-ACM '59 (Western): computer conference*. New York, NY, USA: ACM, 1959. p. 257–261.

[55] MUNOZ, D. et al. Tradeoff of FPGA design of floating-point transcendental functions. In: IEEE. *Proceedings on Very Large Scale Integration*. Brasil, 2009. p. 1 – 4.

[56] HUANG, J. et al. DSP/FPGA-based controller architecture for flexible joint robot with enhanced impedance performance. *Journal of Intelligent and Robotic Systems*, v. 53, n. 3, p. 247–261, 2008.

[57] ZOU, J. *Fuzzy Sliding-Mode Control of a Two-Link Robot Manipulator*. Dissertação (Mestrado) — National Yunlin University of Science and Technology, 2009.

[58] GAJSKI, D. *Principles of Digital Design*. USA: Prentice Hall International Ltd., 1997.

[59] SANCHEZ, D. et al. FPGA implementation for direct kinematics of a spherical robot manipulator. In: IEEE. *2009 International Conference on ReConFigurable Computing and FPGAs*. México, 2009. p. 1–6.