DISSERTAÇÃO DE MESTRADO

IMPLEMENTAÇÃO DE UM FILTRO DE KALMAN ESTENDIDO EM ARQUITETURAS RECONFIGURÁVEIS APLICADO AO PROBLEMA DE LOCALIZAÇÃO EM ROBÓTICA MÓVEL

SÉRGIO MESSIAS CRUZ

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA MECÂNICA

IMPLEMENTAÇÃO DE UM FILTRO DE KALMAN ESTENDIDO EM ARQUITETURAS RECONFIGURÁVEIS APLICADO AO PROBLEMA DE LOCALIZAÇÃO EM ROBÓTICA MÓVEL

SÉRGIO MESSIAS CRUZ

Orientador: Prof. Dr. Carlos H. Llanos Quintero

DISSERTAÇÃO DE MESTRADO

Publicação: ENM.DM - 058 A/13

Brasília, 5 de Abril de 2013

UNIVERSIDADE DE BRASÍLIA

Faculdade de Tecnologia

DISSERTAÇÃO DE MESTRADO

IMPLEMENTAÇÃO DE UM FILTRO DE KALMAN ESTENDIDO EM ARQUITETURAS RECONFIGURÁVEIS APLICADO AO PROBLEMA DE LOCALIZAÇÃO EM ROBÓTICA MÓVEL

SÉRGIO MESSIAS CRUZ

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

Banca Examinadora

Prof. Dr. Carlos H. Llanos Quintero, _____
ENM/UnB
Presidente da Banca

Prof. Dra. Carla Cavalcante Koike, _____
CIC/UnB
Examinador Interno ao PPMEC

Prof. Dr. Geovany Araújo Borges, _____
ENE/UnB
Examinador Externo ao PPMEC

Prof. PhD Guilherme Carvalho Caribé, ENM/UnB _____
Membro Suplente

Brasília, 5 de Abril de 2013

FICHA CATALOGRÁFICA

CRUZ, SÉRGIO MESSIAS
Implementação de um Filtro de Kalman Estendido em Arquiteturas Reconfiguráveis aplicado ao Problema de Localização de Robôs Móveis [Distrito Federal] 2013.
xiv, 81p. 210 × 297 mm (ENM/FT/UnB, Mestre, Sistemas Mecatrônicos, 2013). Dissertação de Mestrado – Universidade de Brasília. Faculdade de Tecnologia.
Departamento de Engenharia Mecânica.

1. Localização
2. FPGAs
3. Robôs Móveis
4. Filtro de Kalman

I. ENM/FT/UnB
II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

CRUZ, SÉRGIO MESSIAS. (2013). IMPLEMENTAÇÃO DE UM FILTRO DE KALMAN ESTENDIDO EM ARQUITETURAS RECONFIGURÁVEIS APLICADO AO PROBLEMA DE LOCALIZAÇÃO DE ROBÔS MÓVEIS. Dissertação de Mestrado em Sistemas Mecatrônicos, Publicação ENM.DM - 058 A/13, Departamento de Engenharia Mecânica, Universidade de Brasília, Brasília, DF, 81p.

CESSÃO DE DIREITOS

AUTOR: Sérgio Messias Cruz.

TÍTULO: IMPLEMENTAÇÃO DE UM FILTRO DE KALMAN ESTENDIDO EM FPGA APLICADO À LOCALIZAÇÃO DE ROBÔS MÓVEIS.

GRAU: Mestre **ANO**: 2013

Sérgio Messias Cruz

Dedicatória

Dedico este trabalho a quatro mulheres de minha vida: primeiramente à Virgem Maria, mãe do meu Deus e Salvador, Jesus Cristo, e madrinha do meu mestrado; em seguida, a minha mãe terrena, Nilma, por suas orações incessantes a Deus por mim e minha família; terceiro, à minha amada esposa, Talita, por seu apoio e dedicação a mim; e por fim, à minha pequena princesa Maria Elis.

SÉRGIO MESSIAS CRUZ

Agradecimentos

Agradeço primeiramente a Deus e à sua estimadíssima mãe, a Bem-Aventurada Virgem Maria, por conceder-me a graça de ter cursado este mestrado em uma instituição de tão grande reputação. Pelas bençãos e graças derramadas sobre mim ao longo deste mestrado.

Agradeço em segundo lugar à minha tão amada e querida esposa Talita Cruz pelo seu companheirismo e apoio.

Agradeço em seguida a minha família, que sempre acreditou em mim e no meu potencial para vencer mais uma etapa na minha vida.

Ao meu orientador, o professor Carlos Humberto Llanos pela amizade, apoio, confiança, dedicação, paciência e orientação. Pelo tempo gasto nas longas reuniões e por ter me permitido trabalhar ao seu lado, aprendendo um pouco mais a cada dia.

Ao Professor Geovany A. Borges pelo suporte à execução deste trabalho, que praticamente abriu os horizontes deste projeto.

Aos meus amigos de laboratório: Jones, Daniel, Janier, Camilo, André, Milton, Guillermo e demais, pela ajuda e colaboração no desenvolvimento deste trabalho. Pela amizades e por me proporcionar momentos de alegrias.

À CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) pelo suporte financeiro a este trabalho.

Ao Grupo de Automação e Controle (GRACO/UnB) e a todos os meus professores pelo suporte e formação acadêmica.

À Altera Inc. e DHW Engenharia pela doação de kits de desenvolvimento e suporte técnico.

SÉRGIO MESSIAS CRUZ

RESUMO

Este trabalho descreve uma arquitetura de *hardware* para a implementação de uma versão sequencial do Filtro de Kalman Estendido (EKF, do inglês *Extended Kalman Filter*). Devido ao fato de que o EKF é computacionalmente intensivo, comumente ele é implementado em plataformas baseadas em PC (do inglês *Personal Computer*) para ser empregado em robótica móvel. Para permitir o desenvolvimento de plataformas robóticas pequenas (por exemplo, aquelas requisitadas em robótica móvel) condições específicas tais como tamanho pequeno, consumo baixo de potência e capacidade de aritmética em ponto flutuante são exigidos, assim como projetos de arquiteturas de *hardware* específicas e adequadas. Desta maneira, a arquitetura proposta foi projetada para tarefas de auto-localização, usando operadores de aritmética de ponto flutuante (em precisão simples), permitindo a fusão de dados provenientes de diferentes sensores tais como ultrassom e ladar. O sistema foi adaptado para ser aplicado em uma plataforma reconfigurável, apropriada para tarefas de pesquisa, e a mesma foi testada em uma plataforma robótica Pioneer 3AT (da Mobile Robots Inc.) a fim de avaliar sua funcionalidade, usando seu sistema de sensoriamento. Para comparar o desempenho do sistema, o mesmo foi implementado em um PC, assim como pela utilização de um microprocessador embarcado na FPGA (o Nios II, da Altera). Neste trabalho, várias métricas foram utilizadas a fim de avaliar o desempenho e a aplicabilidade do sistema, medindo o consumo de recursos na FPGA e seu desempenho. Devido ao fato de que este trabalho só está implementando a fase de atualização do EKF, o sistema geral foi testado assumindo que o robô está parado em uma posição previamente conhecida.

ABSTRACT

This work describes a hardware architecture for implementing a sequential approach of the Extended Kalman Filter (EKF) that is suitable for mobile robotics tasks, such as self-localization, mapping, and navigation problems, especially when FPGAs (*Field Programmable Gate Arrays*) are used to execute this algorithm. Given that EKF is computationally intensive, commonly it is implemented in PC-based platforms to be employed on mobile robots. In order to allow the development of small robotic platforms (for instance those required in microrobotics area) specific requirements such as small size, low-power, and floating-point arithmetic capability are demanded, as well as the design of specific and suitable hardware architectures. Therefore, the proposed architecture has been achieved for self-localization task, using floating-point arithmetic operators (in simple precision), allowing the fusion of data coming from different sensors such as ultrasonic and laser rangefinder. The system has been adapted for achieving a reconfigurable platform, suitable for research tasks, and the same has been tested in a Pioneer 3AT mobile robot platform (from Mobile Robots Inc.) for evaluating its functionality by using its local sensing system. In order to compare the performance of the system, the same localization technique has been implemented in a PC, as well as using an FPGA-embedded microprocessor (the Nios II from Altera Inc.) In this work several metrics have been used in order to evaluate the system performance and suitability, measuring both the FPGA resources consumption and performance. Given that in this work only the update phase of the EKF has been implemented the overall system has been tested assuming that the robot is stopped in a previously well-known position.

RESUMEN

Este trabajo describe una arquitectura de hardware para la implementación de una versión secuencial del filtro de Kalman extendido (EKF del ingles *Extended Kalman Filter*). Debido al hecho de que el EKF es computacionalmente intensivo, típicamente es implementado en plataformas basadas en PC's (del ingles *Personal Computer*) para ser utilizado en robótica móvil. Para permitir el desarrollo de pequeñas plataformas robóticas(como las requeridas en robótica móvil) son exigidos condiciones especificas como su pequeño tamaño, bajo consumo de potencia y capacidad de aritmética en punto flotante, así como arquitecturas de hardware especificas y adecuadas. De esta manera la arquitectura propuesta fue proyectada para tareas de auto-localización, usando operadores de aritmética de punto flotante (en precisión simple), permitiendo la fusión de datos provenientes de diferentes sensores tales como ultrasonido y ladar. El sistema fue adaptado para aplicarlo en una plataforma reconfigurable, apropiada para investigación, y la misma fue probada en una plataforma robótica denominada Pioneer 3AT (de la compañía Mobile Robots Inc.) utilizando el sistema de sensoramiento de este, con el propósito de validar su funcionalidad. Para comparar el desempeño del sistema, este fue implementado en un PC, así como en un microprocesador embarcado en una FPGA (Nios II, de Altera). En este trabajo, varias métricas fueron utilizadas con el propósito de validar el desempeño y la aplicabilidad del sistema, midiendo el consumo de recursos en la FPGA y su desempeño. Debido al hecho de que en el trabajo solo esta implementado la fase de actualizacion del EKF el sistema general fue probado asumiendo que el robot esta parado en una posición previamente conocida.

Sumário

Lista de Figuras

Lista de Tabelas

LISTA DE SÍMBOLOS

ASIC	Application Specific Integrated Circuits
BME	Banco das Matrizes de Entrada
BMI	Banco das Matrizes Intermediárias
BMS	Banco das Matrizes de Saída
CORDIC	COordinate Rotation DIgital Computer
CLB	Configurable Logic Block
CPU	Central Processing Unit
DGPS	Differential Global Positioning System
DMA	Direct Memory Access
DSP	Digital Signal Processor
EKF	Extended Kalman Filter
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GPP	General Purpose Processor
GPS	Global Positioning System
HW/SW	Hardware/Software
IOB	Input Output Block
KF	Kalman Filter
LEs	Logic Elements
LPM	Library Parameterized Modules
LRF	Laser RangeFinder
MCL	Monte Carlo Localization
MSB	Most Significant Bit
PC	Personal Computer
PE	Processing Element
ROM	Read Only Memory

SCG	Sistema de Coordenadas Global
SCR	Sistema de Coordenadas do Robô
SCSi	Sistema de Coordenadas do Sensor Si
SLAM	Simultaneous Localization And Mapping
SMG-SLAM	Scan-Matching Genetic SLAM
VHDL	VHSIC Hardware Description Language
UKF	Unscented Kalman Filter
UPF	Unidade em Ponto Flutuante

1 Introdução

1.1 Contextualização

Nos últimos anos, o uso de FPGA (do inglês ***F**ield **P**rogrammable **G**ate **A**rray*) vem obtendo uma atenção especial da comunidade científica para soluções de problemas computacionais envolvendo cálculos algébricos intensos, aplicados à supercomputação, processamento de imagens, visão computacional, assim como na área da mecatrônica [2]. Uma prova disso é que em uma simples busca pelas palavras *FPGA* e *algorithms* no *site* do Google (realizada em setembro de 2011) quase sete milhões de resultados foram encontrados. Nesse contexto, os FPGAs têm sido usados para a realização de soluções eficientes e adequadas em diversas áreas, a maioria delas relatadas ao desenvolvimento de *sistemas embarcados*.

Sistemas computacionais embarcados representam um campo amplo de desenvolvimento, responsável por um mercado de trabalho crescente [5]. Esses tipos de sistemas podem ser entendidos como sistemas computacionais especializados, desenvolvidos para realizar uma tarefa específica. Comumente, são projetados em unidades de processamento com capacidade limitada (geralmente microcontroladores) e um conjunto de sensores que lhes permitem coletar dados, viabilizando a tomada de decisões e a realização de ações de forma automática [6].

Pode-se afirmar ainda que os sistemas embarcados são componentes de *hardware* e *software* integrados em uma solução mais adequada e abrangente, em ordem de preencher várias funcionalidades de um produto específico, e os mesmos não são encontrados somente em carros, plataformas para robôs e aeronaves, mas também em plantas industriais, tecnologia da automação, aparelhos médicos e tecnologia ambiental e de energia[7]. Neste contexto, o projeto de sistemas embarcados leva naturalmente à aplicação de técnicas de co-projeto de *hardware/software* (*hardware/software codesign*). Tendo em conta o anterior, é necessário verificar que partes do sistema a ser projetado são críticas no desempenho do mesmo. Neste caso, um estudo adequado pode determinar a necessidade de implementar certas partes críticas do sistema em módulos de *hardware* especializados, os quais podem ser implementados em ASICs (do inglês *Application Specific Integrated Circuits*) ou em plataformas reconfiguráveis, usando FPGAs, entre outras possíveis soluções [8].

Uma das principais diferenças entre sistemas embarcados e sistemas computacionais de propósitos gerais, como por exemplo, os computadores pessoais (PCs), é que os sistemas embarcados comumente requerem *hardware* e *software* de propósito específico [5, 6]. Atualmente existe uma grande demanda de sistemas embarcados que permitam não apenas realizar tarefas computacionais de alto desempenho, senão, que também cumpram requerimentos de robustez, bom desempenho, tolerância a falhas, baixo custo e baixo consumo de energia [9].

É importante destacar que os sistemas embarcados comumente são projetados para um propósito definido e, portanto, pode-se usar em benefício a informação disponível sobre os requisitos que o sistema deve cumprir, assim como as restrições às quais está sujeito [6]. Em geral todos os sistemas embarcados são desenhados para operar sobre restrições de portabilidade (tamanho e peso), consumo de recursos, baixa frequência do *clock*, desempenho adequado, baixo consumo de energia e dissipação de potência [10]. Esta última restrição penaliza implementações com alta frequência de operação, portanto novas soluções devem ser analisadas visando cumprir os requerimentos do produto desejado. Uma resposta a este quesito é o uso de soluções de *hardware* que explorem o paralelismo intrínseco dos algoritmos a serem embarcados, permitindo assim gerar implementações com bom desempenho, mesmo operando com baixas frequências de *clock*.

O uso de FPGAs em sistemas embarcados é explicado devido ao fato de que estes últimos estão sujeito à várias restrições em projeto, tais como desempenho, área, custo, flexibilidade e consumo de potência. Nesse contexto, várias aplicações de FPGAs (como parte dos módulos embarcados) direcionadas à área da robótica surgiram nos últimos anos. Esses desenvolvimentos foram focados para acelerar a execução de algoritmos, usando o FPGA como um acelerador de *hardware*, em parte ou no todo, de alguns algoritmos específicos aplicados à robótica móvel, como por exemplo, cadeias de Markov, filtros de Kalman, métodos de Monte Carlo, algoritmo de SLAM (do inglês ***S**imultaneous **L**ocalization **A**nd **M**apping*), entre outros. Esses algoritmos probabilísticos têm sido historicamente usados em robótica móvel, a fim de tornar as tarefas de mapeamento, localização e navegação mais robusto em relação às incertezas e/ou ruído gerados pelos sensores e também às incertezas intrínsecas do ambiente [11].

Robôs móveis são fornecidos com um grande número de tipos de sensores e plataformas de sensoriamento, proporcionando arquiteturas com informações complementares ou às vezes de aspectos redundantes. Em vários casos, os robôs móveis transportam sensores para cálculo de posição, como *encoders* e geomagnéticos, ou para a construção de mapas e auto-localização, tais como ultrassons, infravermelhos e os baseados em laser. Neste caso, a utilização de técnicas para administrar uma grande quantidade de informações provenientes de vários sensores, cada um com parâmetros diferentes de exatidão e precisão, são essenciais. Estas técnicas, que visam a estimação de uma grandeza física (o mensurando) e levam em conta diversas medições, são conhecidas como *Fusão Sensorial*.

O algoritmo de fusão sensorial implementado neste trabalho foi abordado para a plataforma Pioneer 3-AT (P3-AT) [12], que é um sistema robótico com tração nas quatro rodas equipado com comunicação *Ethernet*, sistema de ladar, DGPS (do inglês ***D**ifferential **G**lobal **P**ositioning **S**ystem*), 8 sensores de ultrassom na frente e mais 8 traseiros com detecção de obstáculos de 15 cm a 7 m. O P3-AT utiliza *encoders* com correção inercial recomendado para cálculo de posição para compensar derrapagens. Seu sensoriamento estende-se com opções à base de laser (ladar), correção integrada para compensar a derrapagem, GPS (do inglês ***G**lobal **P**ositioning **S**ystem*), para-choques, garra, visão e bússola.

Na área de robótica móvel, a fusão sensorial é amplamente utilizada, principalmente focada para resolver problemas de localização, navegação e mapeamento. A fusão de sensores é o processo de integração de dados provenientes de diferentes sensores, mesmo usando diferentes princípios físicos, para a detecção de objetos, ou a estimação de parâmetros, ou a definição de estados, que são necessários para a auto-localização, cartografia, *path computing*, planejamento e controle de trajetória, bem como sua execução. Métodos de fusão sensorial, na área de robótica, são necessários a fim de traduzir as diferentes entradas sensoriais em estimativas confiáveis, visando a obtenção de modelos de ambiente confiáveis para as tarefas de navegação.

1.2 Definição do problema e motivações

A fusão de sensores precisa lidar com o comportamento estatístico de cada sensor, que pode ser conhecido ou não. Se o comportamento estatístico for conhecido, a tarefa de fusão depende de técnicas bem desenvolvidas como a estimativa *a posteriori* e *a máxima verossimilhança*, adaptando os resultados da filtragem de Kalman, assim como outras teorias Bayesianas. Neste caso, a maioria dos usos do filtro de Kalman em fusão sensorial (por exemplo, para a navegação) são dirigidas para a construção e manutenção de um modelo de ambiente para robôs móveis e a monitorização da posição desse robô no ambiente. Para isso, as equações do filtro de Kalman são implementadas em *software* sobre uma plataforma de sistema embarcado baseado em microprocessador. A fusão de sensores também pode ser alcançada usando arquiteturas descentralizadas para aumentar a capacidade de cálculo e comunicação, assim como para melhorar a performance. Neste caso, a estimativa local dos parâmetros a partir dos dados disponíveis é feita seguida da fusão global das estimativas locais. As estimativas locais podem ser executadas por *hardwares* específicos em que as equações do filtro de Kalman foram distribuídas. A maioria das aplicações do filtro de Kalman utilizam a versão não-linear (EKF), que também inclui o cálculo Jacobiano sobre o processo [13].

Na aplicação do EKF utilizando FPGAs (focado na fusão sensorial para resolver o problema de localização) trabalhos anteriores têm lidado com o armazenamento de grandes volumes de dados

provenientes de diferentes sensores na memória. Este fato representa um problema grave, uma vez que os tamanhos das matrizes aumentam em função do número de sensores. Neste contexto, sabe-se bem que a complexidade computacional para o algoritmo EKF no SLAM é $O(n^2)$, onde n representa o número de características (*features*) [14]. Adicionalmente, cada característica precisa ser detectada por um sistema de medição, geralmente baseado em vários sensores. O armazenamento prévio de dados dos sensores para a etapa da estimação (ou predição) leva as operações com matrizes (por exemplo, adição/subtração, multiplicação, transposta e inversa) a dimensões maiores. Esse fato, por sua vez, requer a necessidade de projetos de arquiteturas de *hardware* que levem em consideração as necessidades de largura de banda de memória para os EKFs baseados em FPGAs [15, 16], e isto pode representar um caso específico do problema de *memory-wall* [5].

O problema das dimensões elevadas das matrizes pode ser superado usando uma derivação sequencial do algoritmo EKF, em que a operação 'inversa da matriz' pode ser reduzida (mesmo a uma inversão escalar), sendo as observações dos sensores processadas uma por vez. A redução da dimensão da matriz também tem impacto no desempenho das outras operações, como adição/subtração, multiplicação e transposta. Este fato é interessante para implementações em FPGA, dadas os seguintes aspectos: (a) FPGAs têm recursos de *hardwares* limitados, que podem ser essenciais para operações em ponto flutuante, (b) requisitos de entrada e saída dos FPGAs podem ser drasticamente reduzidos utilizando a abordagem sequencial EKF, (c) a abordagem sequencial do EKF permite ao projetista usar dispositivos FPGA menores e mais baratos, (d) o uso do paralelismo intrínseco do EKF na FPGA melhora o desempenho de ambos o algoritmo EKF e a aplicação de localização global, (e) o potencial do paralelismo dos FPGAs pode equilibrar o processamento dos dados em série, e (f) a utilização de pequenos dispositivos com um desempenho apropriado permite aplicar as soluções para a área da microrobótica.

1.3 Objetivos

1.3.1 Objetivo Geral

Como objetivo geral, esta dissertação apresenta o desenvolvimento de uma arquitetura em *hardware* (de baixo custo em área e alta performance) do algoritmo de atualização do Filtro de Kalman Estendido (EKF sequencial) para o problema de localização de robôs, usando multi sensores, tais como ultrassom e ladar.

Para diminuir a complexidade da implementação, considera-se que o mapa do ambiente, assim como a posição do robô é conhecida. O robô também está parado no ambiente e o processamento de dados é *offline*.

1.3.2 Objetivos Específicos

São objetivos específicos deste trabalho os seguintes:

(a) Obtenção do modelo de medição tendo em conta a disposição dos sensores no robô.

(b) Adaptação de arquiteturas de operadores em ponto flutuante (desenvolvidas no GRACO-UnB) para este projeto.

(c) Implementação das equações que formam parte do algoritmo EKF em *software* para efeito de validação do modelo.

(d) Implementação em *hardware* da etapa de correção do algoritmo EKF para o problema de localização em robótica móvel.

(e) Implementação em *hardware* de versão sequencial do algoritmo EKF visando a redução do consumo de recursos no FPGA.

(f) Desenvolvimento das arquiteturas em *hardware*, usando diferentes estratégias, para comparação de resultados de desempenho e consumo de recursos computacionais.

(g) Validação das arquiteturas, comparando resultados em *software* com os resultados em *hardware*.

(h) Criação de um ambiente de validação na plataforma Pioneer 3-AT.

1.4 Resultados Alcançados e Contribuições deste Trabalho

Neste contexto, as novas contribuições deste trabalho são:

(a) Implementação da versão sequencial do algoritmo EKF em FPGA, usando representação em ponto flutuante e;

(b) Validação dos resultados em termos de desempenho de *hardware* e sua funcionalidade em um cenário real, embarcada em uma plataforma robótica.

(c) Apresentação em poster no Sforum 2011 (Chip on the Cliffs); apresentação em poster no LASCAS 2013 (IEEE); apresentação oral no LASCAS 2013 (IEEE) e poster no RAW 2013 -IEEE (a apresentar em Maio/13).

1.5 Organização do Trabalho

O trabalho está organizado da seguinte maneira:

O capítulo 2 apresenta o problema da localização de robôs móveis, bem como os algoritmos usados para a resolução desta tarefa. O capítulo termina apresentando um resumo da revisão bibliográfica sobre as diferentes arquiteturas pesquisadas.

Em seguida, o capítulo 3 aborda o algoritmo EKF, usado em localização de robôs móveis, e as equações obtidas para a versão sequencial da mesma.

O capítulo 4 inicia com um pequeno resumo sobre a tecnologia FPGA, bem como suas vantagens e termina apresentando as estruturas das unidades em ponto flutuante (UPFs) usadas neste trabalho.

Logo após, o capítulo 5 descreve as arquiteturas desenvolvidas para o problema de localização de robôs. São apresentadas duas abordagens para a solução do problema, bem como a estrutura das arquiteturas.

No capítulo 6, apresenta-se os resultados dessas aplicações e, por último, no capítulo 7 conclui-se o trabalho com projeções de trabalhos futuros.

2 Localização de Robôs Móveis

2.1 Algoritmos Aplicados em Localização de Robôs Móveis

Como se sabe, a localização é a auto-determinação da posição de um robô móvel em um ambiente. Como descreve [1], a localização é um dos quatro blocos (ver Fig. 2.1) responsáveis pelo sucesso da navegação de um robô: (a) *percepção*, o robô deve interpretar as informações provenientes de seus sensores para extrair dados significativos; (b) *cognição*, o robô deve decidir como agir para alcançar seus objetivos; (c) *controle de trajetória*, o robô deve modular a velocidade de seus motores para percorrer a trajetória delineada; e (d) a *localização*.

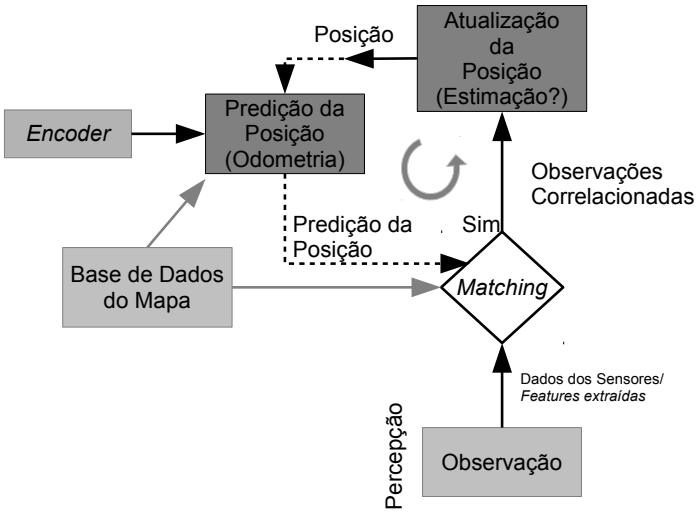


Figura 2.1. Esquema geral para localização de robôs móveis, adaptado de [1]

Desses quatro blocos, a localização tem recebido uma atenção especial da comunidade científica nas últimas décadas. Como resultado disto, avanços significantes foram obtidos nesta frente,

o que é de grande valia, pois além de ser essencial para a navegação, a localização também é importante para a construção de mapas.

A localização de robôs móveis é frequentemente chamada de *position estimation* ou *position tracking* e pode ser vista como um problema de transformação de coordenadas [14]. Neste sentido, os mapas são descritos em um sistema de coordenada global, que é independente da *pose* (ou postura) do robô. A localização passa a ser o procedimento de correlacionar o sistema de coordenada do mapa global com o sistema de coordenada local do robô. O conhecimento dessa transformação de coordenada permite ao robô expressar a localização dos objetos de interesse com sua própria coordenada. Logo, a postura do robô pode ser facilmente expressa por $x_t = \{x\, y\, \theta\}^T$ [14] (Ver Figura 2.2).

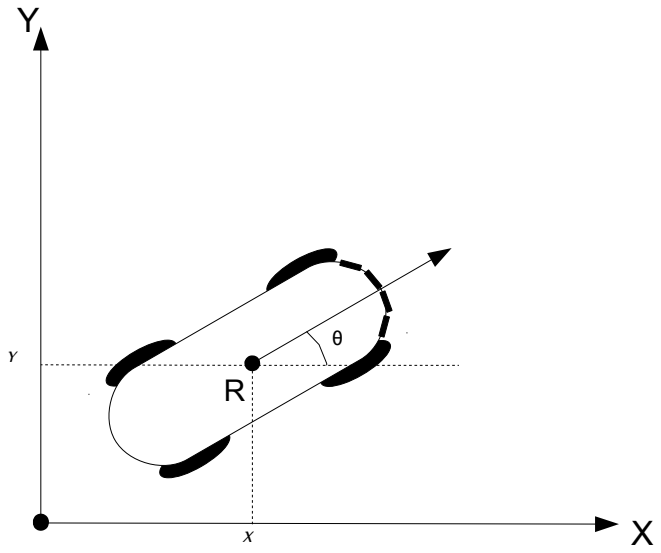


Figura 2.2. Representação da postura de um robô em sistema de coordenadas de 2 dimensões

Neste capítulo, serão apresentados brevemente quatro dos algoritmos usados para o problema de localização de robôs móveis, dentre eles *Localização por GRID*, *Localização de Monte Carlo*, *Localização por Markov* e *Algoritmo EKF*.

2.1.1 Algoritmo *GRID*

A localização por *GRID* (ou *grade* em português) usa um filtro de histograma para representar a crença (ou confiança) *posteriori*, ou crença posterior. Este algoritmo mantêm como crença posterior uma coleção de valores de probabilidades discretas (vide equação 2.1).

$$bel(x_t) = p_{k,t} \tag{2.1}$$

onde cada probabilidade $p_{k,t}$ é definida sobre uma célula-grade x_k.

O conjunto de todas as células-grade forma uma partição do espaço de todas as posturas, como mostrado na equação 2.2:

$$range(X_t) = x_{1,t}\, U\, x_{2,t}\, U\, ...\, x_{k,t} \tag{2.2}$$

Na versão mais básica do algoritmo, a partição do espaço é invariante no tempo e cada célula-grade é do mesmo tamanho. Uma granularidade comum usada em vários ambiente internos é de 15 cm para dimensões x e y (como visto na Figura 2.3), e 5 graus para a dimensão rotacional. Uma representação mais fina obtém melhores resultados, porém aumenta os requisitos computacionais [14].

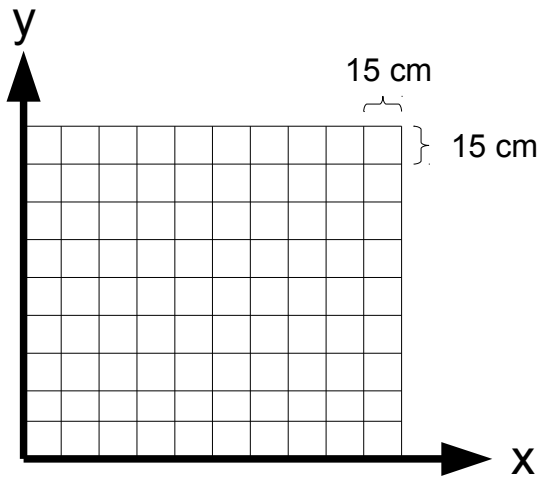


Figura 2.3. Esquema de uma representação por *grid*

Algoritmo 1 Localização por *GRID*

1: **for all** k **do**
2: $\quad \bar{p}_{k,t} = \sum_i p_{i,t-1} f(m(x_k), u_t, m(x_i))$
3: $\quad p_{k,t} = \eta\, g(z_t, m(x_k), m)$
4: **end for**
5: **return** $p_{k,t}$

A localização por *Grid* é em grande parte idêntica ao filtro Bayesiano binário básico, a partir do qual é derivada. O Algoritmo (1) fornece um pseudo-código para a implementação mais básica. Ela exige como entrada os valores de probabilidade discreta $p_{t-1,k}$, juntamente com a medição mais recente, o controle e o mapa. Sua malha interna percorre todas as células-grade. A linha 2 implementa a atualização do modelo de movimento e a linha 3 a atualização de medição.

As probabilidades finais são normalizadas, tal como indicado pelo normalizador η na linha 3. As funções $f()$ e $g()$ representam o modelo do movimento e modelo de medição respectivamente.

O algoritmo (1) assume que cada célula possui a mesma área.

Abaixo segue um resumo da técnica de localização por *GRID* descrita por [14]:

(a) A técnica por *GRID* representa a crença posterior por histogramas;

(b) Pode-se fazer representações do ambiente em duas ou três dimensões;

(c) O tamanho da grade influencia diretamente na precisão e eficiência computacional. Células pequenas implicam em menores erros de estimação porém em alto custo computacional e um maior tempo para localizar-se, ou seja, dificuldade na execução do algoritmo em tempo real;

(d) A localização por *GRID* pode globalmente localizar o robô.

2.1.2 Algoritmo de *Monte Carlo*

O método de Monte Carlo (MCL, do inglês ***M**onte **C**arlo **L**ocalization*), também conhecido como Filtro de Partículas, baseia-se na distribuição aleatória de possíveis estados em um espaço de configurações, ou seja, espaços livres para navegação em um mapa [17].

Cada estado é representado por uma partícula e pode ser usado um número grande de partículas. Este fator interfere na precisão e velocidade de convergência, assim como no calculo computacional. A quantidade de partículas deve ser suficiente para uma cobertura adequada do mapa. A partir disso, cada tomada de controle do robô (movimento pelo ambiente) é aplicada às partículas a partir do modelo estatístico de movimentação (similar ao ocorrido no Filtro de Kalman). Além do mais, cada observação real do estado dos sensores é comparada à observação das partículas (situações hipotéticas).

Dependendo da proximidade entre a leitura do sensor e a leitura estimada para cada partícula, essa partícula recebe um peso, indicando sua probabilidade (também conhecida como *likelihood*) de ser representativa da posição real do robô. A partir disso, realiza-se uma nova seleção aleatória de amostras das possíveis configurações, desta vez levado-se em conta os pesos das partículas. Quanto maior a *likelihood* de uma partícula, maior a probabilidade de que ela seja escolhida novamente. Isto, com tempo resulta na aproximação da real posição do robô, representada pela partícula com a melhor estimativa da posição. Uma vantagem do sistema é que se o robô for removido de sua posição, ele pode se localizar de naturalmente, já que as partículas se espalharão pelo mapa em seleções (ou *samplings*) posteriores [18].

Para melhor exemplificar o método de Monte Carlo, uma descrição básica do do algoritmo pode ser vista no Algoritmo (2). X_t é a mostragem posterior da distribuição (partículas) e pode

Algoritmo 2 Localização por *Monte Carlo*

1: $\bar{X}_t = X_t = \oslash$
2: **for** $m = 1 \rightarrow M$ **do**
3: $\quad x_t^{[m]} = f(u_t, x_{t-1}^{[m]})$
4: $\quad w_t^{[m]} = g(z_t, x_{t-1}^{[m]}, m)$
5: $\quad \bar{X}_t = X_t + \left\langle x_t^{[m]}, w_t^{[m]} \right\rangle$
6: **end for**
7: **for** $m = 1 \rightarrow M$ **do**
8: $\quad$ *desenhe i com probabilidade* $\propto w_t^{[i]}$
9: $\quad$ *adicione* $x_t^{[i]} \rightarrow X_t$
10: **end for**
11: **return** X_t

ser denotada pela equação 2.3.

$$X_t = x_t^{[1]}, x_t^{[2]}, \ldots, x_t^{[M]} \tag{2.3}$$

Cada partícula $x_t^{[M]}$ (com $1 \leq m \leq M$) é uma estância concreta do estado no tempo t, onde M é o número de partículas na partícula X_t. Na Linha 4 contém amostras do modelo de movimento $f()$, usando partículas de crença atual como pontos de partida. O modelo de medição $g()$ é então aplicada na linha 5 para determinar o peso w da referida partícula de importância. A crença inicial $bel(x0)$ é obtida gerando aleatoriamente M tais partículas antes da distribuição $p(x0)$ e atribuindo o fator de importância uniforme M^{-1} para cada partícula. Como na localização *Grid*, as funções $f()$ e $g()$ representam o modelo do movimento e modelo de medição respectivamente [14].

A seguir apresentamos um resumo do algoritmo MCL conforme [14]:

(a) O algoritmo MCL representa a crença posterior usando partículas;

(b) A precisão computacional está ligada diretamente ao número de partículas;

(c) Localiza globalmente o robô;

(d) Pela adição de partículas aleatórias (*random particles*), o MCL resolve problemas de sequestro de robô (*kidnapped robot problem*);

(e) Representação complexa de distribuições de probabilidade multi-modal.

2.1.3 Algoritmo de *Markov*

O algoritmo de Markov é uma variante dos filtros de *Bayes*, cuja base é a teoria das probabilidades. A aplicação direta desses filtros para o problema de localização é chamada de *Localização por Markov* (ou *Markov Localization*) [14].

O algoritmo de localização por Markov é derivado do algoritmo do filtro de Bayes, ou seja, ele transforma uma *crença* probabilística no tempo $t-1$ em uma crença no tempo t. A localização por Markov aborda os problemas da localização global, do rastreamento da posição e da mudança de posição arbitrária do robô em um ambiente estático.

Algoritmo 3 Localização por Markov

1: **for all** x_t **do**
2: $\quad \bar{bel}(x_t) = \int p(x_t \| u_t, x_{t-1}, m) bel(x_{t-1}) dx$
3: $\quad bel(x_t) = \eta\, p(z_t \| x_t, m) \bar{bel}(x_t)$
4: **end for**
5: **return** $bel(x_t)$

O pseudocódigo visto em Algoritmo (3) descreve a sequencia básica da localização por Markov. Como pode ser observado, é um algoritmo relativamente simples, que processa basicamente duas etapas e baseia-se em uma crença $bel()$.

Na primeira etapa (linha 2), o algoritmo processa o controle u_t. Ele faz isso calculando a crença do estado x_t baseado na crença *a priori* do estado x_{t-1}, no controle u_t e na presença de um mapa m. Ou seja, é calculada uma integral (somatória) de um produto de duas distribuições: a distribuição *a priori* assinalada a x_{t-1} e a probabilidade que controla u_t [14]. Esta etapa é chamada de *atualização do controle* ou *predição*.

A segunda etapa (linha 3) é chamada de *atualização da medição*. É uma multiplicação da crença $\bar{bel}(x_t)$ pela probabilidade da medição z_t observada. Isto é feito para cada estado hipotético x_t. Porém, nem sempre o produto resultante será uma probabilidade, ou seja, ele não pode integrar a 1. Assim, o resultado é normalizado, em virtude de uma constante de normalização η.

É importante lembrar que para computar a crença posterior, o algoritmo requer uma crença inicial $bel(x_0)$ no tempo $t = 0$ como condição inicial de contorno.

A seguir apresentamos um resumo (de acordo com [14]) da localização por Markov:

(a) A localização por Markov é somente um nome diferente para o filtro de *Bayes* aplicado ao problema de localização de robô;

(b) É uma técnica baseada em processos estatísticos;

(c) A localização por Markov aborda o problema de localização global, do rastreamento de posição e o problema do robô sequestrado em ambientes *estáticos*.

2.1.4 Algoritmo *EKF*

Em estatística, o filtro de Kalman (KF, do inglês ***K**alman **F**ilter*) é um método matemático criado por Rudolf Kalman [19]. Seu propósito é utilizar medições de grandezas realizadas ao longo do tempo (contaminadas com ruído e outras incertezas), gerando resultados que tendam a se aproximar dos valores reais das grandezas medidas. O filtro de Kalman apresenta diversas aplicações, dentre elas a localização de robôs móveis, sendo parte essencial do desenvolvimento de tecnologias espaciais e militares [14].

O filtro de Kalman produz estimativas dos valores reais de grandezas medidas e valores associados predizendo um valor, estimando a incerteza do valor predito e calculando uma média ponderada entre o valor predito e o valor medido. O peso maior é dado ao valor de menor incerteza. As estimativas geradas pelo método tendem a estar mais próximas dos valores reais que as medidas originais pois a média ponderada apresenta uma melhor estimativa de incerteza que ambos os valores utilizados no seu cálculo.

Do ponto de vista teórico, o filtro de Kalman é um algoritmo para realizar, de forma eficiente, inferências exatas sobre um sistema dinâmico linear, sendo um modelo Bayesiano semelhante ao modelo de Markov, mas onde o espaço de estados das variáveis não observadas é contínuo e todas as variáveis, observadas e não observadas, apresentam uma distribuição normal [14].

Algoritmo 4 Localização EKF

1: $\bar{\mu}_t = g(u_t, \mu_{t-1})$
2: $\bar{\sum}_t = G_t \sum_{t-1} G_t^T + R_t$
3: $K_t = \bar{\sum}_t H_t^T (H_t \bar{\sum}_t H_t^T + Q_t)^{-1}$
4: $\mu_t = \bar{\mu}_t + K_t(z_t - h(\bar{\mu}_t))$
5: $\sum_t = (I - K_t H_t)\bar{\sum}_t$
6: **return** $\mu_t, \sum_t$

Como mencionado no parágrafo anterior, o KF é usado para sistemas dinâmicos lineares enquanto que o EKF é aplicado para sistemas dinâmicos não lineares. O algoritmo do EKF possui basicamente duas etapas: (a) predição e (b) correção (ou atualização), e pode ser conferida no Algoritmo (4). Linhas 1 e 2 correspondem à predição e linhas 3 a 5 à correção.

O filtro de Kalman estendido utiliza um modelo dinâmico não linear de um sistema $g()$ (as equações do movimento, por exemplo), entradas de controle u_t conhecidas, assim como medições z_t (como as de sensores) para gerar uma estimativa das grandezas variáveis do sistema (seus estados) μ_t e um incerteza associada $\sum_t$. A estimativa obtida desta forma é melhor que a estimativa obtida utilizando-se qualquer uma das medidas unicamente. Assim, é um algoritmo apropriado para se implementar uma *fusão de sensores*.

Abaixo segue um resumo da técnica de localização EKF descrita por [14]:

(a) É uma técnica baseada em processos estatísticos;

(b) A Localização EKF é um caso especial da localização por Markov;

(c) É bem implementada para o problema de rastreamento de posição de robô como incerteza limitada e em ambientes com *features* distintas;

(d) Não é aconselhada para localização global ou ambientes onde a maioria dos objetos não possuem semelhança;

(e) Aplica-se para sistemas cujo comportamento é não linear;

(f) Leva em consideração valores de diferentes sensores, ideal para fusão sensorial.

2.2 Trabalhos Correlatos de implementações do EKF em FPGAs

Para a tarefa de localização alguns trabalhos têm sido desenvolvidos usando arquiteturas FPGAs aplicados à robótica móvel. Na área de localização de robôs, [15] apresenta uma arquitetura baseada em FPGA para o algoritmo EKF que é capaz de processar mapas de 2 dimensões (2D) contendo até 1.8k características (*features*) em tempo real (14Hz), em que o sistema simultaneamente estima um modelo do ambiente (mapa) e a posição do robô com base em ambas informações do sensores odométricos e exteroceptivo. Os mesmos autores relatam que sua aplicação é duas ordens de magnitude mais eficiente do que um processador de propósito geral (GPP, do inglês ***G**eneral **P**urpose **P**rocessor*).

Na mesma direção, [11] os mesmos autores apresentam uma arquitetura implementada em *hardware* para resolver o problema de SLAM, cuja implementação foi completamente embarcada em uma FPGA Stratix II da Altera. A arquitetura proposta possui quatro bancos de memória para armazenamento de dados, um certo número de memória embarcada no *chip*, uma máquina de estados e quatro Elementos de Processamento (PE, do inglês ***P**rocessing **E**lements*). Os autores discutem que o número de operações em ponto flutuante para a implementação do algoritmo EKF é proporcional a n^2, sendo n o número de *features*. Eles reportam que a performance de sua arquitetura é no mínimo uma (1) ordem de magnitude melhor do que uma implementação em um PC *top* de linha.

Em contraste com os trabalhos anteriores, [20] apresenta um algoritmo variante do SLAM chamado SMG-SLAM (do inglês *Scan-Matching Genetic SLAM*). Para estimar a nova pose do robô, o algoritmo SMG-SLAM verifica cada nova varredura produzida por um escaneamento proveniente do sensor *Laser RangeFinder* (LRF), ou Ladar, com o mapa que ele construiu até

o momento, a fim de encontrar a rotação e a translação do robô desde a verificação anterior. A correspondência é conseguida utilizando um algoritmo genético (implementado em *hardware*) e uma representação de dados em ponto fixo. Sempre que o algoritmo define uma nova posição do robô, ele atualiza o mapa global. Os autores relatam uma velocidade na ordem de 4,83 vezes mais rápido comparado com o algoritmo implementado em *software*.

Adicionalmente, [21] desenvolveu uma implementação mista *hardware/software* para o problema de localização absoluta utilizando arquitetura reconfigurável FPGA, baseada no processador Nios II e em um acelerador de *hardware*. Neste caso, o método de localização utiliza uma *webcam* externa para rastreamento de imagens, o que os autores chamam de *Algoritmo de Localização Absoluta*. A posição absoluta é calculada comparando as imagens atuais com uma imagem de referência (na verdade, faz-se referência à subtração comum do algoritmo para detectar o movimento do objeto). O problema desta técnica é a necessidade para mapear a imagem de dados para o cenário atual, que introduz erro relacionado com a resolução da imagem e operações de dados, normalmente implementado usando uma representação de número inteiro. Este sistema também implementa o movimento do robô, especificando os pontos inicial e final. No entanto, a abordagem ainda não é suficiente para garantir uma navegação robótica contínua e em tempo real.

Outras abordagens da filtragem de Kalman têm sido propostas para serem implementadas em FPGAs, levando em consideração as vantagens do paralelismo intrínseco do algoritmo e o potencial do FPGA para a sua aplicação. Todas as implementações tentam lidar com a complexidade das operações matriciais. Por exemplo, [22] apresenta em seu trabalho uma solução aproximada do Filtro de Kalman (KF) usando a expansão de Taylor, que é aplicada em uma FPGA Virtex-4 da Xilinx. A ideia principal desse trabalho é eliminar a inversão de matriz na equação do ganho (veja Equação (3.9)) recorrendo à expansão de Taylor e a cálculos matriciais. Como é uma aproximação, existe um erro associado, mas os autores não o calcularam. Segundo eles, isto seria dispensável desde que os parâmetros do KF sejam atualizados. Essa implementação em *hardware* é composta por três (3) módulos: *multiplicação de matriz*, *adição/subtração de matriz* e *soma ponderada de matrizes*. Os autores ainda relatam uma outra forma de implementação do algoritmo KF baseada no algoritmo Bierman-Thorton [23, 24]. No mesmo contexto, os resultados também foram comparados com uma implementação convencional em C++ do filtro de Kalman. Foi demonstrado pelos resultados que a implementação aproximada do filtro de Kalman tem um melhor desempenho em consumo de *hardware* e *throughput* do que as outras implementações. A precisão dos resultados, bem como a taxa de convergência possuem boa aceitabilidade. Porém, não fica claro se os dados são tratados como inteiros, ponto fixo ou flutuante.

[25] apresenta uma solução em FPGA para o problema da filtragem de Kalman usando arranjos sistólicos baseados no algoritmo modificado de Faddeev [26]. De acordo com o texto, o

algoritmo usa uma técnica chamada *Neighbor Pivoting* para triangularização, o que substituiria a eliminação gaussiana e garantiria a estabilidade do fluxo de dados, bem como a redução de área do silicone. Essa implementação usa uma matriz trapezoidal ao invés de uma matriz bi-trapezoidal e é composta por duas sub-matrizes: triangular e retangular. O projeto foi implementado em uma FPGA Stratix da Altera e usou dados em ponto flutuante de 32 *bits*. O projeto contou também com bibliotecas de módulos parametrizados (LPM, do inglês ***L**ibrary **P**arameterized **M**odules*).

Este cenário apresenta um desafio para alcançar implementações reais do algoritmo EKF em FPGAs (usando ponto flutuante), em termos de baixa potência e consumo de recursos de *hardwares*, produzindo um desempenho adequado para a área da robótica móvel. Neste contexto, a abordagem sequencial do EKF surge como uma técnica adequada para ser incorporada em plataformas FPGA, devido ao fato das dimensões pequenas das matrizes, o que torna fácil suas operações, e também reduz o número de operações aritméticas tais como adição, subtração, multiplicação e divisão.

2.3 Conclusões do Capítulo

Neste capítulo foram apresentados o conceitos básicos de localização de robô, bem como alguns dos algoritmos usados para a sua implementação: *Localização por GRID*, *Localização de Monte Carlo*, *Localização por Markov* e *Algoritmo EKF*.

Ao final de cada subseção (onde se descreve os algoritmos) foi apresentado um resumo enumerando algumas vantagens e desvantagens das mesmas. E, na Tabela 2.1 apresentamos um resumo dos principais pontos relacionados a estas técnicas.

Tabela 2.1. Comparação das diferentes implementações da localização por Markov

	EKF	**MCL**	**GRID (Fina)**
Medições	*Landmarks*	Linhas de medição	Linhas de medição
Ruído (Medição)	Gaussiana	Qualquer	Qualquer
Posterior	Gaussiana	Partículas	Histograma
Eficiência (Memória)	++++	+++	+
Eficiência (Tempo)	++++	+++	+
Implementação	+++	++++	+
Resolução	+++	++	++
Robustez	+	+++	+++
Localização Global	Não	Sim	Sim

Visando um trabalho mais complexo, cuja ideia é trabalhar com o problema de localização e mapeamento simultâneos (SLAM), principalmente aplicados à robôs de pequeno porte, escolheu-se o algoritmo EKF pelos seguintes motivos:

(a) Ele serve de base para o algoritmo SLAM;

(b) Implementa a fusão de sensores;

(c) É indicado como preditor de estados para Sitemas Dinâmicos Não-Lineares.

Por fim, foram apresentados alguns trabalhos correlatos à esta implementação e verificou-se que a abordagem sequencial EKF com aplicação de operadores em ponto flutuante ainda não foi implementada na literatura em plataformas FPGAs.

3 Localização EKF

3.1 O Filtro de Kalman Estendido

O Filtro de Kalman é uma das ferramentas de estimativa mais úteis disponíveis hoje em dia, proporcionando um método repetitivo de estimar o estado de um sistema dinâmico na presença de ruído. O filtro de Kalman pode produzir estimativas dos valores reais das medições e os seus valores calculados associados ao prever um valor, estimar a incerteza do valor previsto, bem como o cálculo de uma média ponderada do valor previsto e o valor medido. O maior peso é dado para o valor com o menor grau de incerteza [14].

A estimativa produzida pelo método tende a estar mais perto dos valores reais do que as medições iniciais, dado que a média ponderada tem uma incerteza de estimativa melhor do que qualquer um dos valores individuais que realizam essa mesma média [14]. A Figura 3.1 mostra a estrutura geral do Filtro de Kalman e suas duas fases: (a) predição e (b) correção. A filtragem de Kalman tem muitas aplicações na tecnologia e as respectivas extensões e generalizações do método também foram desenvolvidas, por exemplo *estendida* (EKF) e *unscented* (UKF, do inglês ***U**nscented **K**alman **F**ilter*).

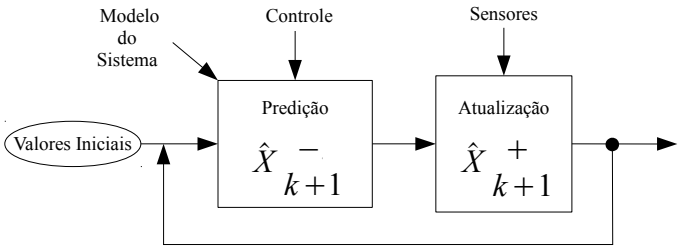


Figura 3.1. Estrutura Geral do Filtro de Kalman

Como mencionado anteriormente, o Filtro de Kalman é um excelente estimador de estados para sistemas lineares, porém pode ser difícil a sua aplicação na prática. Por exemplo, um robô que se move por um ambiente com velocidades rotacionais e translacionais constantes, tipicamente descreve uma trajetória circular, cujo o próximo estado não pode ser estimado pelo KF. O filtro de Kalman estendido supera essa limitação: a presunção de linearidade do sistema.

Considere o sistema não linear representado pelas equações (3.1) e (3.2):

$$x_{k+1} = f(x_k, u_k, w_k) \tag{3.1}$$

$$y_{k+1} = h(x_{k+1}) + v_{k+1} \tag{3.2}$$

Aqui, assume-se que a probabilidade do próximo estado e as probabilidades da medição são dirigidas pelas funções não lineares $f()$ e $h()$, sendo $f()$ uma função não linear do *sistema do processo* e $h()$ uma função não linear do *sistema de medição*. Neste caso, essas funções podem ser usadas para propagar ambos o vetor de estados x_{k+1} e o vetor de saída y_{k+1}. E por fim, u_k e w_k representam o controle e o ruído do processo respectivamente, enquanto que v_{k+1} está associado ao ruído da medição.

O filtro de Kalman estendido (EKF) calcula uma aproximação do valor verdadeiro. Ele representa essa aproximação por uma Gaussiana. Em particular, a crença $bel(x_t)$ no tempo t é representada por uma média μ_t e covariância $\sum t$, como apresentados no Algoritmo (4) [14].

A principal ideia para a aplicação do EKF se chama *linearização*. Suponha que temos uma função $f()$ não linear. A Gaussiana projetada através desta função é tipicamente não Gaussiana. Isto acontece porque a não linearidade em $f()$ distorce a forma Gaussiana para o próximo estado. A linearização aproxima $f()$ por uma função linear, que é tangente a $f()$ na média da Gaussiana. Ao projetar a Gaussiana através desta aproximação linear, a crença posterior é Gaussiana. De fato, uma vez que $f()$ é linearizada, a mecânica de propagação da crença é equivalente à do filtro de Kalman. O mesmo se aplica para a multiplicação de Gaussianas quando a função de medição de $h()$ está envolvida. Mais uma vez, o EKF aproxima $h()$ por uma tangente à função linear $h()$, mantendo assim a natureza gaussiana da crença posterior [14].

Existem muitas técnicas para linearizar funções não lineares. O EKF utiliza um método chamado de *Expansão de Taylor* (primeira ordem) [27]. A expansão de Taylor constrói uma aproximação linear da função $f()$ a partir de seu valor e inclinação. Esta inclinação é dada pela derivação parcial da função $f()$, como visto na equação 3.3:

$$f'(x_k, u_{k+1}) = \frac{\partial f(x_k, u_{k+1})}{\partial x_k} \tag{3.3}$$

As linearizações de $f()$ e $h()$ em relação às variáveis de estado e ao ruído do processo são usadas pelo algoritmo EKF. Elas são denominadas de matrizes Jacobianas e podem ser vistas

nas equações 3.4, 3.5 e 3.6.

$$F = Df(x) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix} \tag{3.4}$$

$$H = Dh(x) = \begin{bmatrix} \frac{\partial h_1}{\partial x_1} & \frac{\partial h_1}{\partial x_2} & \cdots & \frac{\partial h_1}{\partial x_n} \\ \frac{\partial h_2}{\partial x_1} & \frac{\partial h_2}{\partial x_2} & \cdots & \frac{\partial h_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial h_n}{\partial x_1} & \frac{\partial h_n}{\partial x_2} & \cdots & \frac{\partial h_n}{\partial x_n} \end{bmatrix} \tag{3.5}$$

$$W = Df(w) \begin{bmatrix} \frac{\partial f_1}{\partial w_1} & \frac{\partial f_1}{\partial w_2} & \cdots & \frac{\partial f_1}{\partial w_n} \\ \frac{\partial f_2}{\partial w_1} & \frac{\partial f_2}{\partial w_2} & \cdots & \frac{\partial f_2}{\partial w_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial w_1} & \frac{\partial f_n}{\partial w_2} & \cdots & \frac{\partial f_n}{\partial w_n} \end{bmatrix} \tag{3.6}$$

No algoritmo EKF, as matrizes Jacobianas F e H não são constantes. Entretanto, se elas forem avaliadas em um valor específico do vetor de estados, $x = x_0$, as matrizes Jacobianas correspondentes tornam-se constantes [27]. Nesse caso, a etapa da *predição* (no caso discreto do EKF) pode ser definida pelas Equações (3.7) e (3.8). Já, a etapa da *correção*, pelas equações (3.9), (3.10) e (3.11).

Predição:

$$\hat{X}_k^- = f(\hat{X}_{k-1}^+, u_{k-1}) \tag{3.7}$$

$$P_k^- = F P_{k-1}^+ F^T + W Q_{k-1} W^T \tag{3.8}$$

Correção:

$$K_k = P_k^- H^T \left[H P_k^- H^T + R_k \right]^{-1} \tag{3.9}$$

$$\hat{X}_k^+ = \hat{X}_k^- + K_k \left[Y_k - h(\hat{X}_k^-) \right] \tag{3.10}$$

$$P_k^+ = P_k^- - K_k H P_k^- \tag{3.11}$$

Como poder ser visto, as Equações (3.7) a (3.11) representam operações matriciais tais como adição, subtração, multiplicação e inversão. A Tabela 3.1 exibe a notação empregada do algoritmo EKF.

Tabela 3.1. Descrição dos símbolos usados no algoritmo EKF

Símbolos	Descrição
X	Pose do Robô
u	Controle do Robô
P	Covariância da Pose do Robô
F	Jacobiana do Movimento do Robô
W	Jacobiana do Ruído do Movimento do Robô
Q	Ruído do Movimento Permanente
K	Ganho do EKF
H	Jacobiana da Medição
R	Ruído da Medição Permanente
Y	Medição do Sensor
h()	Função Não Linear da Medição
k	Tempo/Estado Anterior
k+1	Tempo/Estado Actual
-	Informação *a priori*
+	Informação *a posteriori*

3.2 O Algoritmo Sequencial do EKF

Como visto na seção anterior, as equações do algoritmo EKF são praticamente operações matriciais: soma, subtração, multiplicação e inversão. Sabe-se que tais operações demandam intenso calculo computacional e tempo, com o aumento dimensional das matrizes [2]. Por exemplo, a Figura 3.2 apresenta um gráfico relacionando o tempo de execução da operação inversão pela dimensão da matriz; como observado, quanto maior a dimensão matricial, maior será o tempo de execução da operação.

O tempo e a demanda computacional podem ser reduzidos se trabalha-se com dimensões matriciais menores. Para isto, optou-se por utilizar a versão sequencial do algoritmo EKF, versão que está ligada diretamente com a matriz Y (medição do sensor). Esta matriz, vista logo mais abaixo, pode ser composta (a cada instante de tempo) pelo valor de um sensor, ou de vários sensores. Se opta-se pelo processamento de um sensor por instante de tempo, tem-se a versão sequencial; porém, se deseja-se trabalhar com todos os sensores no mesmo instante de tempo,

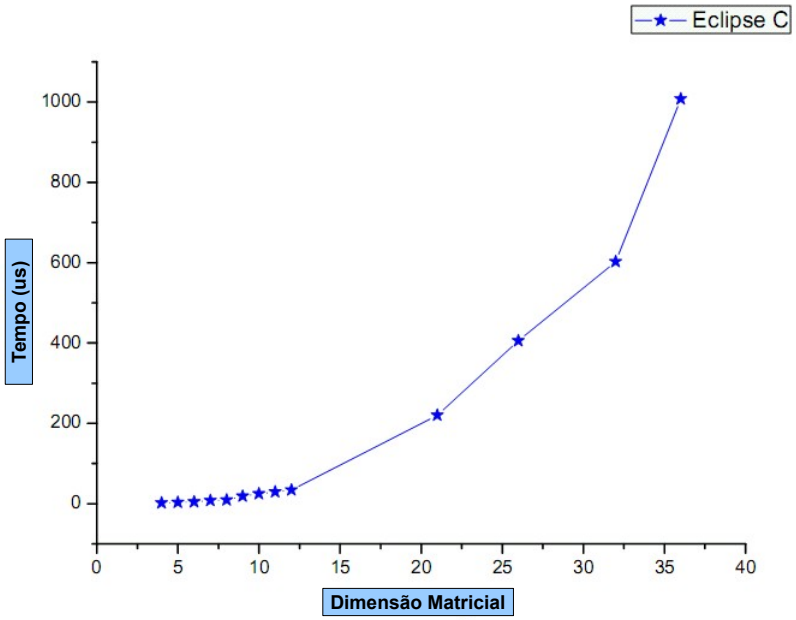


Figura 3.2. Tempo consumido por um PC para realizar inversão de matriz escrita em linguagem C, adaptado de [2]

tem-se a versão cheia (*full*) do algoritmo.

3.2.1 Representação do Vetor de Estado e da Matriz de Covariância

Em nosso sistema o vetor de estados X é representado pela estimação da pose do robô, assim como a covariância de cada variável de estado é representada pela matriz P, veja as Equações (3.12) e (3.13), respectivamente. Os elementos diagonais de P são as variâncias e os não diagonais são as covariâncias entre as variáveis de estado.

$$\hat{X}_k = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}_{3x1} \tag{3.12}$$

$$P_k = \begin{bmatrix} \sigma_x^2 & \sigma_{xy} & \sigma_{x\theta} \\ \sigma_{yx} & \sigma_y^2 & \sigma_{y\theta} \\ \sigma_{\theta x} & \sigma_{\theta y} & \sigma_\theta^2 \end{bmatrix}_{3x3} \tag{3.13}$$

3.2.2 Representação do Robô e seu Sistema de Medição

Este trabalho focaliza somente o problema de localização de robôs, também é considerado que tanto o mapa do ambiente como a posição inicial do robô são conhecidas. Para cada medição feita através do sensores, os dados são equiparados (*matched*) com o mapa. A Figura 3.3 descreve o robô Pioneer 3AT posicionado no sistema global com orientação θ e a representação dos símbolos é mostrada na Tabela 3.2.

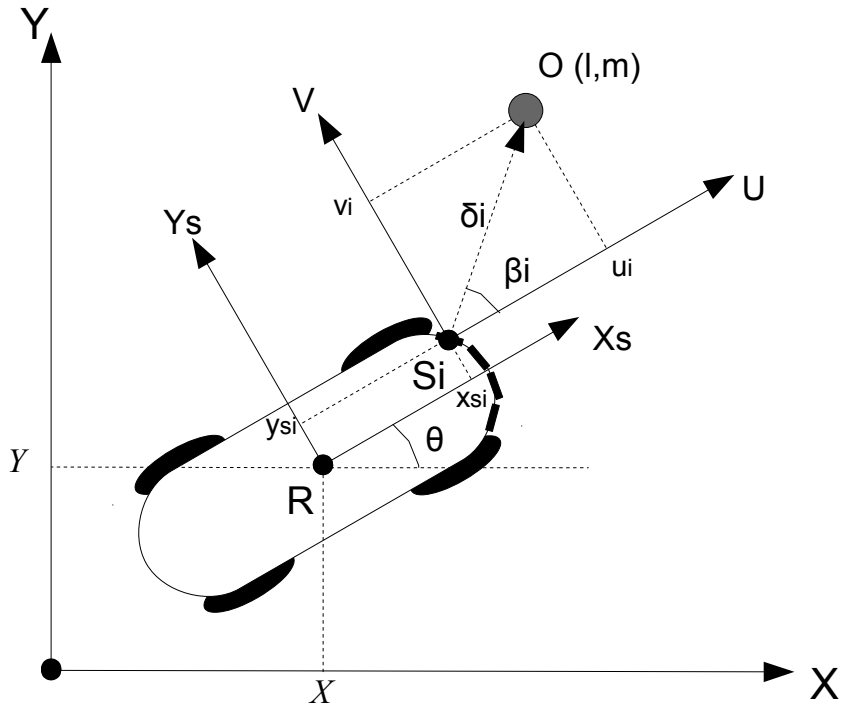


Figura 3.3. Robô Pioneer 3-AT posicionado em um ambiente de duas dimensões

3.2.3 Obtenção da Representação Sequencial do EKF

Dado que foi implementada a versão sequencial do algoritmo EKF, as medições foram realizadas também na forma sequencial. Em outras palavras, uma estimação do vetor de estados é efetuada por cada medição ao invés de todas todas as medições. Assim, o tamanho das matrizes são reduzidas, bem como a complexidade da implementação em *hardware*. Por exemplo, as Equações (3.14) e (3.15) representam medições sequenciais e cheias, respectivamente, para o vetor de medição do sensor.

$$Y_k = \begin{bmatrix} u_i \\ v_i \end{bmatrix}_{2x1} \tag{3.14}$$

Tabela 3.2. Descrição dos símbolos usados na representação da posição do Pioneer 3AT

Símbolos	Descrição
(X,Y)	Sistema de Coordenadas Global - SCG
(x,y)	Posição do Robô no SCG
θ	Orientação do Robô no SCG
(Xs,Ys)	Sistema de Coordenadas do Robô - SCR
(Xsi,Ysi)	Posição do Sensor i no SCR
(Ui,Vi)	Sistema de Coordenadas do Sensor Si - SCSi
σ_i	Distância entre o objeto O e o sensor Si
β_i	Ângulo de medição
(ui,vi)	Posição do objeto O no SCSi
(l,m)	Posição do objeto O no SCG

$$Y_k = \begin{bmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \\ \vdots \\ u_n \\ v_n \end{bmatrix}_{2nx1} \tag{3.15}$$

Para resolver o problema de localização, usando a versão sequencial, uma transformação de coordenadas do objeto O do SCG para o SCSi foi realizada e assim, a função não linear $h_i(\hat{X}_k^-)$ ficou representada pelas Equações (3.16) e (3.17).

$$\left\{ \begin{bmatrix} cos(\hat{\theta}_k) & sin(\hat{\theta}_k) \\ -sin(\hat{\theta}_k) & cos(\hat{\theta}_k) \end{bmatrix} \cdot \left[\begin{pmatrix} l_k \\ m_k \end{pmatrix} - \begin{pmatrix} \hat{x}_k \\ \hat{y}_k \end{pmatrix} \right] \right\} - \begin{bmatrix} x_{si} \\ y_{si} \end{bmatrix} \tag{3.16}$$

ou,

$$\begin{bmatrix} (l_k - \hat{x}_k) \cdot cos(\hat{\theta}_k) + (m_k - \hat{y}_k) \cdot sin(\hat{\theta}_k) - x_{si} \\ (m_k - \hat{y}_k) \cdot cos(\hat{\theta}_k) + (\hat{y}_k - L_k) \cdot sin(\hat{\theta}_k) - x_{si} \end{bmatrix}_{2x1} \tag{3.17}$$

A matriz Jacobiana H foi obtida aplicando a Equação (3.5), logo, a versão final ficou representada pelas Equações (3.18) e (3.19).

$$H = \frac{\partial h()}{\partial X_k} = \begin{bmatrix} \frac{\partial h_1}{\partial x} & \frac{\partial h_1}{\partial y} & \frac{\partial h_1}{\partial \theta} \\ \frac{\partial h_2}{\partial x} & \frac{\partial h_2}{\partial y} & \frac{\partial h_2}{\partial \theta} \end{bmatrix}_{2x3} \tag{3.18}$$

ou,

$$\begin{bmatrix} -cos(\hat{\theta}_k) & -sin(\hat{\theta}_k) & -(l_k - \hat{x}_k) \cdot sin(\hat{\theta}_k) + (m_k - \hat{y}_k) \cdot cos(\hat{\theta}_k) \\ sin(\hat{\theta}_k) & -cos(\hat{\theta}_k) & -(l_k - \hat{x}_k) \cdot cos(\hat{\theta}_k) - (m_k - \hat{y}_k) \cdot sin(\hat{\theta}_k) \end{bmatrix}_{2x3} \tag{3.19}$$

Por outro lado, R é a matriz de ruído permanente da medição, sendo intrínseca de cada sensor. Ela pode ser representada pela variância da medição em cada eixo do SCSi (vide Equação (3.20)).

$$R = \begin{bmatrix} \sigma_{ui}^2 & 0 \\ 0 & \sigma_{vi}^2 \end{bmatrix}_{2x2} \tag{3.20}$$

Finalmente, a Tabela 3.3 exibe as dimensões matriciais do algoritmo sequencial EKF.

Tabela 3.3. Dimensão matricial dos símbolos usados no algoritmo sequencial EKF

Símbolos	**Dimensão Matricial**
X	3x1
u	Não usado
P	3x3
F	Não usado
W	Não usado
Q	Não usado
K	3x2
H	2x3
R	2x2
Y	2x1
h()	2x1

3.3 Conclusões do Capítulo

Neste capítulo foram explicados os conceitos da filtragem de Kalman estendida (EKF) para o problema de localização robótica, bem como os algoritmos que serão implementados (em *hardware*) neste trabalho para a solução da mesma.

Os algoritmos apresentados neste capítulo possuem um certo custo computacional devido à certas operações matriciais, tais como multiplicação e inversão. Porém, isso pode ser contornado se aplicarmos a versão sequencial do algoritmo EKF, que reduz a dimensão das matrizes e facilita a implementação em *hardware*.

Para a execução correta do algoritmo, considera-se o conhecimento do mapa onde o robô se encontra, bem como a posição do robô na mesma. Com isso, o problema de *match* é solucionado.

4 Os FPGAs e as Unidades em Ponto Flutuante

4.1 Arquiteturas Reconfiguráveis FPGA

O FPGA é um dispositivo semicondutor que pode ser programado após sua fabricação. Especificamente, um FPGA contém componentes lógicos programáveis chamados de elementos lógicos (LEs, do inglês ***L**ogical **E**lements*) e uma hierarquia das interconexões reconfiguráveis que permitem aos LEs estarem fisicamente ligados. Pode-se configurar os LEs para executar funções combinacionais complexas, ou portas lógicas meramente simples, como AND e XOR. Na maioria dos FPGAs, os blocos lógicos também incluem elementos de memória, que podem ser simples *flip-flops* ou blocos completos de memória [28].

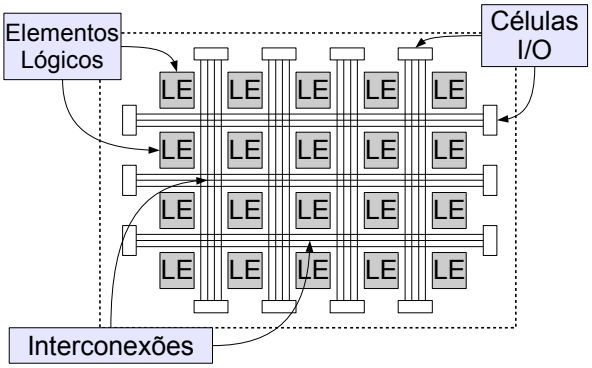


Figura 4.1. Estrutura geral de um FPGA

O FPGA é composto basicamente por três tipos de componentes (ver Figura 4.1): blocos de entrada e saída (IOB, do inglês ***I**nput **O**utput **B**lock*), blocos lógicos configuráveis (CLB, do inglês ***C**onfigurable **L**ogic **B**lock*) e chaves de interconexão (*Switch Matrix*). Os blocos lógicos estão dispostos de forma bidimensional, as chaves de interconexão estão dispostas em formas de trilhas verticais e horizontais entre as linhas e as colunas dos blocos lógicos. Os CLBs são circuitos idênticos, construído pela reunião de *flip-flops* e a utilização de lógica combinacional. Utilizando os CLBs, um usuário pode construir elementos funcionais lógicos. Os IOB são circuitos responsáveis pelo interfaceamento das saídas provenientes das saídas das combinações dos CLBs. São basicamente *buffers*, que funcionarão como um pino bidirecional de entrada e saída do

FPGA. As Chaves de Interconexões são trilhas utilizadas para conectar os CLBs e IOBs. Este terceiro grupo é composto pelas interconexões. Os recursos de interconexões possuem trilhas para conectar as entradas e saídas dos CLBs e IOBs para as redes apropriadas. Geralmente, a configuração é estabelecida por programação interna das células de memória estática, que determinam funções lógicas e conexões internas implementadas no FPGA entre os CLBs e os IOBs. O processo de escolha das interconexões é chamado de roteamento [29].

Como principais vantagens dos FPGAS citamos:

(a) Velocidade de processamento - implementações baseadas em fluxo de dados (ao invés de fluxo de instruções) e executadas diretamente em *hardware*;

(b) Flexibilidade - Alto poder de ajuste a funções específicas, mediante reconfiguração estática ou dinâmica;

(c) Baixo custo;

(d) Rápido desenvolvimento de protótipos;

(e) Relativamente fácil de se usar.

Este capítulo faz uma descrição das unidades de ponto flutuante desenvolvidas, previamente, em diferentes trabalhos de pesquisa realizados no GRACO-UnB, e que foram utilizadas e adaptadas durante o desenvolvimento deste trabalho.

4.2 Unidades em Ponto Flutuante - (UPFs)

Esta seção apresenta uma breve descrição das arquiteturas em *hardware* para operações aritméticas em ponto flutuante de precisão simples desenvolvidas por [30, 31, 4], devido a que este trabalho de mestrado fez uso dessas unidades. Para a utilização das mesmas foram realizadas algumas modificações, especificamente na arquitetura de soma/subtração para adequá-la aos FPGAs da Altera, já que as mesmas foram concebidas para FPGAs da Xilinx.

4.2.1 O Somador/Subtrator em Ponto Flutuante

Os passos para realizar a soma/subtração em ponto flutuante são descritos abaixo:

(a) Separar as entradas sinal, o exponente e a mantissa e checar se elas são zero, infinito ou uma representação inválida no padrão IEEE-754. Adicionar o *bit* escondido (*hidden*) à mantissa.

(b) Comparar as duas entradas: uma operação lógica de deslocamento à direita deve ser realizada sobre o menor dos dois números. O número de *bits* da mantissa deslocados à direita, baseado na diferença do exponente, e essa diferença é o resultado preliminar do cálculo do exponente. Finalmente, some/subtrai as mantissas atuais.

(c) Desloque à esquerda a mantissa alcançada até seu *bit* mais significativo (MSB, do inglês ***M**ost **S**ignificant **B**it*) ser 1. Para cada deslocamento decremente o exponente atual por 1. Por fim, concatene o sinal, o exponente e a mantissa do resultado final.

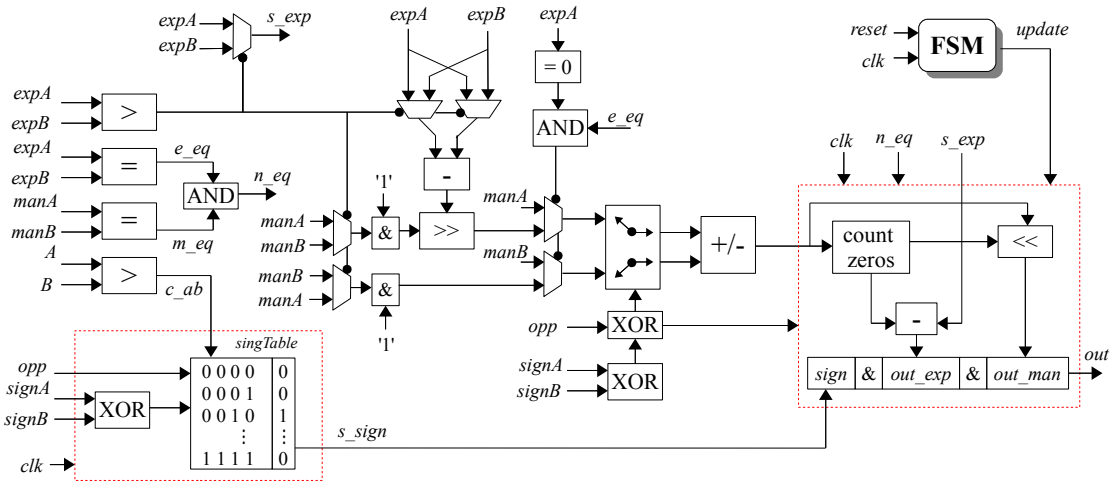


Figura 4.2. Arquitetura em *hardware* da unidade soma/subtração *FPadd* [3]

A Figura (4.2) descreve a arquitetura em *hardware* da unidade somador/subtrator, nomeada *FPadd*. Os blocos em linha pontilhada são processos sensíveis ao relógio. Essa arquitetura faz uso de dois ciclos de relógio para realizar a operação de soma/subtração em aritmética de ponto flutuante. No primeiro ciclo de relógio algumas exceções são validadas em ordem de identificar se os argumentos de entrada são zero, infinito ou uma representação padrão IEEE 754 (ver Figura 4.3) não válida. Os argumentos de entrada também são comparados, o resultado do sinal e o resultado preliminar do exponente são computados e as mantissas são alinhadas de acordo com as diferenças dos exponentes dos argumentos de entrada. Durante o segundo ciclo do relógio, o resultado da mantissa é normalizado e concatenado com o resultado do exponente e o *bit* de sinal. Uma máquina de estado finitos (FSM, do inglês ***F**inite **S**tate **M**achine*) controla o processo de atualização do resultado final, que é registrado usando *flip-flops* [3].

4.2.2 O Multiplicador em Ponto Flutuante

Os passos para realizar a multiplicação em ponto flutuante são descritos abaixo:

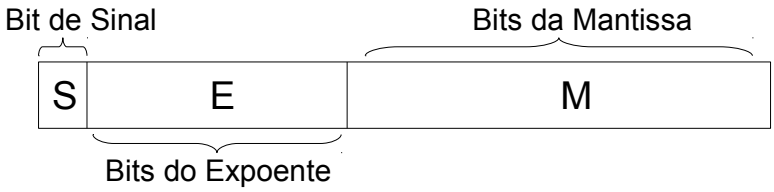


Figura 4.3. Estrutura da representação numérica no padrão IEEE-754

(a) Separar as entradas sinal, o exponente e a mantissa e checar se elas são zero, infinito ou uma representação inválida no padrão IEEE 754. Adicionar o *bit* escondido (*hidden*) à mantissa.

(b) Multiplicar as mantissas, adicionar exponentes e determinar o sinal produto.

(c) Se o MSB é 1 no resultado da multiplicação das mantissas, portanto, não é necessária normalização. A mantissa atual é deslocada à esquerda até que um 1 seja alcançado. Para cada operação de deslocamento, o exponente atual é decrementado por 1. Finalmente, concatena-se o sinal, exponente e mantissa dos resultados finais.

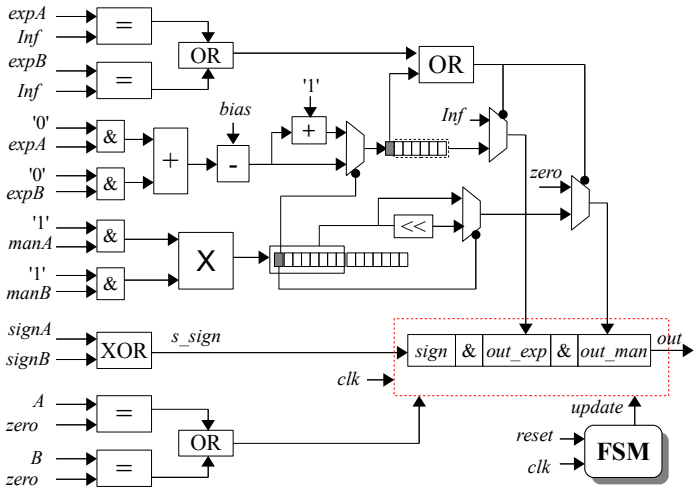


Figura 4.4. Arquitetura em *hardware* da unidade multiplicação *FPmul* [3]

A Figura 4.4 descreve a arquitetura de *hardware* da unidade de multiplicação (*FPmul*), o qual é dividido em duas fases, que requer um ciclo de relógio, para cada um deles. Na primeira fase, algumas exceções são validadas, a fim de identificar se os operandos são zero, infinito ou uma representação padrão IEEE-754 não válido. Além disso, o sinal, o expoente preliminar e multiplicação das mantissas são computadas. A operação lógica *xor* é usada para comparar o sinal de entrada de argumentos e, em seguida, determinar o resultado do sinal. Note que os expoentes são adicionados com um *bit* extra indicando *overflow*. Por conseguinte, o valor resultante tem

uma polarização dupla e o expoente preliminar é calculado subtraindo-se o enviesamento (*bias*) da adição do expoente. Também é importante ressaltar que nesta arquitetura a mantissa de M_w *bits* com o *bit* implícito são multiplicadas, resultando em uma palavra $2(M_w1)$ que é truncado nos primeiros $M_w + 2$ *bits* [3].

A segunda etapa avalia se o MSB do resultado da multiplicação de mantissas é 1. Este *bit* controla um multiplexador, que aborda o resultado da mantissa final. Além disso, se o MSB do resultado da multiplicação mantissas é um expoente preliminar deve ser adicionado por 1. Em seguida, o resultado do expoente é avaliado de modo a identificar um excesso ou uma representação de um número infinito. Em caso afirmativo, o resultado final é infinito, caso contrário o sinal, os resultados de expoente e mantissa são concatenadas e registradas. Uma FSM controla o processo de atualização do resultado final [3].

4.2.3 A Divisão em Ponto Flutuante

O algoritmo generalizado para calcular a divisão é descrita abaixo.

(a) Seja X e Y números reais representados no padrão IEEE-754, onde X representa o dividendo e Y o divisor.

(b) Separar as entradas sinal, o exponente e a mantissa e checar se elas são zero, infinito ou uma representação inválida no padrão IEEE 754. Adicionar o *bit* escondido (*hidden*) à mantissa.

(c) Calcular o resultado da mantissa usando o algoritmo de Newton-Raphson para divisão descrito abaixo. Em paralelo a isso, avaliar o expoente resultado, ou seja, expoente (X) - expoente (Y) + *Bias*, e avaliar o sinal do resultado.

4.2.3.1 O algoritmo Newton-Raphson para divisão

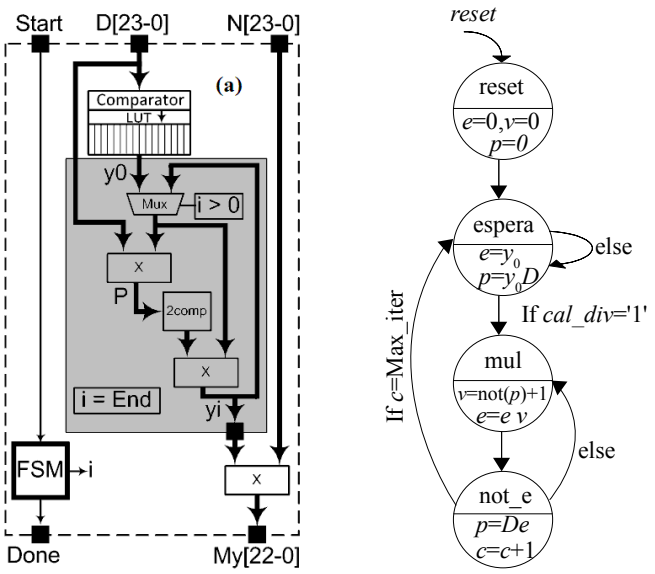
O algoritmo Newton-Raphson tem duas entradas de $n - bits$, N e D, que satisfaçam $1 = N$, $D < 2$, a partir de uma aproximação inicial para $y_0 = 1/D$. Equações 4.1a e 4.1b devem ser executadas em um caminho iterativo [32].

$$p = Dy_i \tag{4.1a}$$

$$y_{i+1} = y_i(2 - p) \tag{4.1b}$$

Após a $i - sima$ iteração, a aproximação N/D é encontrada multiplicando $N \cdot y_{i+1}$. O algoritmo de Newton-Raphson usa o valor de D para refinar, a cada iteração, a aproximação

inicial de $1/D$ e na última iteração a aproximação final é multiplicada por N [32]. Este fato permite que o algoritmo de Newton-Raphson calcule menos multiplicações do que algoritmos baseados na iteratividade que aproximam o valor N/D em cada iteração [4]. A implementação em *hardware* desse algoritmo e sua respectiva descrição de máquina de estado finito é mostrada na Figura 4.5 [3].



(a) arquitetura (b) FSM

Figura 4.5. Arquitetura em *hardware* da unidade de divisão baseado no algoritmo Newton-Raphson [3]

4.2.4 CORDIC

O algoritmo CORDIC (do inglês ***Co**ordinate **R**otation **Di**gital **C**omputer*) é um método iterativo baseado em deslocamento-adição (*add-shift*) para calcular a rotação de um vetor bi-dimensional e para calcular o comprimento e ângulo de um vetor, em outras palavras, é um algoritmo simples e eficiente para calcular funções hiperbólicas e trigonométricas. Conforme [4], as equações do CORDIC iterativo podem ser as seguintes:

$$X_{i+1} = X_i - m \cdot \sigma_i \cdot 2^{-i} \cdot Y_i \tag{4.2}$$

$$Y_{i+1} = Y_i + \sigma_i \cdot 2^{-i} \cdot X_i \tag{4.3}$$

$$Z_{i+1} = Z_i - \sigma_i \cdot \theta_i \tag{4.4}$$

onde, i é o índice de interação, θ_i são micro rotações pré computadas, σ_i é a direção da rotação e m indica o tipo de coordenada.

As Equações (4.2) a (4.4) afirmam que em cada iteração do algoritmo CORDIC precisa computar três operações em ponto flutuante soma/subtração e uma busca na memória ROM (do inglês ***R**ead **O**nly **M**emory*) no qual são armazenados a micro rotações pré computadas. A arquitetura em *hardware* FP-CORDIC para funções seno/cosseno e arco tangente difere apenas nas condições iniciais permitindo que a arquitetura para implementar dois modos diferentes de operação [3].

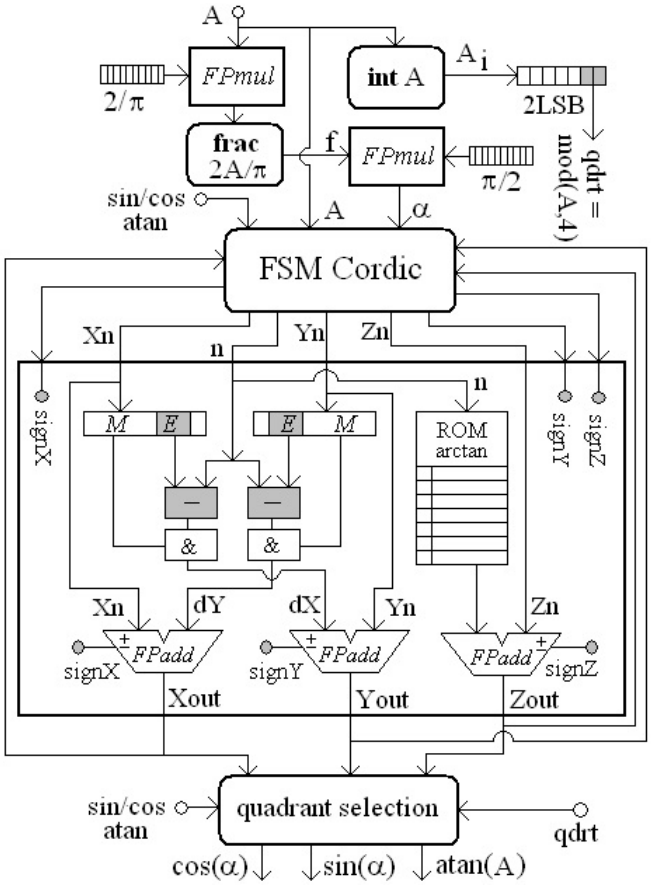


Figura 4.6. Função *sin*, *cos* e *atan* para a UPF CORDIC [4]

A arquitetura é composta por quatro unidades: *redução de argumento*, *máquina de estados finitos*,*micro rotação* e *normalização*. A estrutura base da unidade de micro rotação são três paralelo FP-add/sub (*FPadd*), ver Figura 4.6. As operações 2^{-n} são realizadas subtraindo o índice n da micro rotação atual para o expoente (E) de X e Y. Depois, uma concatenação com a mantissa e o sinal de X e Y é realizada. A unidade FSM seleciona as condições iniciais para as variáveis X, Y e Z, de acordo com a operação desejada (seno/cosseno ou atan). Além disso, a unidade *FSM* controla o processo de retorno e o sentido das rotações, enquanto, um sinal *ready* indica uma saída válida após N micro rotações. Os sinais *Start*, *reset*, *ready* e *clock* não são

desenhadas em questão de clareza [4].

A unidade *redução de argumento* considera uma redução do argumento ao quadrante para os cálculos de seno e cosseno confirmadas por $(\alpha = \delta\pi/2, \delta = (2\theta/\pi) - \kappa)$. Inicialmente, o argumento A é multiplicado por $2/\pi$ e então, a unidade *fracFP* separa as partes inteira (i) e fracional (f). Finalmente, o argumento reduzido é obtido pelo cálculo $\delta\pi/2$. Após a iteração N, a unidade *normalização* realiza a seleção do quadrante no caso do modo de rotação (seno e cosseno). Lembre que os dois *bits* menos significativos da parte inteira (i) do argumento reduzido correspondem ao valor $mod(k, 4)$, que é usado para endereçar as saídas de um dos seguinte valores: $sin(a), cos(a)$ [4].

4.2.5 Implementação em FPGA

Como mencionado anteriormente, foi necessária uma pequena alteração no código das UPFs para compilar com sucesso estas arquiteturas nos FPGAs da Altera. O código original realizava uma operação de deslocamento de *bits* usando os operadores *sll* e *srl*. Devido à incompatibilidade dos operadores com o tipo de sinal usado no código, foram criados dois processos para deslocamento de *bits* (um à direta e outro à esquerda).

Figura 4.7. Exemplo de uma operação de deslocamento de *bit*

Esses deslocamentos (ver Figura 4.7) fazem parte do processo para calcular o alinhamento das mantissas segundo a diferença dos expoentes, ou seja, dependo do valor da diferença entre os expoentes dos operandos A e B, este será o número de *bits* deslocados. Com essa correção, obteve-se sucesso na compilação dos códigos.

As arquiteturas UPFs foram compiladas usando o *software* Quartus II [33] e na Tabela 4.1 é possível observar os resultados para recurso de *hardware* para um FPGA Cyclone IV EP4CE115F29C7N. Para a utilização dos operadores aritméticos os mesmos foram ligados ao processador Nios via barramento Avalon (vide figura 4.8).

O seguinte passo da implementação foi submeter as arquiteturas em uma simulação comportamental usando a ferramenta computacional ModelSim-Altera Start Edition [34]. Neste tipo de simulação, verificou-se o comportamento das arquiteturas através de *loops* e processos, bem como

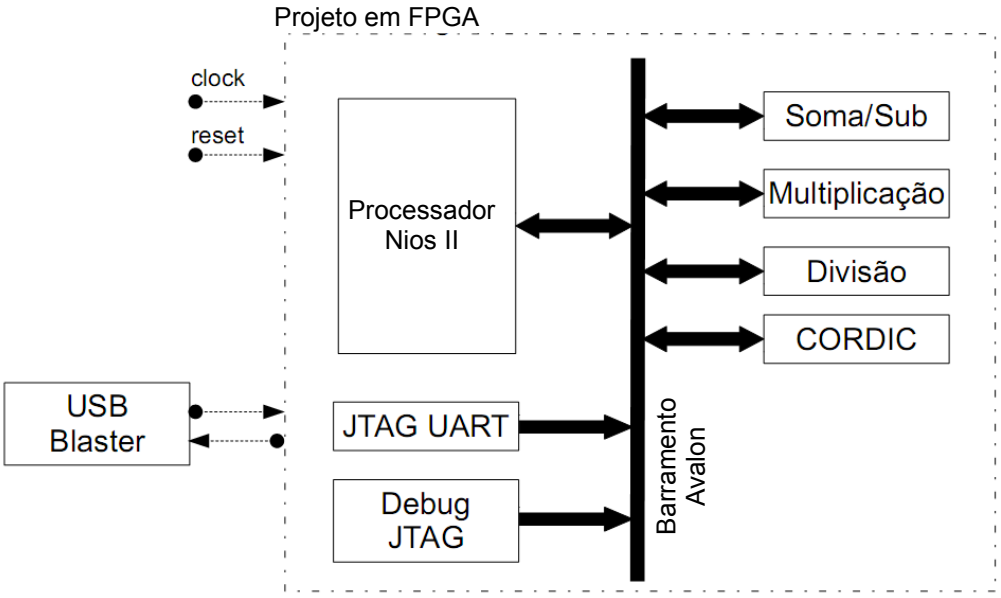


Figura 4.8. Projeto em FPGA para comunicação entre as UPFs e o processador Nios

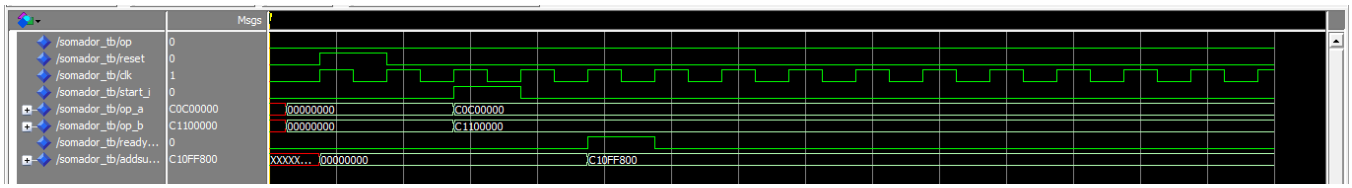
Tabela 4.1. Recursos de *hardwares* para as UPFs em um FPGA Cyclone IV EP4CE115F29C7N

UPF	LEs (*)	DSPs (*)	Fmáx (MHz)	Potência (mW)
Soma/Subtração	940 (< 1%)	0	51,47	165,09
Multiplicação	139 (< 1%)	7 (3%)	76,02	163,78
Divisão	634 (< 1%)	28 (11%)	52,44	157,24
CORDIC	3811 (3%)	7 (1%)	40,42	181,70

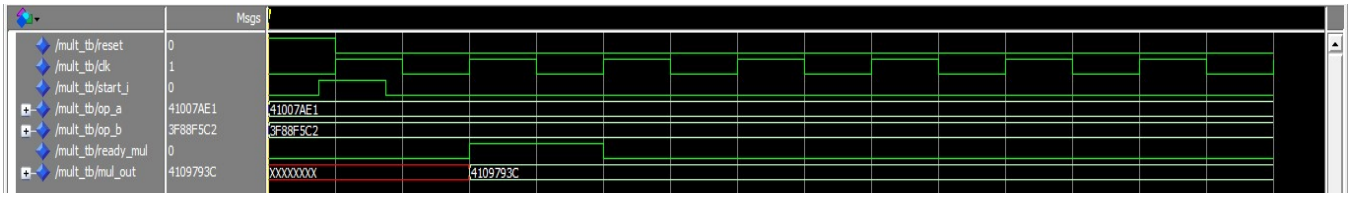
* Ocupação do *hardware*

o funcionamento das mesmas em termos de lógica combinacional (*if, then, else*) e booleana. A Figura 4.9 exibe alguns dos resultados das simulações comportamentais realizados com as UPFs. Os valores que aparecem nas figuras estão na forma hexadecimal padrão IEEE 754, por exemplo, na Figura 4.9(a), pode-se ver o resultado (add_sub) de uma soma entre o operando A (op_a) e o operando B (op_b). O resultado é C10FF800 (-15 em decimal) e os operandos são C0C00000 (-6 em decimal) E C1100000 (-9 em decimal). Pode-se observar também o ciclo de relógio para a execução da operação, no caso da soma 1 ciclo. As figuras 4.9(b), 4.9(c) e 4.9(d), representam respectivamente os resultados da simulação para multiplicação, divisão e seno (CORDIC), com ciclos de relógio de 1, 10 e 60 respectivamente.

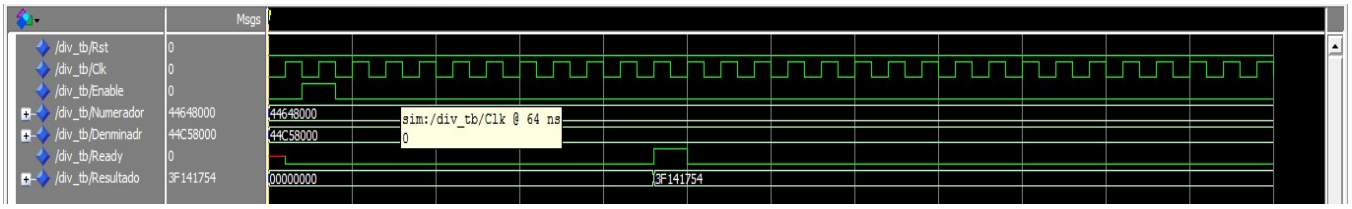
A validação em *hardware* das arquiteturas foi feita agregando-se as UPFs como *hardwares* customizados ao processador embarcado Nios, como visto na Figura 4.8. Usando a *J-tag* do



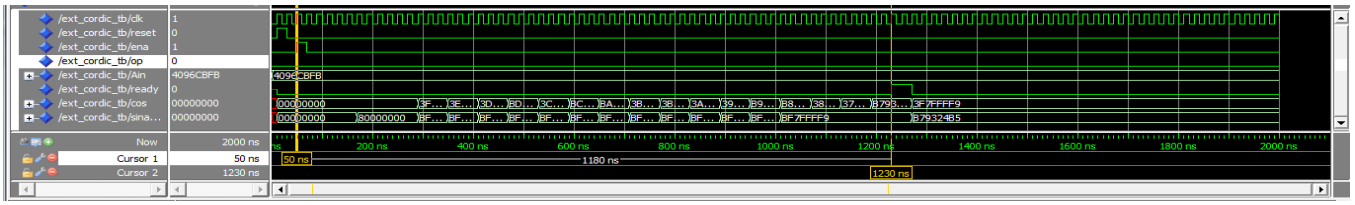
(a)



(b)



(c)



(d)

Figura 4.9. Resultado da simulação comportamental para a arquitetura (a) Soma/Subtração (b) Multiplicação (c) Divisão e (d) CORDIC

processador foi possível estabelecer uma comunicação entre o *software NiosII tools* e o processador embarcado. Para objeto de validação do sistema de cálculo em ponto flutuante, diversos valores de operandos foram enviados para as arquiteturas, e o resultados foram avaliados. Através desta comunicação (via interface *J-tag*) as arquiteturas foram avaliadas com os mais diversos valores para os operandos, introduzindo valores reais positivos e negativos.

4.3 Conclusões do Capítulo

Neste capítulo foram apresentados os FPGAs e suas principais vantagens para uso em sistemas embarcados, bem como as unidades em ponto flutuante para resolução de operações aritméticas como soma/subtração, multiplicação, divisão, seno e cosseno.

Foi detalhado, também, as alterações realizadas no código das UPFs para que fosse possível o uso dessas arquiteturas nos FPGAs da Altera, já que as mesmas foram concebidas para FPGAs da Xilink.

Por fim, foi apresentado o processo de implementação dessas arquiteturas em ponto flutuante em *hardware*, bem como os resultados de compilação. Verificou-se que as mesmas funcionaram perfeitamente e que sua aplicabilidade não se restringiu somente aos FPGAs da Xilink, mas também aos da Altera.

5 Metodologia Proposta

5.1 Primeira Abordagem para a Solução em *Hardware* do Problema de Localização: Implementação em FPGA Usando Módulos Individuais para as três Equações

Devido ao fato deste trabalho focar somente o problema de localização, as arquiteturas desenvolvidas somente implementam as equações de correção do algoritmo EKF, ou seja as Equações (3.9), (3.10) e (3.11). Neste caso, tanto a matriz Jacobiana de medição quanto a função não linear não estão incluídas no *hardware*; seus elementos são previamente calculados em *software* (usando o Nios II e a arquitetura CORDIC) e endereçados às arquiteturas em *hardware*. O cálculo do seno e cosseno, neste caso, é realizado pela arquitetura CORDIC para acelerar o computo de H e $h()$. A Figura (5.1) mostra a implementação em FPGA usando módulos individuais para as três equações de correção do EKF.

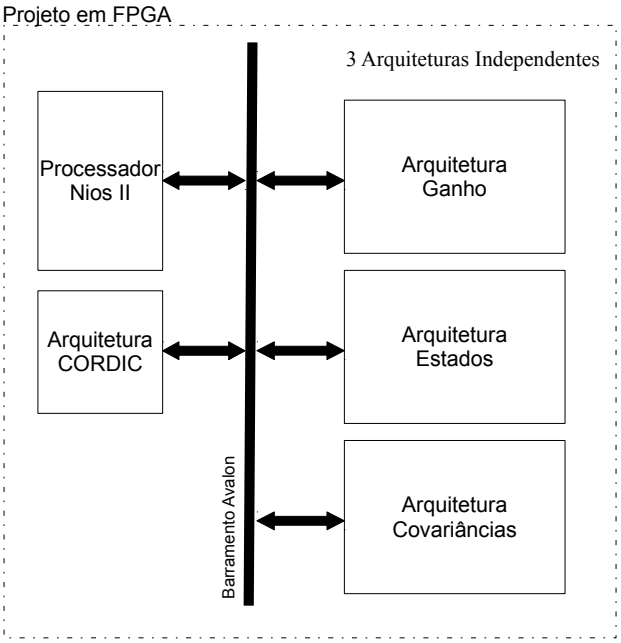


Figura 5.1. Primeira implementação em FPGA com três arquiteturas independentes: *Ganho, Estados e Covariâncias*

5.1.1 A Arquitetura do *Ganho*

A Figura (5.2) exibe a decomposição da equação do *ganho* (veja Equação (3.9)), que inclui basicamente operações matriciais, tais como multiplicação, adição e inversão. As Equações (5.1) a (5.5) descrevem a decomposição do processo, e para atingi-lo (como um todo) uma FSM foi projetada, a fim de controlar as operações em ponto flutuante envolvidas na arquitetura em *hardware*.

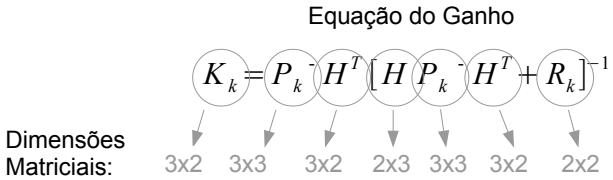


Figura 5.2. Equação do *Ganho* e suas dimensões matriciais

$$1^a etapa : O^1_{3\times 2} = P_k^- \cdot H^T \tag{5.1}$$

$$2^a etapa : O^2_{2\times 2} = H \cdot O^1_{3\times 2} \tag{5.2}$$

$$3^a etapa : O^3_{2\times 2} = O^2_{2\times 2} + R_k \tag{5.3}$$

$$4^a etapa : O^4_{2\times 2} = inv(O^3_{2\times 2}) \tag{5.4}$$

$$5^a etapa : K_k = O^1_{3\times 2} \cdot O^4_{2\times 2} \tag{5.5}$$

Pode ser observado na Equação (5.1) que a primeira etapa para computar a equação do *ganho* é uma multiplicação de matrizes ($P_k^- \cdot H^T$), como mostrado na Equação (5.6).

$$O^1_{3\times 2} = \begin{bmatrix} p^-_{11} & p^-_{12} & p^-_{13} \\ p^-_{21} & p^-_{22} & p^-_{23} \\ p^-_{31} & p^-_{32} & p^-_{33} \end{bmatrix}_{3\times 3} \cdot \begin{bmatrix} h^T_{11} & h^T_{12} \\ h^T_{21} & h^T_{22} \\ h^T_{31} & h^T_{32} \end{bmatrix}_{3\times 2} \tag{5.6}$$

O resultado da multiplicação gera uma nova matriz O^1 com dimensão 3×2, cujo os elementos são representados pelas Equações (5.7) a (5.12).

$$o^1_{11} = p^-_{11} \cdot h^T_{11} + p^-_{12} \cdot h^T_{21} + p^-_{13} \cdot h^T_{31} \tag{5.7}$$

$$o_{12}^{1} = p_{11}^{-} \cdot h_{12}^{T} + p_{12}^{-} \cdot h_{22}^{T} + p_{13}^{-} \cdot h_{32}^{T} \tag{5.8}$$

$$o_{21}^{1} = p_{21}^{-} \cdot h_{11}^{T} + p_{22}^{-} \cdot h_{21}^{T} + p_{23}^{-} \cdot h_{31}^{T} \tag{5.9}$$

$$o_{22}^{1} = p_{21}^{-} \cdot h_{12}^{T} + p_{22}^{-} \cdot h_{22}^{T} + p_{23}^{-} \cdot h_{32}^{T} \tag{5.10}$$

$$o_{31}^{1} = p_{31}^{-} \cdot h_{11}^{T} + p_{32}^{-} \cdot h_{21}^{T} + p_{33}^{-} \cdot h_{31}^{T} \tag{5.11}$$

$$o_{32}^{1} = p_{31}^{-} \cdot h_{12}^{T} + p_{32}^{-} \cdot h_{22}^{T} + p_{33}^{-} \cdot h_{32}^{T} \tag{5.12}$$

Por exemplo, para implementar a Equação (5.7), uma FSM foi projetada, realizando três multiplicações e duas adições ambas em ponto flutuante. Nesse caso, a estrutura FSM é composta por cinco estados: *waiting*, *mult1*, *mult2*, *multadd* e *add*. Como mostrado na Figura (5.3), a FSM está descrita na forma de máquina de *Mealy* (sequenciadores), onde a saída para cada transição é omitida para simplificar a sua representação.

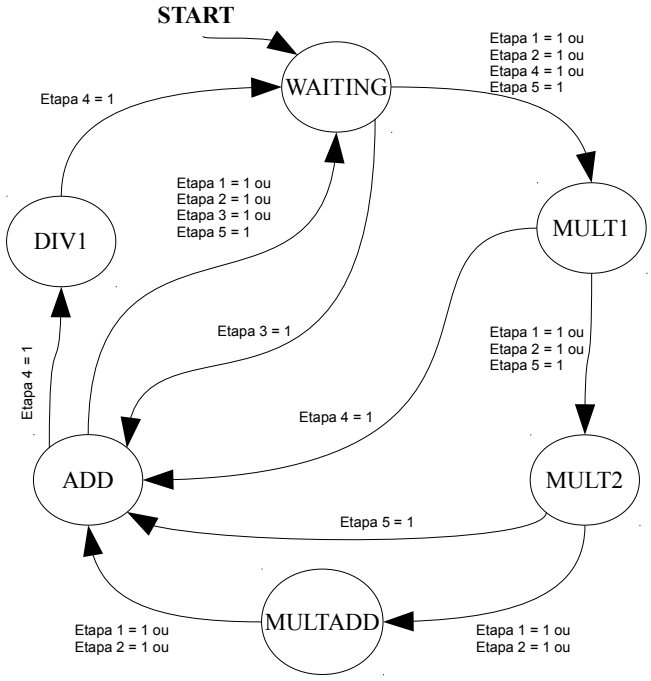


Figura 5.3. FSM usada na arquitetura para computar a equação do *ganho*

O estado *waiting* espera um sinal de *start* do processador Nios II para inicializar a operação ou o fim de um passo anterior. O estado *mult1* realiza a primeira operação de multiplicação

da esquerda para a direita na Equação (5.7). O estado *mult2* realiza a segunda operação de multiplicação. O estado *multadd* computa simultaneamente a terceira multiplicação e a adição entre o resultado da primeira e segunda multiplicação. Por fim, o estado *add* realiza a operação de adição entre os resultados da primeira adição e o da terceira multiplicação. De modo similar, as Equações (5.8) a (5.12) são computadas usando a mesma FSM.

Pode-se observar que as etapas dois e três da equação do *ganho* (ver equações (5.2) e (5.3)) são realizadas pelos mesmos passos da FSM e com a mesma operações em ponto flutuante tais como adições e multiplicações. Entretanto, a quarta etapa (ver equação (5.4)) representa uma operação de inversão de matriz, que é realizada como $adjunta(O^3)/determinante(O^3)$ (na qual usa-se uma operação de divisão). Portanto, um estado de divisão (*div1*) foi adicionado à FSM (ver Figura 5.3) para realizar a divisão entre a *adjunta*() e a *determinante*(). Vale lembrar que a determinante é calculada usando as operações de multiplicação e subtração que são realizadas no estado *multadd* e *add* (ver as Figuras (5.3) e (5.4)).

A Figura (5.4) mostra o escalonamento (em inglês *scheduling*) das operações na arquitetura, que compreende um módulo multiplicador, outro somador/subtrator e um divisor. Pode-se observar que todas as operações em ponto flutuante de um único elemento da matriz são realizados de forma sequencial (seis passos), mas os seis elementos da matriz são calculados ao mesmo tempo (em paralelo), replicando o mesmo caminho de dados (em inglês *datapath*) da arquitetura (ver Figura (5.5)).

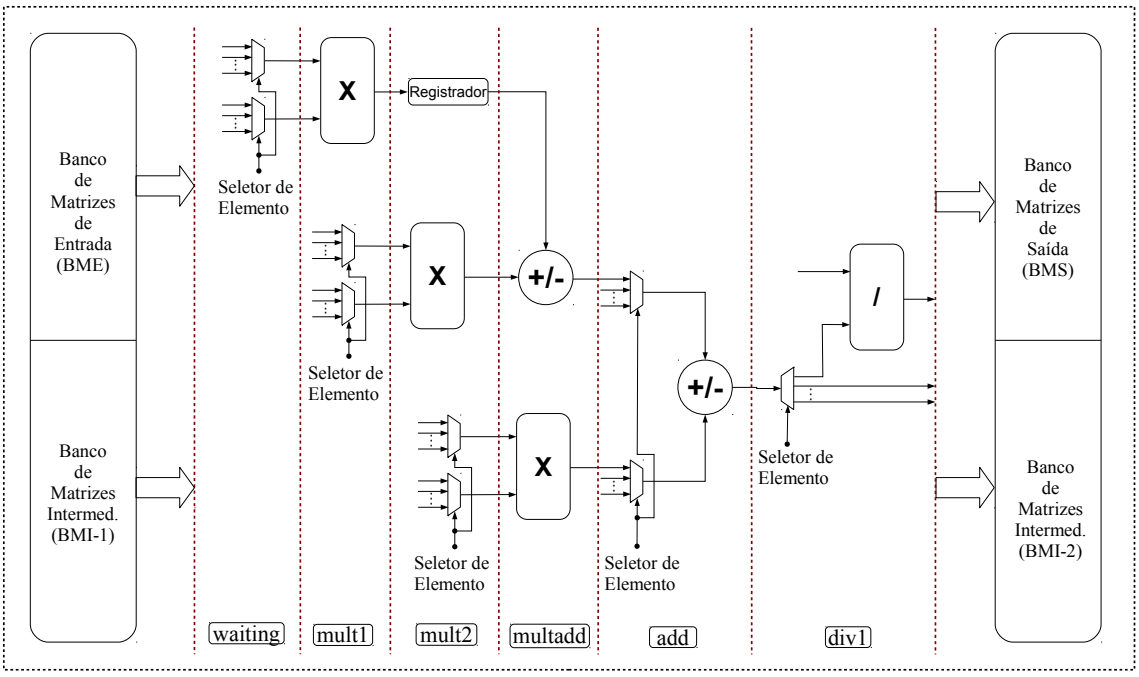


Figura 5.4. Escalonamento que gera o caminho de dados para computar a equação do *ganho*

A Figura (5.4) também mostra três bancos de registros da matrizes usadas na arquitetura

implementada (BME, BMI e BMS). Esses registros podem ser acessados a qualquer momento, para leitura e escrita de operações, a partir do processador Nios II através do barramento Avalon (ver Figura 5.1). Nesse caso, funções especiais no Nios II estão disponíveis para executar esses processos de leitura e escrita na matrizes. Depois de preencher o Banco das Matrizes de Entrada (BME), um comando de arranque (*start*) é enviado para a arquitetura, a fim de iniciar o processo. Sempre que o processo for concluído, os valores a partir do Banco das Matrizes de Saída (BMS) podem ser lido. A arquitetura também inclui registros intermediados, que são representados pelo Banco das Matrizes Intermediária (BMI). Na equação do *ganho*, o BME é composto por P_k^-, H^T e R_k, o BMS é composto por K_k e o BMI por $O^1_{3\times2}$, $O^2_{2\times2}$, $O^3_{2\times2}$ e $O^4_{2\times2}$.

Um importante detalhe no projeto refere-se ao caminho de dados. A Figura (5.5) também exibe os seis blocos de caminho de dados, cada um deles representando a computação de um único elemento de matriz. O número de caminho de dados é devido ao fato de a dimensão da maior matriz resultante durante o processo na equação do *ganho* ser 3×2. Também é importante ressaltar que os caminhos de dados 5 e 6 não contém um bloco de divisão, isso porque o número máximo de elementos na matriz de inversão é quatro, logo, somente os quatro caminhos de dados executam a operação de inversão de matriz.

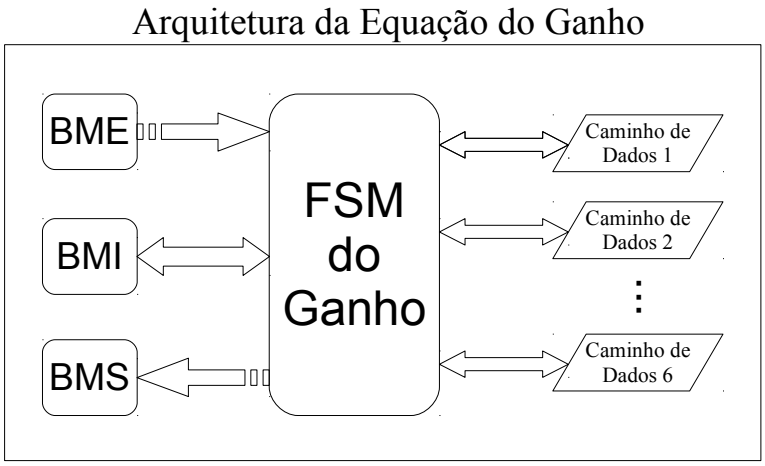


Figura 5.5. Arquitetura da equação do *ganho*

5.1.2 A Arquitetura dos *Estados*

A equação dos *estados* foi implementada decompondo a Equação (3.10) em três etapas, como visto nas Equações de (5.13) a (5.15). Para realizar isso, uma FSM de quatro estados (*waiting*, *mult1*, *mult2* e *add*) foi projetada. Como mostrado na Figura (5.6), a FSM está descrita na forma de máquina de *Mealy* (sequenciadores), onde a saída para cada transição é omitida para

simplificar a sua representação.

$$1^a etapa : O^1_{2\times1} = Y_k - h(\hat{X}^-_k) \tag{5.13}$$

$$2^a etapa : O^2_{3\times1} = K_k \cdot O^1_{2\times1} \tag{5.14}$$

$$3^a etapa : \hat{X}^+_k = \hat{X}^-_k + O^2_{3\times1} \tag{5.15}$$

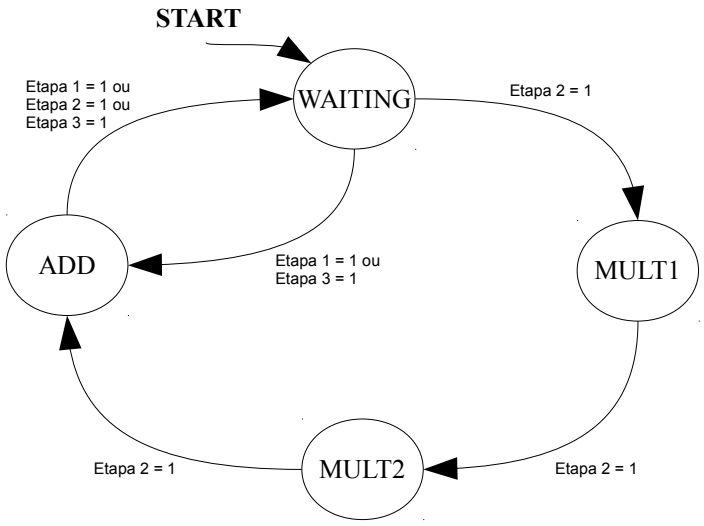


Figura 5.6. FSM usada na arquitetura para computar a equação dos *estados*

A Figura (5.7) mostra o caminho de dados para computar a equação dos *estados*. Nesse caso, o BME é composto por Y_k, $h()$, K_k e $\hat{X}^-_k$, o BMS é composto por $\hat{X}^+_k$ e o BMI é composto por $O^1_{2\times1}$ e $O^2_{3\times1}$.

A Figura (5.8) exibe a arquitetura da equação dos *estados*. Aqui, há somente três blocos de caminhos de dados, cada um deles representa o caminho de dados para computar um único elemento da matriz. Há somente três caminhos de dados, devido ao fato da dimensão máxima da matriz usada no processo de computação da equação dos *estados* ser 3×1.

5.1.3 A Arquitetura das *Covariâncias*

A equação das *covariâncias* foi implementada pela decomposição da Equação (3.11) em três etapas (ver Equações (5.16) a (5.18)). Para a realização desse cálculo em *hardware*, uma FSM de cinco estados foi implementada. A FSM pode ser vista na Figura (5.9) e é composta pelos estados: *waiting*, *mult1*, *mult2*, *multadd* e *add*. A FSM está descrita na forma de máquina

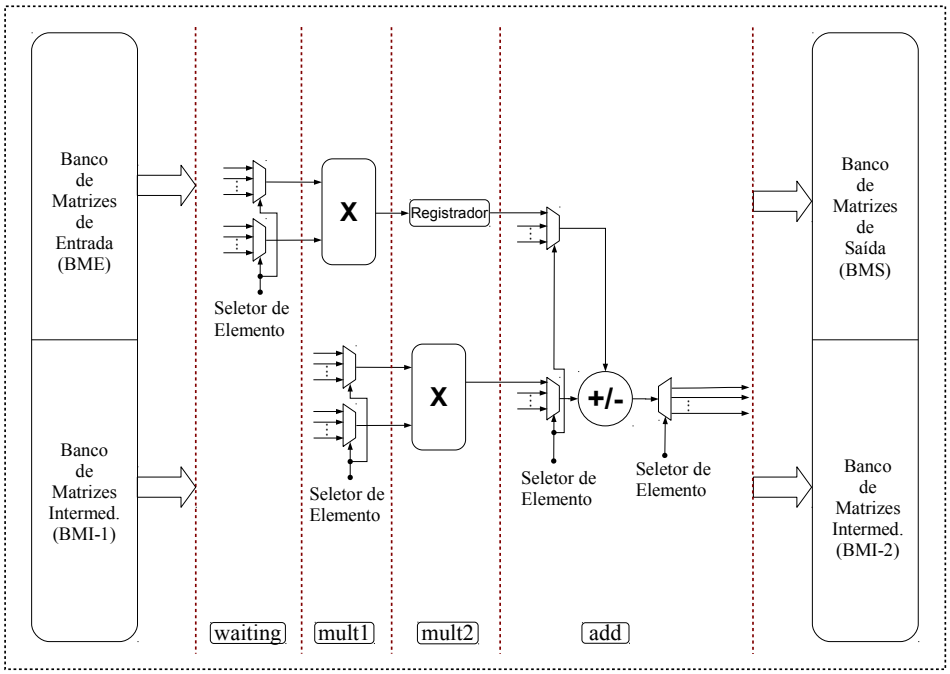


Figura 5.7. Escalonamento que gera o caminho de dados para computar a equação dos *estados*

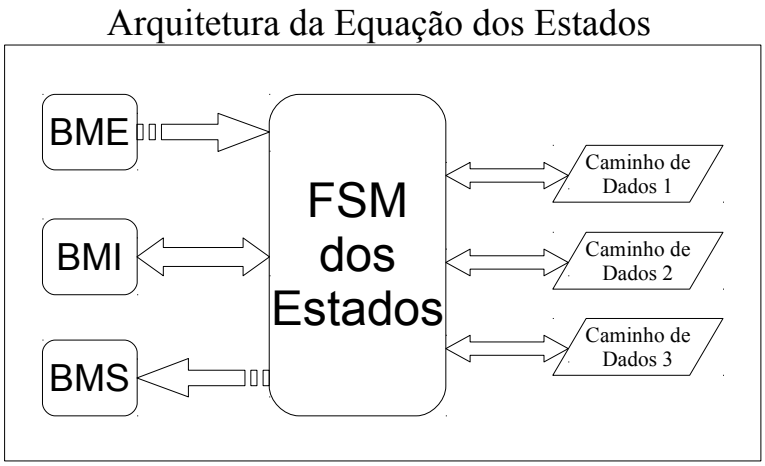


Figura 5.8. Arquitetura da equação dos *estados*

de *Mealy* (sequenciadores), onde a saída para cada transição é omitida para simplificar a sua representação.

$$1^a etapa : O^1_{3\times3} = K_k \cdot H \tag{5.16}$$

$$2^a etapa : O^2_{3\times3} = O^1_{3\times3} \cdot P^-_k \tag{5.17}$$

$$3^a etapa : P^+_k = P^-_k - O^2_{3\times3} \tag{5.18}$$

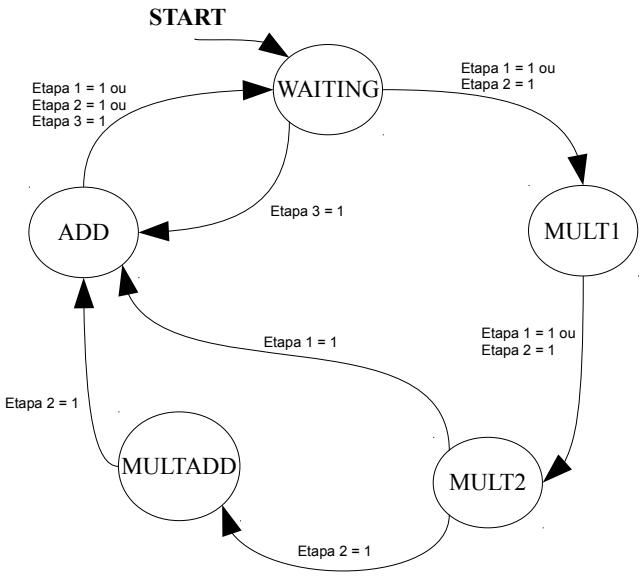


Figura 5.9. FSM usada na arquitetura para realizar a equação das *covariâncias*

A Figura (5.10) mostra o caminho de dados para a computação da equação das *covariâncias*. Nesse caso, o BME é composto por K_k, H e P_k^-, o BMS é composto por P_k^+ e o BMI é composto por $O^1_{3\times3}$ e $O^2_{3\times3}$.

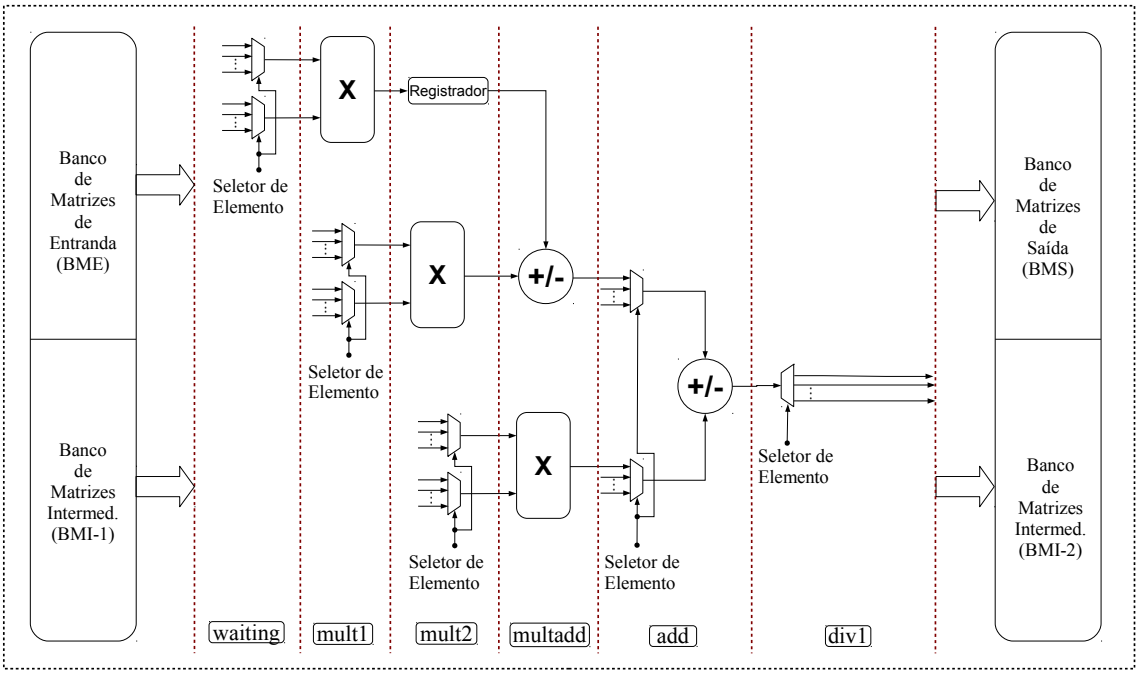


Figura 5.10. Escalonamento que gera o caminho de dados para computar a equação das *covariâncias*

Pode-se observar também que a Figura (5.11) mostra a arquitetura da *covariâncias*. Aqui, há nove blocos de caminhos de dados, cada um deles representa o caminho de dados para computar

um único elemento da matriz. Nesse caso, há nove caminhos de dados devido ao fato da dimensão máxima da matriz usada no processo de computação da equação das *covariâncias* ser 3×3.

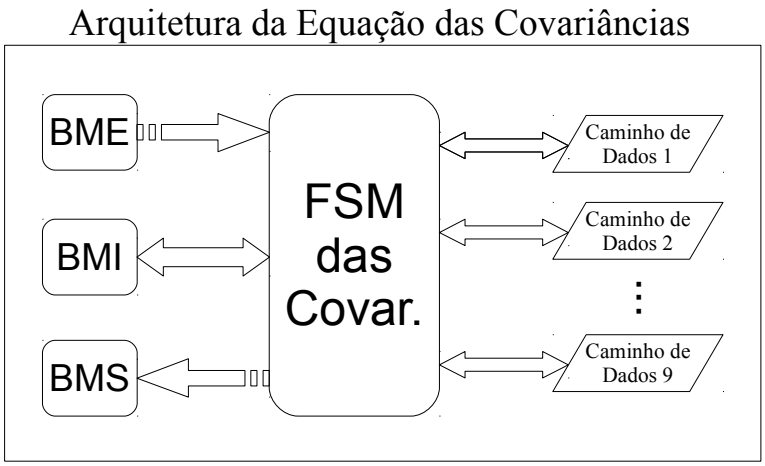


Figura 5.11. Arquitetura da equação das *covariâncias*

Por fim, a Tabela (5.1) apresenta o número de blocos em ponto flutuante requerido para a implementação de cada arquitetura.

Tabela 5.1. Recursos de blocos em ponto flutuante para a primeira abordagem

Arquitetura	Somador/Subtrator	Multiplicador	Divisor
Ganho	6	6	4
Estados	3	3	-
Covariâncias	9	9	-
Total	18	18	4

5.2 Segunda Abordagem para Solução em *Hardware* do Problema de Localização: Implementação em FPGA usando um Módulo Integrado para as três Equações

A segunda abordagem para a solução em *hardware* (módulo integrado) é mostrada na Figura (5.12), em que a comunicação entre o processador Nios II e arquitetura em *hardware* é simplificada. Essa abordagem também reduz os recursos de *hardwares* requeridos, em termos de blocos em ponto flutuante como multiplicadores, divisores e somadores/subtratores. Isso ocorre porque é possível compartilhar o mesmo caminho de dados para a implementação em *hardware* das três equações (*ganho*, *covariâncias* e *estados*), o que resulta na redução do número de operações

em ponto flutuante e, consequentemente, na área do FPGA. Vale ressaltar que, assim como na primeira abordagem, tanto a matriz Jacobiana de medição quanto a função não linear não estão incluídas no *hardware*; seus elementos são previamente calculados em *software* (usando o Nios II e a arquitetura CORDIC) e endereçado à arquitetura em *hardware*. O cálculo do seno e cosseno, neste caso, é realizado pela arquitetura CORDIC para acelerar o computo de H e $h()$.

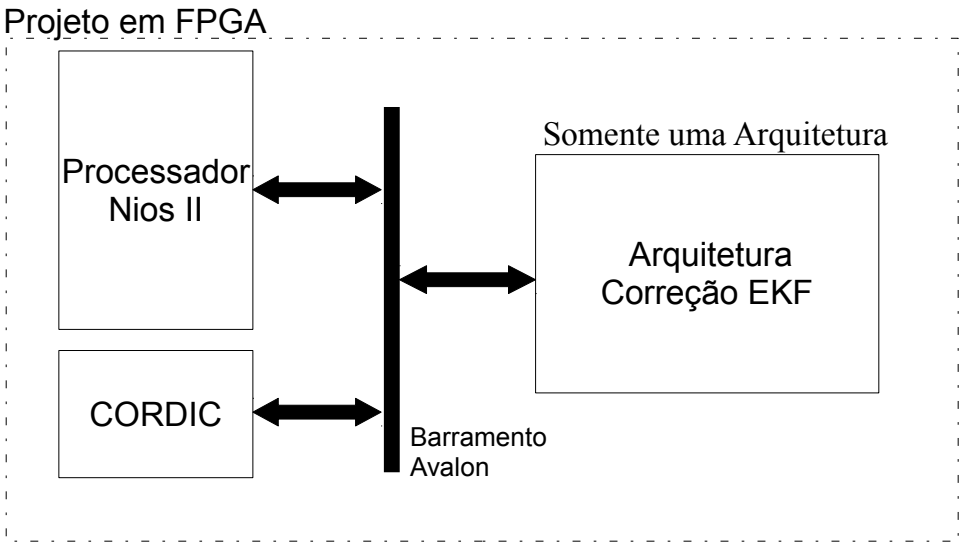


Figura 5.12. Segunda implementação em FPGA com somente uma arquitetura

A implementação do algoritmo de correção do EKF foi dividida em onze etapas, como mostradas nas Equações (5.19) a (5.29).

$$1^a etapa : O^1_{3\times2} = P^-_k \cdot H^T \tag{5.19}$$

$$2^a etapa : O^2_{2\times2} = H \cdot O^1_{3\times2} \tag{5.20}$$

$$3^a etapa : O^3_{2\times2} = O^2_{2\times2} + R_k \tag{5.21}$$

$$4^a etapa : O^4_{2\times2} = inv(O^3_{2\times2}) \tag{5.22}$$

$$5^a etapa : K_k = O^1_{3\times2} \cdot O^4_{2\times2} \tag{5.23}$$

$$6^a etapa : O^5_{2\times1} = Y_k - h(\hat{X}^-_k) \tag{5.24}$$

$$7^a etapa : O^6_{3\times1} = K_k \cdot O^5_{2\times1} \tag{5.25}$$

$$8^a etapa : \hat{X}^+_k = \hat{X}^-_k + O^6_{3\times1} \tag{5.26}$$

$$9^a etapa : O^7_{3\times3} = K_k \cdot H \tag{5.27}$$

$$10^a etapa : O^8_{3\times3} = O^7_{3\times3} \cdot P^-_k \tag{5.28}$$

$$11^a etapa : P^+_k = P^-_k - O^8_{3\times3} \tag{5.29}$$

A estrutura da FSM usada na arquitetura da equação de correção do EKF é exibida na Figura (5.13) e pode-se observar que ela é composta de seis estados, nomeados de *waiting*, *mult1*, *mult2*, *multadd*, *add* e *div1*. Já a Figura (5.14) exibe o caminho de dados para computar a FSM. Na arquitetura EKF, o BME é composto por P^-_k, H, R_k, Y_k, $h()$ e $\hat{X}^-_k$; o BMS é composto por K_k, $\hat{X}^-_k$ e P^+_k; e o BMI é composto por $O^1_{3\times2}$, $O^2_{2\times2}$, $O^3_{2\times2}$, $O^4_{2\times2}$, $O^5_{2\times1}$, $O^6_{3\times1}$, $O^7_{3\times3}$ e $O^8_{3\times3}$.

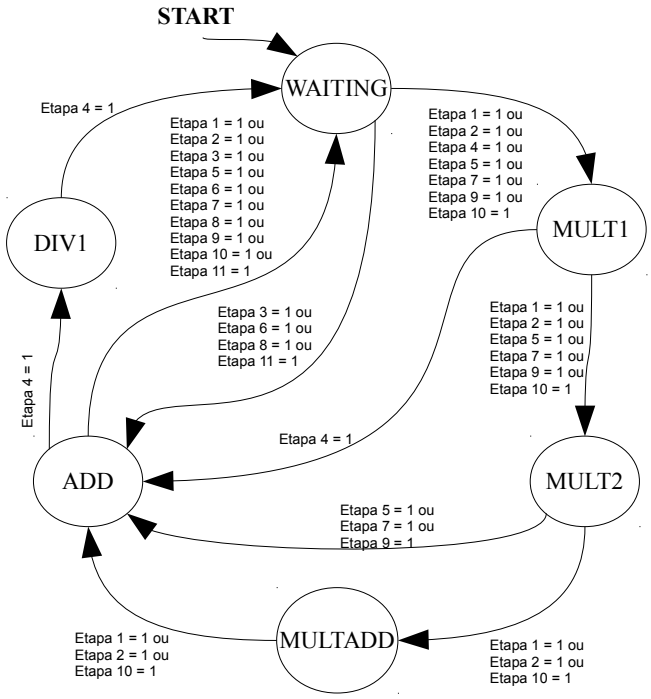


Figura 5.13. FSM usada na arquitetura para realizar o algoritmo de correção do EKF

Pode-se observar que a Figura (5.15) mostra toda a arquitetura do algoritmo de correção do EKF. Nesse caso, há nove blocos de caminhos de dados, cada um deles representando o caminho

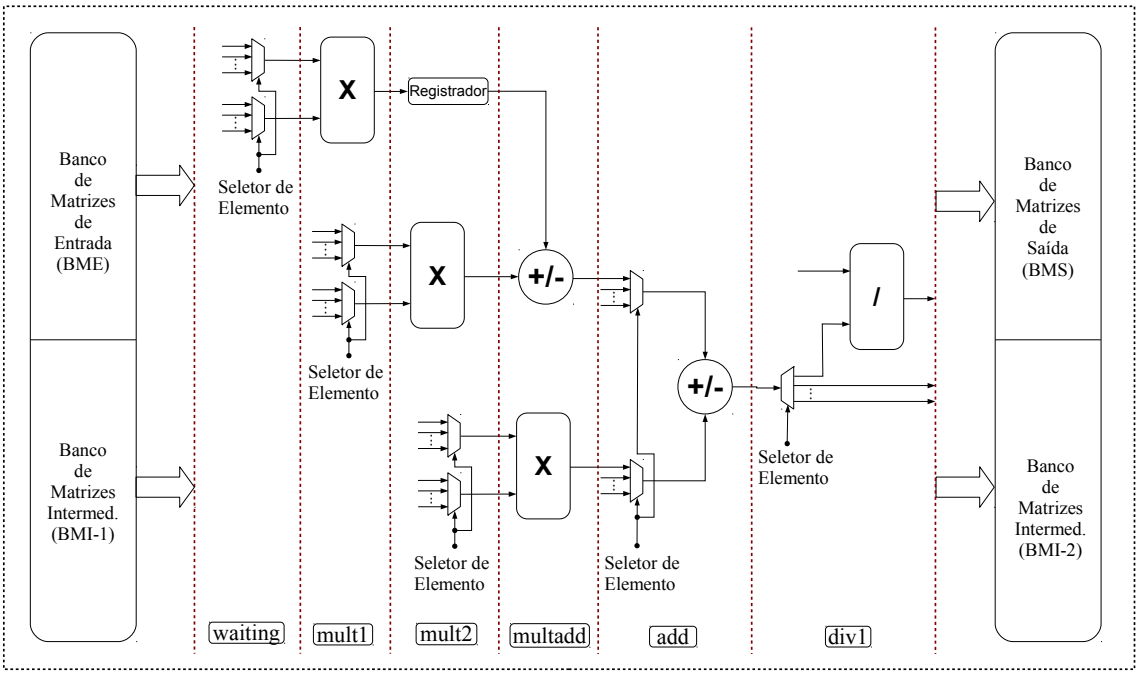


Figura 5.14. Escalonamento que gera o caminho de dados para computar o algoritmo de correção do EKF

de dados para calcular um elemento da matriz. Isso se deve ao fato de a maior dimensão matricial usada na computação do algoritmo de correção do EKF ser 3×3. Vale lembrar que os caminhos de dados 5 a 11 não contem blocos de divisão porque o número de elementos na inversão matricial do processo é quatro, logo, somente os caminhos de dados 1 a 4 possuem blocos de divisão.

Por fim, a Tabela (5.2) apresenta o número de blocos em ponto flutuante requerido pela implementação do módulo integrado.

Tabela 5.2. Recursos de blocos em ponto flutuante para a segunda abordagem

Arquitetura	Somador/Subtrator	Multiplicador	Divisor
Módulo Integrado	9	9	4

5.3 Conclusões do Capítulo

Neste capítulo apresentou-se duas abordagens de implementação das equações de correção de Kalman. Na primeira delas considerou-se uma arquitetura em *hardware* para cada equação do algoritmo de correção de Kalman, enquanto que na segunda abordagem foi apresentada uma única arquitetura que computa todo o algoritmo.

Para a primeira abordagem (arquiteturas independentes) podemos destacar sua modulari-

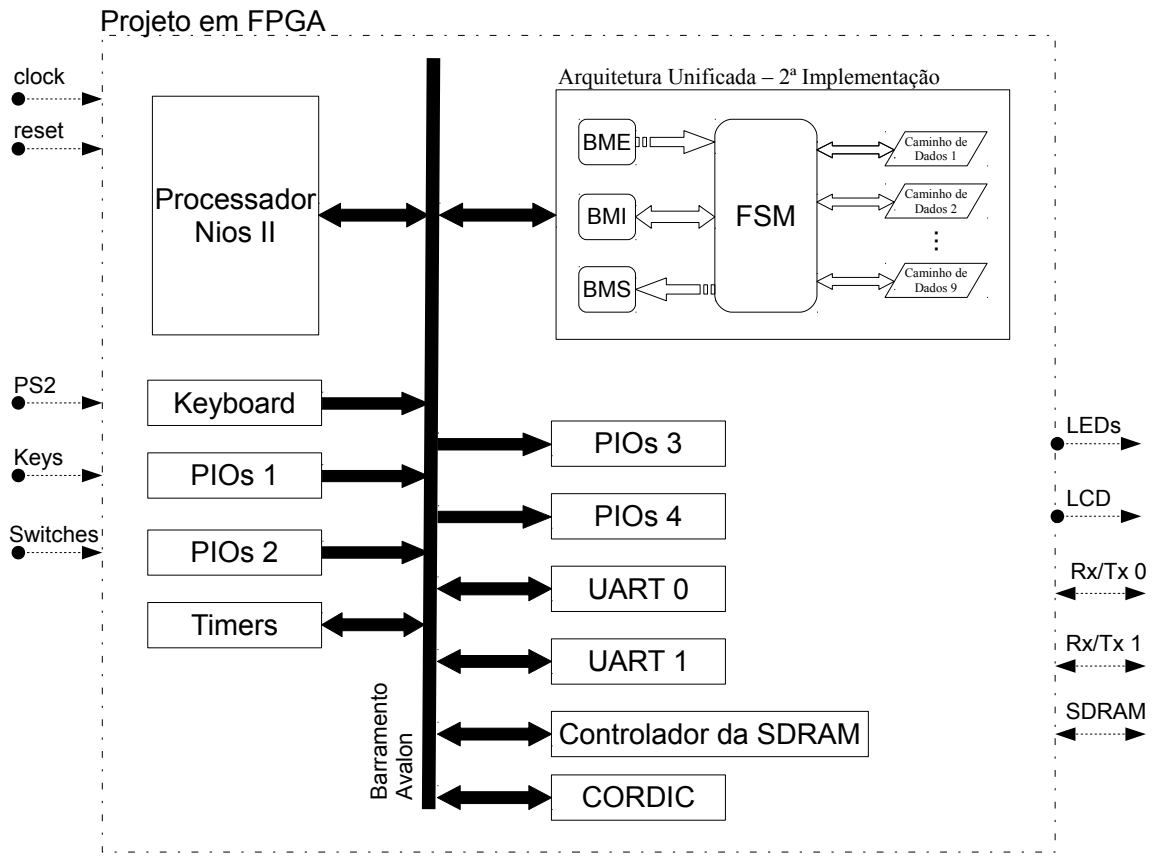


Figura 5.15. Projeto do FPGA com a arquitetura do algoritmo de correção do EKF

dade, ou seja, pode-se utilizar no projeto uma ou mais dessas arquiteturas. Isto é útil se visamos FPGAs de baixa densidade (elementos lógicos), uma parte do algoritmo é executada em *software* e a outra em *hardware*. Para a segunda abordagem (arquitetura integrada) podemos destacar a diminuição no consumo de *hardware* e desempenho, bem como no uso de unidades em ponto flutuante, conforme visto na Figura (5.16).

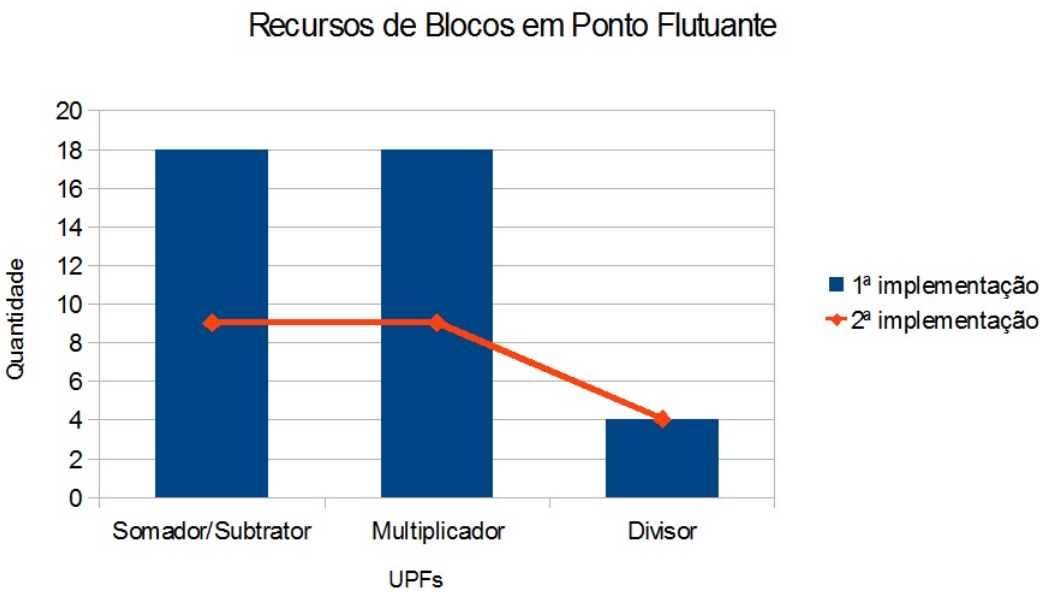


Figura 5.16. Comparativo de Consumo de UPFs entre a 1^a e a 2^a abordagem

6 Resultados da Implementação

Este capítulo apresenta os resultados experimentais obtidos através da primeira e segunda abordagens para a solução em *hardware* do problema de localização, levando em consideração recursos de *hardwares* e desempenho. As arquiteturas propostas foram desenvolvidas em linguagem VHDL (do inglês ***VHSIC Hardware Description Language***) e compiladas usando o *software* Quartus II [33]. As mesmas foram validadas no *kit* de desenvolvimento DE-115, que possui um FPGA de baixo custo Cyclone IV EP4CE115F29C7N [35].

6.1 Recursos de *Hardwares* e Consumo de Potência

Os resultados de recursos de *hardwares* para as arquiteturas propostas são descritas na Tabela 6.1 bem como o consumo de potência, na qual foi estimada pelo módulo PowerPlay Power Analyzer do *software* Quartus II.

Tabela 6.1. Recursos de *hardwares* para as arquiteturas EKF propostas

Arquitetura	LEs (*)	DSPs (*)	Fmáx (MHz)	Potência (mW)
Ganho (1ª implementação)	11336 (10%)	154 (29%)	49,07	170,87
Estados (1ª implementação)	4363 (4%)	21 (4%)	56,6	151,55
Covariâncias (1ª implem.)	12862 (11%)	63 (12%)	52,98	170,46
EKF** (2ª implementação)	22089 (19%)	175 (33%)	49,08	201,13

* Ocupação do *hardware*

** Somente fase de correção

Como pode ser observado, na primeira abordagem, a arquitetura que obteve melhor desempenho foi a equação dos *estados*, isso porque as operações e matrizes (dimensões) envolvidas são mais simples. Como consequência, menor consumo de *hardware* (4363 LEs) e maior frequência de operação (56,6 MHz). A de pior desempenho, para a 1ª abordagem, foi a arquitetura do *ganho*. Isso porque a arquitetura possui 4 unidades de divisão, aumentando o número de elementos lógicos (LEs) e o caminho entre a entrada e saída do sinal, fazendo com que a frequência máxima que a arquitetura possa rodar diminua (49,07 MHz), contra 52,98 MHz da arquitetura

das *covariâncias*.

Já para a segunda abordagem (arquitetura unificada), o número de elementos lógicos é maior (22089), porém se somarmos o número de LEs das arquiteturas da 1^a implementação (28561), temos uma redução de aproximadamente 23%. Isso é válido também para o consumo de DSPs (do inglês ***D**igital **S**ignal **P**rocessor*), a segunda implementação consome 175 DSPs enquanto que a soma das três arquiteturas da primeira abordagem dá 238 DSPs, ou seja, uma redução de 26%. A frequência máxima é praticamente a mesma da arquitetura do ganho e o consumo de potência 15% a mais.

Já, fazendo uma análise do consumo de potência, entre a primeira e a segunda abordagem, a última leva uma grande vantagem, pois a soma das potências das três arquiteturas (*ganho, estados e covariâncias*) é quase 60% maior do que a arquitetura unificada.

Por fim, a Tabela 6.2 exibe uma série de resultados referentes à compilação das arquiteturas apresentadas até o momento para diferentes tipos de FPGAs da Altera. Nesta tabela pode-se observar os resultados para consumo de elementos lógicos (LEs), DSPs e frequência máxima de operação da arquitetura. As compilações são direcionadas para os FPGAs Cyclone IV EP4CE22F17C6N (FPGA de baixo custo, disponível no *kit* de desenvolvimento DE0_Nano), Arria II GX EP2AGX260FF35I3 (FPGA de médio porte) e Stratix IV GX EP4SGX230KF40C2 (FPGA de alto desempenho).

6.2 Simulação Comportamental

As arquiteturas foram simuladas usando a ferramenta computacional ModelSim-Altera Start Edition [34]. A Tabela 6.3 exibe o desempenho em termos de ciclos de relógio para cada arquitetura. A simulação comportamental para as arquiteturas apresentadas no capítulo de metodologia (primeira e segunda abordagens) são mostradas na Figura 6.1 e os valores que aparecem estão na forma hexadecimal padrão IEEE 754. Pode ser observado que a arquitetura do *Ganho* gasta 980 nanosegundos (ns) para realizar a execução do algoritmo (20 nanosegundos $\times$ 49 ciclos de relógio), já a arquitetura dos *Estados* gasta 360ns, a arquitetura das *Covariâncias* 540ns e a arquitetura *Correção EKF* 1900ns. É importante lembrar que, enquanto a tarefa está sendo processada, o sinal *ready* está em nível alto. Quando o algoritmo é finalizado, o mesmo sinal vai para o nível baixo.

Tabela 6.2. Recursos de *hardwares* para as arquiteturas EKF propostas em outras famílias de FPGA Altera

Arquitetura	FPGA	LEs (*)	DSP (*)	Fmáx (MHz)
Multiplicação	DE0_Nano	139 (1%)	7 (5%)	82,97
	Arria II GX	73 (< 1%)	4 (< 1%)	112,31
	Stratix IV GX	73 (< 1%)	4 (< 1%)	150,13
Soma/Subtração	DE0_Nano	940 (4%)	0	53,47
	Arria II GX	719 (< 1%)	0	66,16
	Stratix IV GX	719 (< 1%)	0	72,41
Divisão	DE0_Nano	589 (3%)	28 (21%)	53,05
	Arria II GX	333 (< 1%)	16 (2%)	72,54
	Stratix IV GX	333 (< 1%)	16 (1%)	89,32
CORDIC	DE0_Nano	3777 (17%)	7 (5%)	43,5
	Arria II GX	2844 (1%)	4 (1%)	49,39
	Stratix IV GX	2842 (2%)	4 (< 1%)	55,82
Ganho	DE0_Nano	12598 (56%)	132 (100%)	58,24
	Arria II GX	9352 (5%)	88 (12%)	64,22
	Stratix IV GX	9519 (5%)	88 (7%)	80,63
Estados	DE0_Nano	4383 (20%)	21 (16%)	63,87
	Arria II GX	3522 (2%)	12 (2%)	79,05
	Stratix IV GX	3566 (2%)	12 (1%)	86,31
Covariância	DE0_Nano	13007 (58%)	63 (48%)	65,3
	Arria II GX	11795 (6%)	36 (5%)	68,99
	Stratix IV GX	11858 (7%)	36 (3%)	75,09
EKF	DE0_Nano	25461 (114%)	132 (100%)	-
	Arria II GX	16505 (8%)	100 (14%)	62,64
	Stratix IV GX	16685 (9%)	100 (8%)	64,03

Tabela 6.3. Ciclos de relógio das arquiteturas propostas

Implementação	Arquitetura	Ciclo de Relógio
1^a	*Ganho*	49
1^a	*Estados*	18
1^a	*Covariâncias*	27
2^a	*EKF**	95

* Somente algoritmo de correção

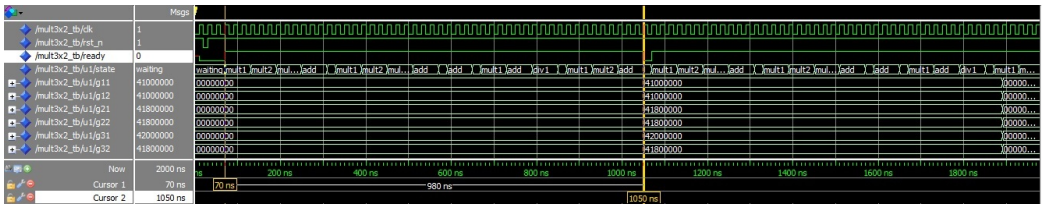
6.3 Desempenho da Arquitetura EKF (2^a Implementação)

O algoritmo de correção do EKF (implementação do módulo integrado) foi executada em cinco plataformas diferentes e seus resultados foram comparados entre si. Primeiramente, o algoritmo foi rodado no processador embarcado Nios II [36], posteriormente, o mesmo algoritmo foi executado tanto nos ambientes Matlab® e Dev-C++[37] (usando um processador Intel Core 2 Duo 2.66GHz, 65W). Adicionalmente, o desempenho dos dados também foram avaliados rodando o mesmo algoritmo no PC embarcado do robô Pioneer 3AT, cuja a CPU (do inglês ***C**entral **P**rocessing **U**nit*) é um processador Intel Pentium M 1.80GHz, 21W.

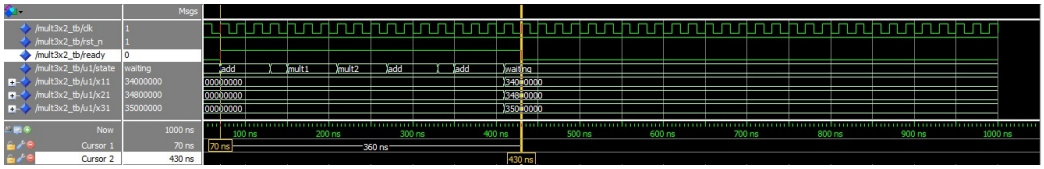
O desempenho dos dados obtidos nas plataformas anteriores foram comparados com o desempenho da arquitetura implementada em *hardware* do módulo integrado. Os resultados desses desempenhos podem ser vistos na Tabela 6.4, cujo o tempo de execução está expresso em microssegundos (us). Nesse caso, tanto a implementação em Nios II quanto em *hardware* rodaram com um relógio de 50MHz. Vale ressaltar que o algoritmo das plataformas Dev-C++ e PC embarcado do P3-AT fez o uso da biblioteca para C/C++ chamada GMATRIX [38].

Tabela 6.4. Comparação do tempo de execução do algoritmo de correção entre as plataformas

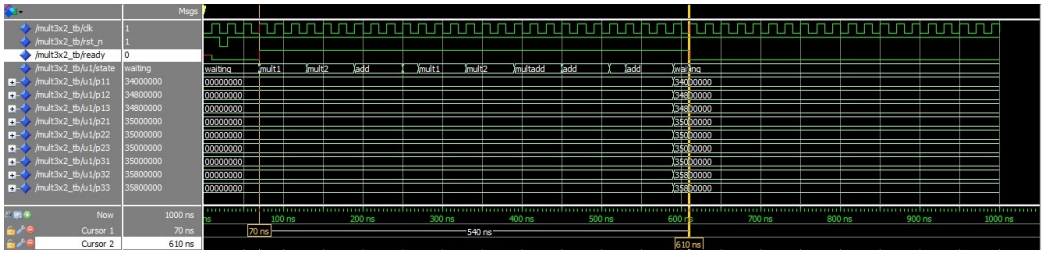
Plataformas	Tempo de Execução (us)
Processador Nios II	577
Implementação em *Hardware*	2,08
Nios II + Implementação em *Hardware*	68
Matlab®	149
Dev-C++	22
PC embarcado do Pioneer 3AT	21



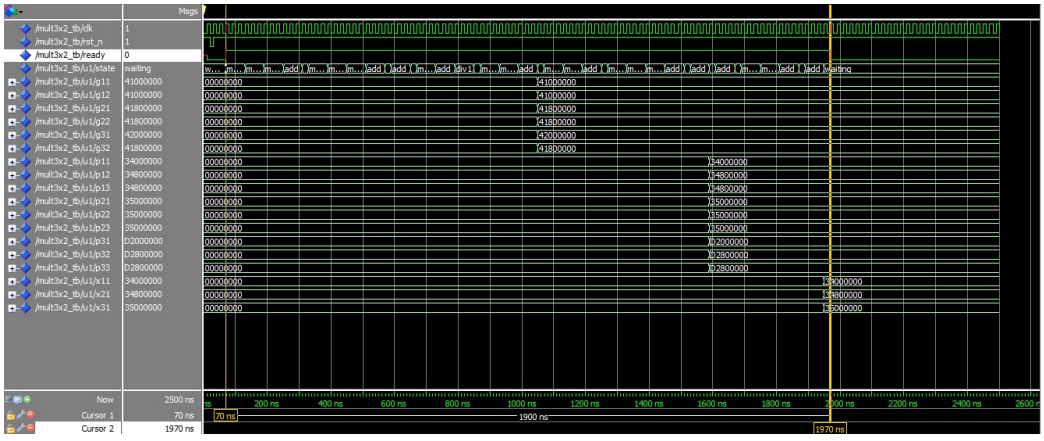
(a)



(b)



(c)



(d)

Figura 6.1. Resultado da simulação comportamental para a arquitetura (a) do *Ganho* (b) dos *Estados* (c) das *Covariâncias* e (d) *Correção EKF*

6.4 Validação da Arquitetura EKF (2^a Implementação)

Um cenário foi preparado para a implementação da tarefa de validação da arquitetura do módulo integrado. Um robô, baseado na plataforma Pioneer 3AT, também foi usado, bem como os dados dos seus sensores de ultrassom. Um sensor ladar também foi usado como medidor de distância, ajudando assim, a diminuir a matriz de covariâncias. Os dados vindos desses sensores

foram usados para estimar a pose do robô ($\hat{X}_k^+$). Nesse caso, a posição atual do robô é conhecida bem como as medidas do cenário. Pode-se observar na Figura 6.2 o ambiente de teste e a posição do robô.

Figura 6.2. Cenário para a validação das arquiteturas EKF

Após a realização dos testes, os resultados foram plotados em um gráfico juntamente com os resultados obtidos da simulação feita no programa Matlab®. A estimação de cada variável de estado está mostrada na Figura 6.3. Pode ser observado que ambas as estimações (feitas pelo Matlab® e arquitetura em *hardware*) são muito parecidas. Ambas convergem para a posição real do robô na *coordenada global*, que é zero para x, y e θ.

Para exemplificar visualmente o funcionamento da fusão sensorial, pegou-se os resultados obtidos através da aplicação em *hardware* e graficou-se os resultados desta fusão em x, y e θ (através das distribuições Gaussianas, para um dado instante de tempo), como visto na Figura (6.4). Como era de se esperar, a distribuição da fusão sensorial (u) está mais próxima do sensor de menor desvio padrão (Z_2, que nesse caso é o ladar).

6.5 Conclusões do Capítulo

Neste capítulo foram visto os diferentes resultados para as arquiteturas propostas de implementação do algoritmo de correção do EKF. Primeiramente, mostrou-se os resultados de recurso de *hardware* e consumo de potência. Viu-se para este resultado os recursos consumidos em termos de elementos lógicos, DSPs, potência e frequência máxima de operação das arquiteturas. Para a primeira implementação, a vantagem está na modularização das equações, ao trabalhar com FPGAs de baixa densidade, pode-se implementar em *hardware* uma ou duas arquiteturas enquanto que a outra implementa-se em *software*.

Foi mostrado também os resultados da simulação comportamental de tais arquiteturas. Verificou-se que a arquitetura dos *Estados* tem um menor tempo de simulação, enquanto que a

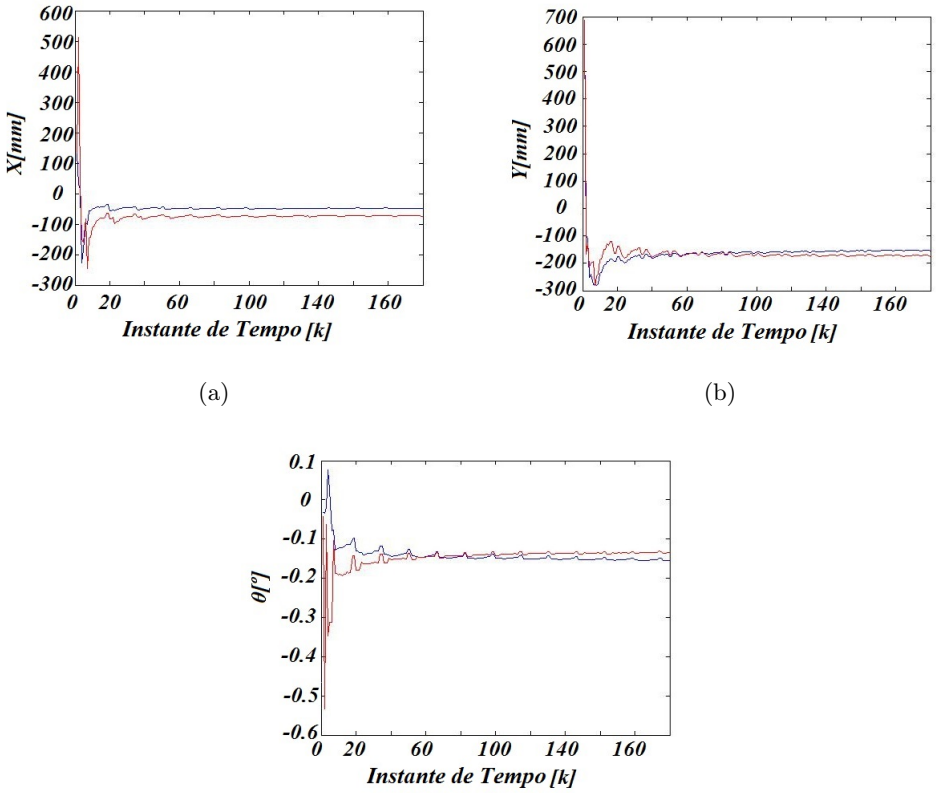
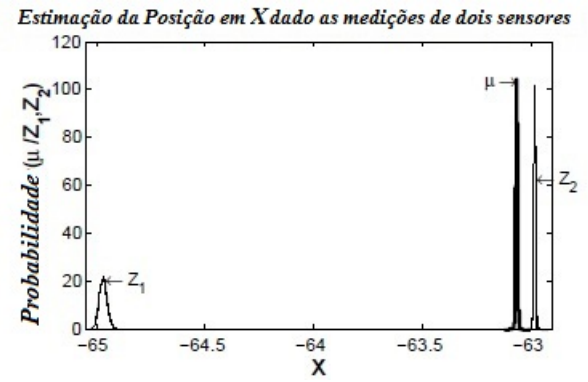


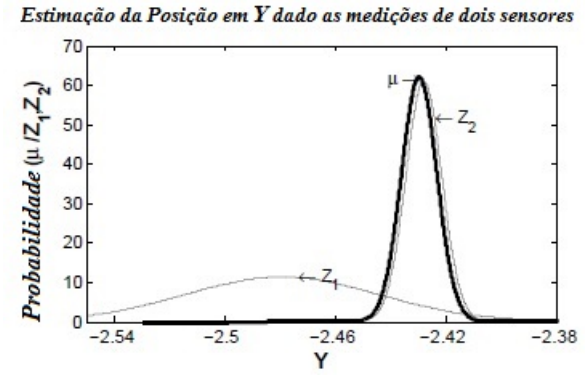
Figura 6.3. Estimação da pose do robô pelo Matlab® (linha azul) e arquitetura EKF (linha vermelha) para a mesma aquisição de dados em (a) X, (b) Y e (c) θ

arquitetura do *Ganho* tem o maior tempo. Esse resultado nos diz que a arquitetura do Ganho é uma forte candidata para implementar em *hardware*, já que o seu tempo de simulação é maior.

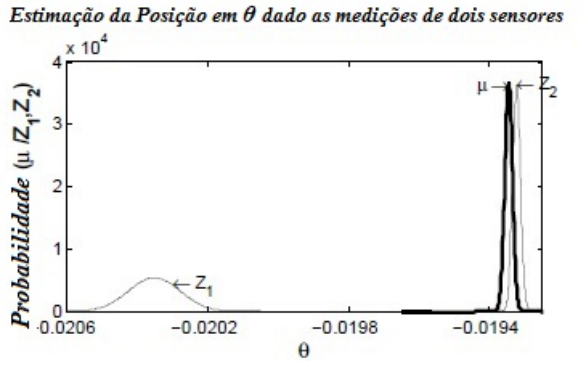
Por fim, foi visto com relação à segunda implementação o seu desempenho (tempo de execução) e uma implementação em um caso real. Verificou-se que os resultados da arquitetura são tão bons quanto os resultados obtidos em *software*. Falando de desempenho, vê-se um ganho em torno de 90% da arquitetura em *hardware* em relação ao algoritmo executado no PC do Pioneer, mas em uma aplicação conjunta *software* (Nios II) e *hardware*, esse tempo sobe quase 3,24 vezes, isso deve-se ao tempo de escrita e leitura na arquitetura em *hardware* via o barramento Avalon.



(a)



(b)



(c)

Figura 6.4. Resultado para a fusão de dados (μ) dos sensores sonar (Z_1) e ladar (Z_2) em (a) X, (b) Y e (c) θ (aplicação em *hardware*)

7 Conclusões e Trabalhos Futuros

7.1 Comentários Finais e Conclusões

Neste trabalho foram apresentadas duas propostas de arquitetura em *hardware* (baseada em FPGA) para resolver o problema de localização de robôs móveis. Primeiramente, foi realizada uma revisão da teoria envolvida no projeto bem como um aprofundamento na localização EKF (algoritmo usado para implementação). Na sequencia, foram apresentadas as UPFs (Unidades em Ponto Flutuante), arquiteturas usadas para realizar as operações algébricas em ponto flutuante. Em seguida, no capítulo de metologia, foram mostradas as duas formas de implementação do algoritmo de correção EKF. Na primeira abordagem, criou-se uma arquitetura para cada equação do algoritmo, enquanto que na segunda, implementou-se uma única arquitetura. Também foi visto neste trabalho o consumo de recursos de *hardwares* e potência, bem como a performance (desempenho) das implementações.

Para provar o funcionamento apropriado da arquitetura, uma tarefa de validação foi realizada. Os resultados apontaram para um desempenho satisfatório do módulo EKF (referente a etapa de correção), com pouco consumo de *hardware* que é consequência direta da versão sequencial do algoritmo. A comunicação de dados entre o processador Nios II e a arquitetura EKF é a responsável direta pelo decaimento da performance como um todo.

Verificou-se que as arquiteturas trabalham tão bem quanto a versão em *software*; entretanto, se levarmos em consideração que estamos trabalhando com uma tecnologia de baixíssimo consumo de potência e pequena área de silicone, concluímos que as arquiteturas propostas são perfeitamente aceitas para aplicações reais.

Para o melhor de nosso conhecimento, não há outros trabalhos que propõem este tipo de abordagem de *hardware* para o algoritmo EKF, analisando seus impactos em diferentes aspectos, sendo endereçados para pequenos robôs com sérias restrições no consumo de energia, e usando pequenos dispositivos com relógios de baixa frequência.

7.2 Propostas de Trabalhos Futuros

Como trabalhos futuros sugere-se:

- A implementação em *hardware* das equações de predição do algoritmo EKF;
- A implementação do periférico *DMA (do inglês **D**irect **M**emory **A**ccess)* para transferência de dados entre *hardware* e *software*, assim espera-se uma diminuição no tempo de execução da implementação HW/SW;
- A implementação do algoritmo EKF em *hardware* usando arranjos sistólicos para comparar desempenho, consumo de *hardware* e potência;
- A implementação da estimativa não somente da pose do robô, mas do mapa do ambiente, assim abordaria a questão do SLAM;
- Estudo e aplicação do SLAM Desacoplado.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] SIEGWART, R.; NOURBAKHSH, I. R. *Introduction to Autonomous Mobile Robots*. Cambridge, MA, USA: The MIT Press, 2004.

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

[3] nOZ, D. M. M. *Otimização por inteligência de exames usando arquiteturas paralelas para aplicações embarcadas*. Tese (Doutorado) — Universidade de Brasília, Brazil, 2012.

[4] NOZ, D. M. M. et al. FPGA-based floating-point library for CORDIC algorithms. In: *Proc. Int. Southern Programmable Logic Conf.* Porto de Galinhas, Brazil: [s.n.], 2010. p. 55–60.

[5] HARTENSTEIN, R. Why we need reconfigurable computing education. In: *1st International Workshop on Reconfigurable Computing Education*. Karlsruhe, Germany: [s.n.], 2006. Disponível em: <http://fpl.org/RCeducation/>.

[6] SASS, R.; SCHMIDT, A. *Embedded Systems Design with Platform FPGAs: Principles and Practices*. Elsevier Science, 2010. (Titolo collana). ISBN 9780123743336. Disponível em: <http://books.google.com.br/books?id=Ki7zs-Ex2d0C>.

[7] PTC. Systems engineering: Development of mechatronics and software need to be integrated closely. In: . [S.l.]: Parametric Technology Corporation (PTC), 2012.

[8] BARROS, E. et al. *Hardware/software co-design: projetando hardware e software concorrentemente*. IME-USP, 2000. Disponível em: <http://books.google.com.br/books?id=IvtQHAAACAAJ>.

[9] WOF, W. *High-Performance Embedded Computing, Architectures, Applications, and Methodologies*. [S.l.]: Elsevier Science, 2007.

[10] GAJSKI, D. et al. *Embedded System Design: Modeling, Synthesis and Verification*. New York, NY, USA: Springer, 2009. ISBN 9781441905031. Disponível em: <http://books.google.com.br/books?id=g6MbqAAACAAJ>.

[11] BONATO, V. *Proposal of an FPGA hardware architecture for SLAM using multi-cameras and applied to mobile robotics*. Tese (Doutorado) — Institute of Computer Science and Computational Mathematics - University of São Paulo, São Carlos, SP, Brazil, January 2008.

[12] ADEPTMOBILEROBOTS. *"P3-AT Datasheet"*. 2012. Disponível em: <http://www.mobilerobots.com/Libraries/Downloads>.

[13] KAM, M.; ZHU, X.; KALATA, P. Sensor fusion for mobile robot navigation. *Proceedings of the IEEE*, v. 85, n. 1, p. 108 – 119, jan. 1997. ISSN 0018-9219.

[14] THRUN, S.; BURGARD, W.; FOX, D. *Probabilistic Robotics*. Cambridge, MA, USA: The MIT Press, 2005.

[15] BONATO, V.; MARQUES, E.; CONSTANTINIDES, G. A floating-point extended kalman filter implementation for autonomous mobile robots. In: *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*. [S.l.: s.n.], 2007. p. 576 –579.

[16] BONATO, V. et al. An fpga implementation for a kalman filter with application to mobile robotics. In: *Industrial Embedded Systems, 2007. SIES '07. International Symposium on*. [S.l.: s.n.], 2007. p. 148 –155.

[17] DELLAERT, F. et al. Monte carlo localization for mobile robots. In: *Robotics and Automation, 1999. Proceedings. 1999 IEEE International Conference on*. [S.l.: s.n.], 1999. v. 2, p. 1322 –1328 vol.2. ISSN 1050-4729.

[18] COUTO, L. N. *Sistema para localização roboótica de veículos autônomos baseado em visão computacional por pontos de referência*. Dissertação (Mestrado) — Instituto de Ciências Matemáticas e de Computação - Universidade de São Paulo, São Carlos, SP, Brasil, Julho 2012.

[19] KALMAN, R. E. A new approach to linear filtering and prediction problems. *Journal of Basic Engineering*, v. 82 (Series D), p. 35–45, 1960.

[20] MINGAS, G.; TSARDOULIAS, E.; PETROU, L. An fpga implementation of the smg-slam algorithm. *Microprocessors and Microsystems*, v. 36, n. 3, p. 190 – 204, 2012. ISSN 0141-9331. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0141933111001244>.

[21] GHORBEL, A. et al. A hw/sw implementation on fpga of a robot localization algorithm. In: *Systems, Signals and Devices (SSD), 2012 9th International Multi-Conference on*. [S.l.: s.n.], 2012. p. 1 –7.

[22] LIU, Y.; BOUGANIS, C.-S.; CHEUNG, P. Efficient mapping of a kalman filter into an fpga using taylor expansion. In: *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*. [S.l.: s.n.], 2007. p. 345 –350.

[23] BIERMAN, G. J. *Factorization methods for discrete sequential estimation*. [S.l.]: Academic Press, 1977.

[24] THORNTON, C. L. *Triangular covariance factorizations for kalman filtering*. Tese (Doutorado) — School of Engineering - University of California, 1976.

[25] CHEN, G.; GUO, L. The fpga implementation of kalman filter. In: *Proceedings of the 5th WSEAS International Conference on Signal Processing, Computational Geometry & Artificial Vision*. Stevens Point, Wisconsin, USA: World Scientific and Engineering Academy and Society (WSEAS), 2005. (ISCGAV'05), p. 61–65. ISBN 960-8457-35-1. Disponível em: <http://dl.acm.org/citation.cfm?id=1369774.1369784>.

[26] NASH, J. G.; HANSEN, S. Modified faddeeva algorithm for matrix manipulation. In: *SPIE, Conference on*. San Diego, CA, USA: [s.n.], 1984.

[27] AGUIRRE, L. A. *Introdução à Identificação de Sistemas: Técnicas Lineares e Não-Lineares Aplicadas a Sistemas Reais*. 3rd. ed. Belo Horizonte, Brasil: Editora UFMG, 2007.

[28] ALTERA. *"What is an FPGA?"*. 2012. Disponível em: <http://www.altera.com>.

[29] VALLEJO, M. L. L.; RODRIGO, J. L. A. *FPGA: Nociones básicas y Implementación*. [S.l.], April 2004.

[30] SáNCHEZ, D. F. et al. Parameterizable floating-point library for arithmetic operations in fpgas. In: *Proceedings of the 22nd Annual Symposium on Integrated Circuits and System Design: Chip on the Dunes*. New York, NY, USA: ACM, 2009. (SBCCI '09), p. 40:1–40:6. ISBN 978-1-60558-705-9. Disponível em: <http://doi.acm.org/10.1145/1601896.1601948>.

[31] NOZ, D. M. M. et al. Tradeoff of FPGA design of a floating-point library for arithmetic operators. *Journal of Integrated Circuits and Systems*, v. 5, n. 1, p. 42–52, 2010.

[32] MARKSTEIN, P. Software division and square root using goldschmidt's algorithms. In: *Proc. Conference on Real Nambers ans Computers*. Dagstuhl, Germany: RNC, 2004. p. 146–157.

[33] ALTERA. *"Quartus II Handbook"*. 2012. Disponível em: <http://www.altera.com/literature/>.

[34] ALTERA. *"ModelSim-Altera Start Edition"*. 2012. Disponível em: <www.altera.com/literature/>.

[35] TERASIC. *"Altera DE-115 Development and Education Board"*. 2012. Disponível em: <http://www.terasic.com.tw>.

[36] ALTERA. *"Nios II Processor Reference Handbook"*. 2012. Disponível em: <www.altera.com/literature/>.

[37] BLOODSHED. *Cpp*. 2012. Disponível em: <http://www.bloodshed.net/devcpp.html>.

[38] BORGES, G. A. *GMATRIX: Uma bilioteca matricial para C/C++*. Brasilia-DF, Outubro 2005. Disponível em: <http://lara.unb.br/ gaborges/recursos/programacao/gmatrix/gmatrixdoc.pdf>.

Anexos

A. AMBIENTE DE DESENVOLVIMENTO

A.1 Kit de Desenvolvimento DE2-115

Todas as implementações utilizaram como base um *kit* de desenvolvimento para FPGAs da Terasic Inc. Esse *kit* utiliza um FPGA Cyclone IV E da Altera Corp., fornecendo uma grande variedade de interfaces de entrada e saída para diversos periféricos. As especificações do *kit* são as que seguem (Figura A.1):

- FPGA Altera Cyclone IV EP4CE115
- Dispositivo de configuração Altera EPCS64
- USB Blaster para programação e controle de dispositivos
- 4 *pushbuttons*
- 18 *switches*
- 18 leds vermelhos
- 9 leds verdes
- 8 Displays de 7 segmentos
- Osciladores de 50MHz
- Interface de entrada e saída de áudio
- Porta de saída VGA com conversor DA de 8 bits
- Decodificador NTSC/PAL, para entrada de vídeo
- Controlador Ethernet 10/100 Mbits/s
- Controlador USB Host/Slave
- Interface serial RS-232
- Interface PS2
- Interface infravermelho
- Uma porta de 40 pinos de entrada/saída de uso geral

- Um conector HSMC (do inglês ***H**igh **S**peed **M**ezzanine **C**ard*)
- Memória SRAM de 2MB (1Mx16)
- Memória SDRAM de 128MB (32Mx32bit)
- Memória Flash de 8MB (4Mx16)
- Memória EEPROM de 32Kbit

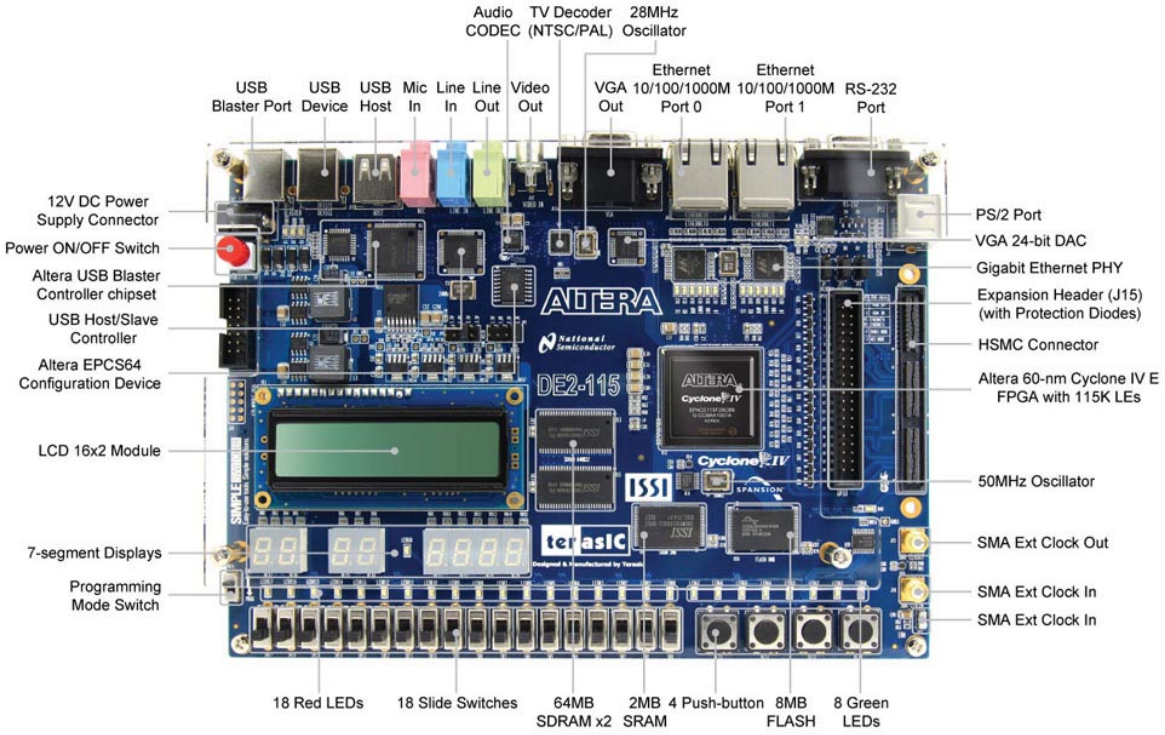


Figura A.1. Kit de desenvolvimento DE2-115, Terasic Inc.

A.2 *Software* Quartus II da Altera Corp.

Foi utilizada a versão 10.1 da ferramenta EDA Quartus II, da Altera Corp., a qual permite a criação e simulação de projetos, além da síntese e programação dos dispositivos FPGAs. As Figura A.2 mostra uma captura de do ambiente de programa.

O Quartus II suporta o desenvolvimento em três linguagens diferentes de descrição de *hardware*:

- VHDL - *Very High Speed Integrated Circuits Hardware Description Language*. É uma linguagem padrão ISO para a descrição de *hardware*.

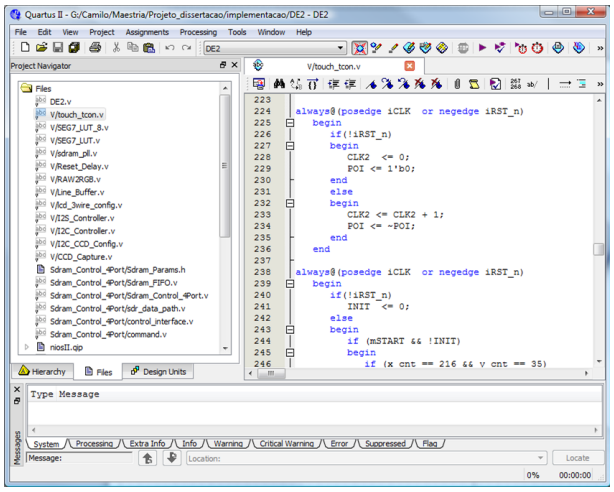


Figura A.2. Tela do Quartus II, mostrando um exemplo de código Verilog.

- Verilog *Hardware Description Language*. Linguagem desenvolvida pela *Cadence Systems*, empresa especializada em ferramentas CAD para projetos de circuitos.
- AHDL - *Altera Hardware Description Language*. Linguagem desenvolvida pela própria Altera.

Existe ainda a possibilidade de se criar projetos utilizando diagramas de blocos representativos dos códigos descritos nas HDLs. A Figura A.3 mostra um exemplo no Quartus II.

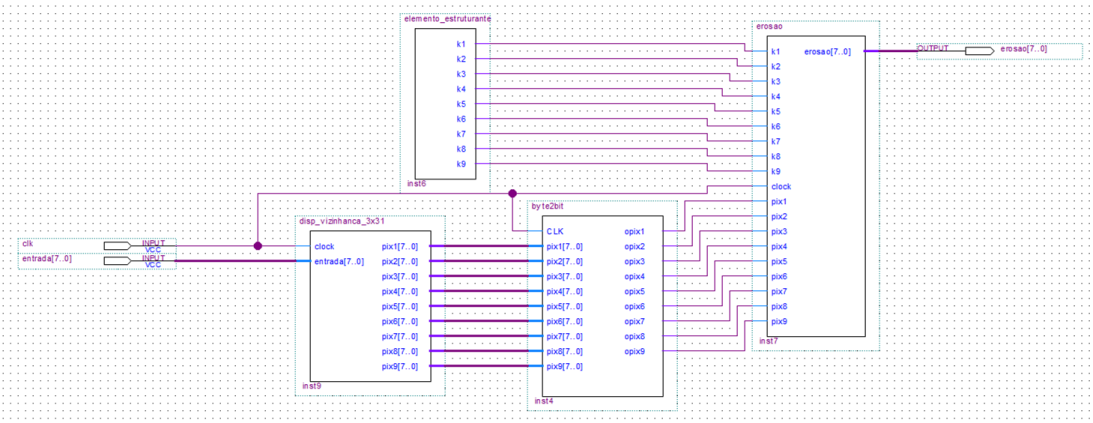


Figura A.3. Tela do Quartus II, exemplificando um projeto utilizando Diagramas de Blocos.

A.3 Processador Nios II da Altera Corp.

O processador Nios II é equivalente a um microcontrolador que inclui um processador e uma variedade de periféricos e memórias em um chip só. Na Figura A.4 mostra-se um exemplo de

um processador Nios II. O Nios II é um processador RISC de proposito geral com as seguintes características:

- Conjunto de instruções completo de 32 bits
- 32 registradores de proposito geral
- 32 fontes de interrupção
- Multiplicadores e divisores 32×32 que produzem resultados de 32 bits.
- Instruções dedicadas para o cômputo de produtos de multiplicações de 64 e 128 bits
- Instruções de ponto flutuante para operações de precisão simples
- Acesso a uma grande variedade de periféricos no chip e interfaces para memórias e periféricos externos.
- Ambiente de desenvolvimento *software* baseado na ferramenta GNU C/C++ e o *Nios II Software Build Tools (SBT) for Eclipse* (Figura A.5).
- Desempenho acima de 250 DMIPS

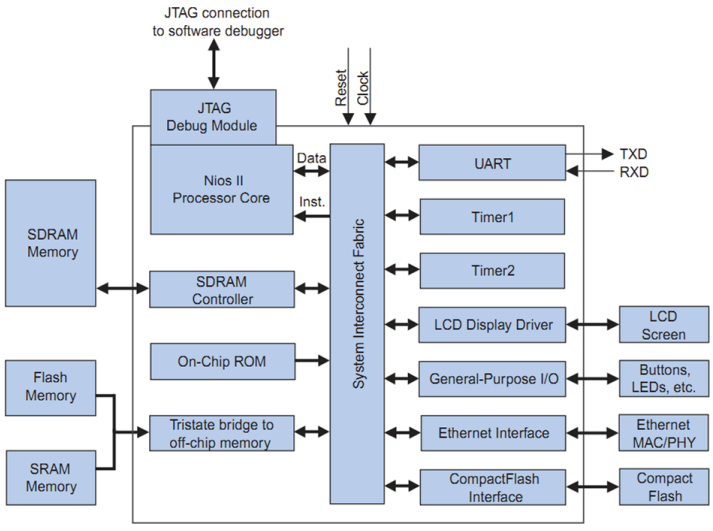


Figura A.4. Exemplo de um sistema em FPGA baseado no Nios II.

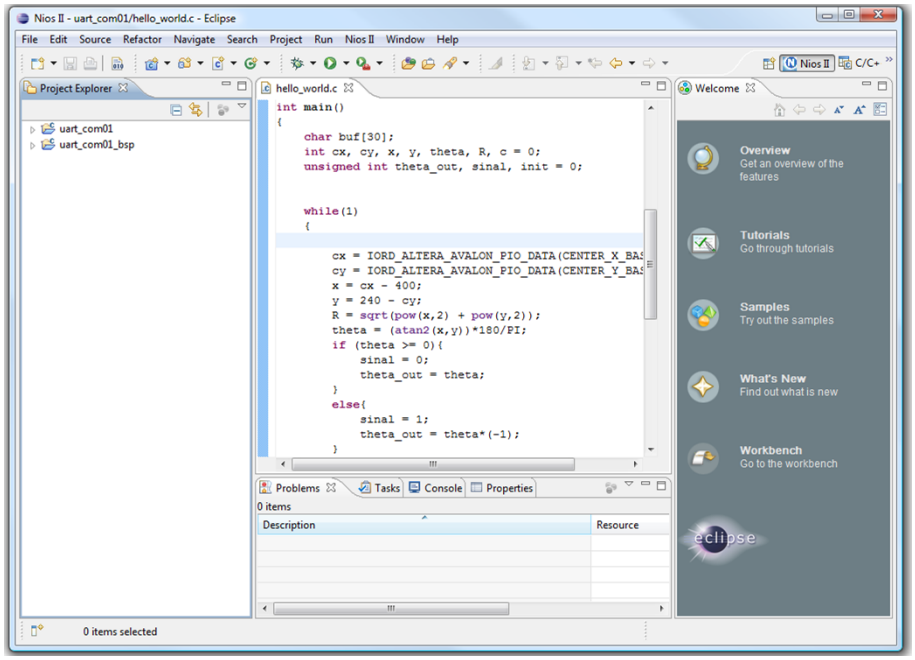


Figura A.5. Ambiente de desenvolvimento *software Nios II Software Build Tools (SBT) for Eclipse*

B. SENSORES

B.1 Sensor Ultrasônico - Sonar

O Sonar (do inglês ***SO****und* ***NA****vigation and* ***R****anging*), ou navegação e determinação da distância pelo som, é um sensor de alcance ativo em tempo de voo, quer dizer, usa a velocidade de propagação do sinal de ultrassom para calcular a distância entre o sensor e o objeto atingido. O termo sonar refere-se a qualquer sistema que usa ondas sonoras para medir distâncias.

O Sonar é muito utilizado como um instrumento auxiliar da navegação marítima. Este sistema inicialmente era empregado na localização de submarinos (em guerras), mas hoje em dia é também usado no estudo e pesquisa dos oceanos (determinação de profundidades ou de depressões) e na pesca, para a localização de cardumes.

Já na área da robótica móvel, este sensor é muito utilizado para o desvio de obstáculos e para a construção de mapas a partir da exploração de ambientes. A razão de sua popularidade é seu baixo custo e a simplicidade da resposta que fornece, a qual consiste na distância que existe entre o sensor e o obstáculo presente no alcance deste.

Quanto a sua classificação podem ser ativos ou passivos. O sonar ativo usa um transmissor e um receptor de som. O sensor cria um pulso de som, geralmente chamado de *ping* e espera pela reflexão do pulso. Já, O sonar passivo somente escuta sem transmitir. Usado em submarinos para detectar presença de navios.

B.1.1 Princípio de Funcionamento

O princípio de funcionamento de um sensor ultrassom utiliza ondas mecânicas sonoras. Faremos a seguir, uma breve revisão a respeito do som.

B.1.1.1 Princípio do Som

O som é uma vibração mecânica transmitida por um meio elástico. Essas ondas sonoras são ondas longitudinais, isto é, são produzidas por uma sequência de pulsos longitudinais. Estes pulsos podem ser produzidos, por exemplo, quando uma lâmina de aço muito fina oscila. O movimento de vai e vem da lâmina transfere energia para as moléculas do ar e estas transfere para as moléculas seguintes até serem atenuadas.

As ondas sonoras podem se propagar com diversas frequências, porém o ouvido humano

é sensibilizado somente quando elas chegam a ele com frequência entre 20 Hz e 20.000 Hz, aproximadamente (ver Figura B.1).

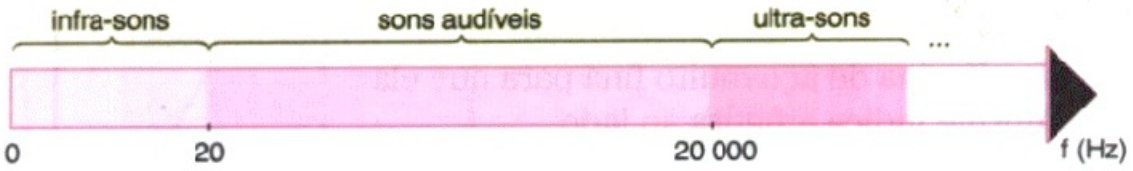


Figura B.1. Classificação do som quanto a sua frequência

Quando a frequência é maior que 20 KHz, as ondas são ditas ultra-sônicas, e menor que 20 Hz, infra-sônicas.

Velocidade do Som

A velocidade do som depende bastante do meio que ele atravessa. Suas propriedades físicas e sua velocidade mudam de acordo com as condições do ambiente. Em geral, a velocidade do som depende da pressão local, densidade e temperatura do meio. A velocidade do som V_s propagando-se no ar, com a influência da temperatura (T), possui a fórmula conforme (B.1).

$$V_s \approx 331.4 + 0.61 \times T \tag{B.1}$$

A Tabela B.1 apresenta a velocidade de propagação do som a 25řC para alguns materiais.

Tabela B.1. Velocidade de propagação do som em diferentes meios a 25žC

Meio	**Velocidade(m/s)**
Ar	346
Água	1498
Vidro	4540
Ferro	5200

B.1.1.2 Princípio de Funcionamento do Sonar

O princípio de funcionamento do sonar baseia-se na propagação de ondas ultra-sônicas (similar ao funcionamento de um radar, sendo este por ondas eletromagnéticas). Depois que o pacote de ciclos é transmitido pelo transmissor, o transdutor permanece um período de tempo sem transmitir sinal nenhum, o que é conhecido como *blanking time*. Uma vez que o eco do sinal transmitido é recebido pelo sensor (que pode ser o mesmo que emitiu), este alimenta um amplificador operacional de ganho variável; ganho que varia na medida em que o eco demora

mais ou menos tempo a ser detectado pelo sensor, isto com a finalidade de compensar possíveis atenuações ou dispersões da onda. Finalmente, a distância é calculada uma vez que o valor do limiar do circuito é superado, momento no qual a conta do tempo transcorrido entre transmissão e recepção finaliza. Isto pode ser melhor entendido conforme a Figura B.2.

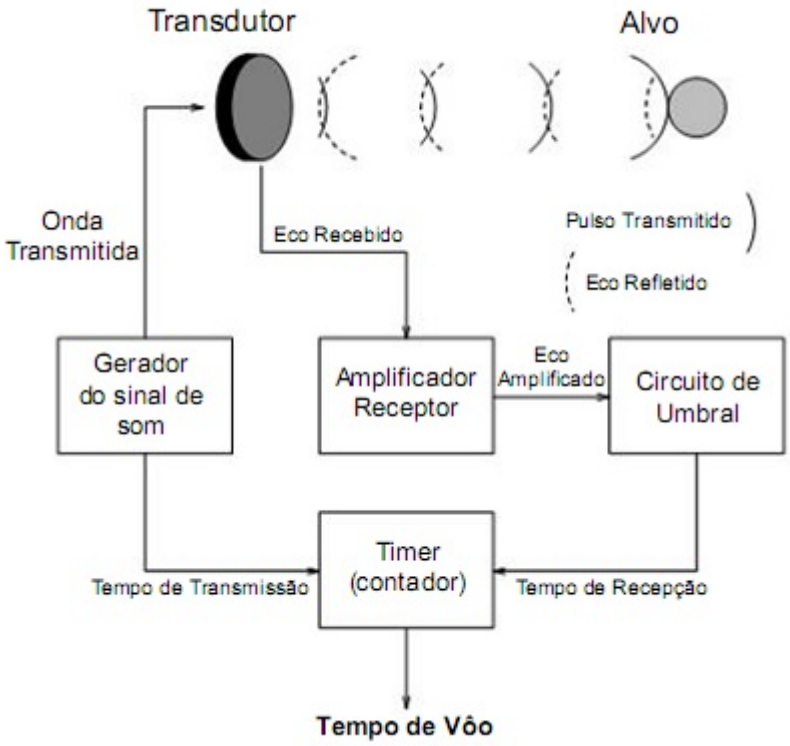


Figura B.2. Diagrama geral do funcionamento de um sensor de ultrassom

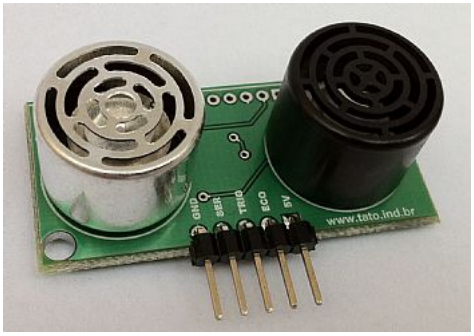
O transdutor, como mostrado na Figura B.2, pode conter os emissores e receptores encapsulados juntos, formando um único dispositivo, ou podem estar separados em dois dispositivos. Isto pode ser observado na Figura B.3.

Cálculo da Distância do Objeto

Sabendo-se a velocidade de propagação do som no meio em que ele se encontra é possível calcular a distância entre o sonar e o objeto medindo apenas o tempo gasto pela onda para atingir o objeto e retornar ao sensor. A onda refletida é comumente chamada de eco. Essa técnica de medição da distância pelo tempo do trajeto percorrido pela onda é chamada de Tempo de Voo (TOF, do inglês ***T****ime* ***O****f* ***F****light*).

Considere que a onda sonora percorre um caminho de ida e volta, a distância D entre o sensor e o objeto é dada pela Equação B.2. Note que o tempo é dividido por 2, pois, como mencionado,

(a)



(b)

Figura B.3. Módulos sonares de (a) um único encapsulamento e de (b) dois encapsulamentos

a distância é somente o caminho da ida e não o caminho de ida e volta.

$$D = c \times \Delta t/2 \tag{B.2}$$

onde c é a velocidade do som no ar e Δt é o tempo decorrido entre o envio e a detecção do sinal ultrassonico.

B.1.1.3 Erros na medição da distância

Para medir a distância que o som percorreu, ele precisa ser refletido de volta. Este som é uma onda transversal que bate em uma superfície geralmente plana. O som é então refletido, desde que as dimenções do objeto sejam grandes, comparado com o comprimento da onda.

O sonar, como todos os sensores, possui algumas desvantagens que comprometem seu desempenho. Tipicamente, um sistema de sensoriamento baseado em sensores de ultrassom tende a sofrer de imprecisões nas leituras de distância devido, principalmente, a erros relacionados com o circuito do transdutor, ou devido as características dos alvos a serem atingidos.

Erros causados por características do circuito do sensor

(a) O cálculo do tempo de transmissão-recepção assume que o começo do pulso transmitido faz parte do eco que posteriormente será recebido, e que deve superar o limiar do circuito. Se este não for o caso, o erro de distância poderia ser de até 23 cm.

(b) O amplificador de ganho variável faz uma aproximação de só 16 passos da curva exponencial ideal que compensa perdas por causa de atenuação e de múltiplas reflexões. Isto pode comprometer a magnitude do sinal amplificado que posteriormente entrará ao circuito de umbral e que definirá o tempo que transcorreu entre transmissão e recepção.

(c) O mecanismo de funcionamento do circuito de limiar envolve o uso de um capacitor de carga, que se vê fortemente afetado pela intensidade do eco recebido. No caso de reflexões de onda fortes, só se precisa de três ciclos do sinal para carregar o capacitor. Entretanto, para reflexões fracas, a carga de capacitor pode demorar um tempo considerável, o que pode representar, em alguns casos, um cálculo de distancia errado.

(d) O padrão de radiação do sensor é irregular (como já foi mencionado acima), o que significa que os objetos detectados sejam atingidos pelo sinal com uma energia que depende da inclinação da onda sobre a superfície atingida. Especialmente, para orientações perpendiculares ao alvo (isto significa um desvio de zero grau do eixo do sensor), o sinal que incide sobre o objeto provém do lóbulo principal, e portanto, a medida de distância será mais exata. Já para o caso de sinais emitidos nas regiões definidas pelos demais lóbulos, a precisão na medida distância vai se degradando conforme estes sejam mais pequenos.

Erros causados pela característica do objeto

(a) *Superfície*: A superfície ideal do objeto é dura e lisa. Esta superfície reflete uma quantidade maior do sinal do que uma macia e porosa. Um eco fraco é resultado de um objeto pequeno e macio. Isto reduz a distância de operação do sensor e reduz a sua precisão.

(b) *Distância*: Quanto menor for a distância do sensor ao objeto, mais forte será o eco. Deste modo, a medida que a distância aumenta, o objeto precisa ter melhores características de reflexão para retornar um eco suficiente.

(c) *Tamanho*: Um objeto grande tem mais superfície para refletir o sinal do que um menor. A área de superfície reconhecida como alvo é a área mais próxima ao sensor.

(d) *Ângulo*: A inclinação da superfície do objeto em relação ao sensor afeta o modo que o objeto reflete a onda. A porção perpendicular ao sensor retorna o eco. Se o objeto todo estiver a um ângulo grande, o sinal é então refletido para longe do sensor e o eco não é detectado.

A Figura B.4 exibe as melhores características de um objeto para uma boa detecção pelo sensor sonar.

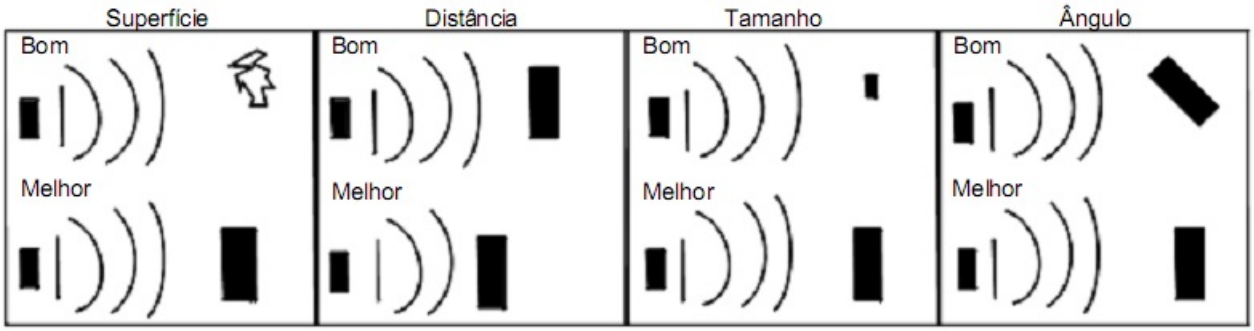


Figura B.4. Tipos de reflexão do som

Uma característica importante que define o desempenho do sensor é o padrão de radiação do transdutor, apresentado na Figura B.5. Como pode ser observado, o transdutor não emite a energia de forma homogênea, em vez disto, o padrão de radiação está conformado por lóbulos que vão decrescendo em intensidade na medida em que o sinal de ultrassom transmitido se afasta do eixo principal do sensor. Este padrão de radiação define um cone sonoro dentro da qual se encontra a medição de distância.

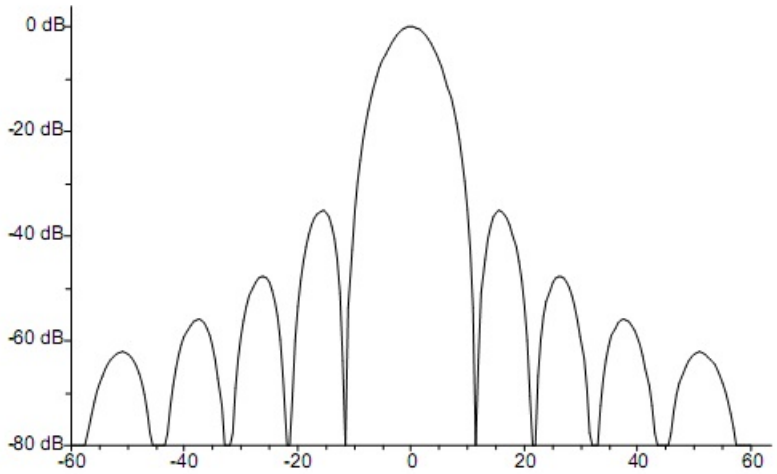


Figura B.5. Padrão de propagação do sinal de ultrassom. O eixo x mostra a distância angular em relação ao eixo do transdutor, em graus. O eixo y exibe a intensidade acústica, em dB.

B.2 Sensor Laser - Ladar

O Ladar (do inglês ***LA****ser* ***D****etection* ***A****nd* ***R****anging*), ou detecção e determinação da distância pela luz, é uma tecnologia óptica de detecção remota que mede propriedades da luz refletida de modo a obter a distância e/ou a velocidade de um determinado objeto.

Também conhecido como *Laser Rangefinder*, o ladar fornece relativamente um novo sensor de alta resolução para a robótica. Comum em robótica de precisão por muitos anos, estes sensores estão se tornando mais comuns em aplicações relativamente baratas, devido à sua resolução, que é rica e geram dados rapidamente.

Quando um sensor gera toneladas de dados de forma rápida como a de um ladar, exige-se um maior desempenho do computador embarcado no robô. Isso, para tirar pleno aproveitamento das capacidades únicas deste sensor. Este não é o território de 8051's, Basic, ou PIC's de 8 ou 16 bits. Normalmente, é um processador de maior potência de 32 bits rodando um sistema operacional completo como o Linux.

O ladar emite e recebe dados em feixe de luz para calcular a distância do obstáculo. A resolução é cerca de milímetros. O consumo de energia varia, mas tende a ser um pouco mais do que nos sensores menores e menos sofisticados, como sonares, por exemplo.

É aplicada no âmbito da geodesia, arqueologia, geografia, geologia, geomorfologia, sismologia, engenharia florestal, oceanografia costeira, detecção remota e física da atmosfera. Também é usado para fins militares e como sensores na área de robótica móvel. A Figura B.6 mostra um exemplo de um sensor ladar.

B.2.1 Princípio de Funcionamento do Ladar

Os ladares podem ser pensados como pequenos sonares usando luz em vez de som para criar mapas 2D em sua proximidade, ou seja, detectando objetos próximos. Como os ladares utilizam a luz em vez de som, eles podem medir de forma extremamente rápida e com um campo de visão super estreito.

Como descrito na Figura B.7, o ladar emite um feixe de infravermelho e um espelho girante muda a direção desse feixe. O laser choca-se contra a superfície do objeto, que reflete a luz. A direção da luz refletida é novamente mudada pelo espelho e capturada pelo fotodiodo, que fica dentro do sensor.

O ladar é um equipamento que faz uma varredura do ambiente em um plano (ver Figura B.8). Ele faz uma leitura emum determinado angulo, passa alguma fração da resolução angular total

Figura B.6. O Ladar Hokuyo URG-04LX

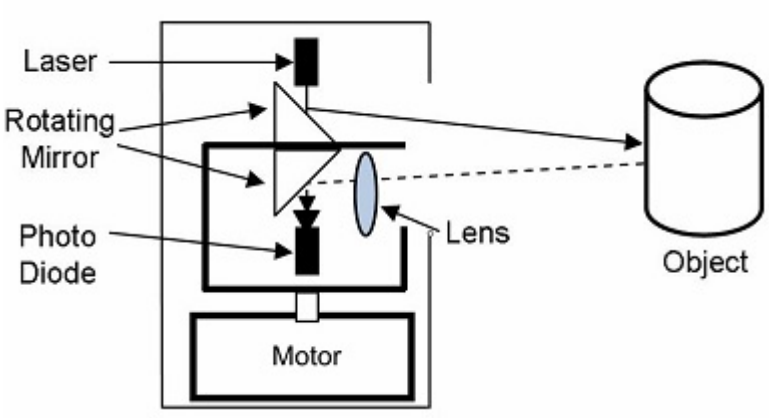


Figura B.7. Princípio de funcionamento de um ladar

e depois toma uma outra leitura. Isso é feito de forma contínua e repetida até que se complete um círculo.

Devido à restrições mecânicas, muitas vezes fica uma pequena porção do caminho radial 'cega' para o sensor, como visto na Figura B.8. Os dados coletados do laser formam tipicamente um mapa de duas dimensões. Então, em um diagrama polar, tem-se pontos da origem até o limite mecânico do sensor, havendo uma região sem imagem.

Especificações

Os Ladares são muitas vezes especificados pela resolução da medição, frequência e resolução angular. A resolução da medição se refere como a medida exata da distância em uma determinada direção. Ela é normalmente em torno de 1 a 3 mm. A frequência refere-se com que frequência

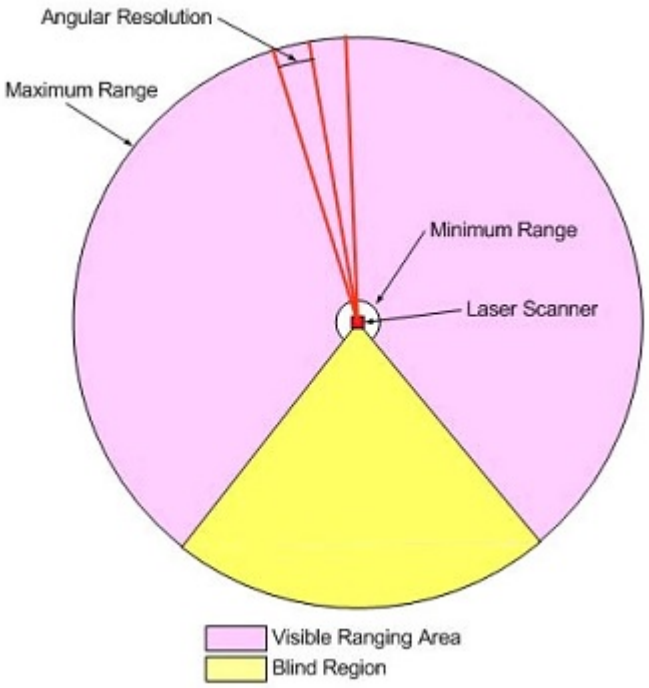


Figura B.8. Campo de visão de um ladar

o arco de 360 graus é varrido pelo sensor de varredura e é normalmente de 10Hz ou até mais. A resolução angular se refere para quantas amostras são tomadas a cada varredura completa de 360 graus do sensor. Este valor é muitas vezes mais de 500 passos.

Os *lasers rangerfinders* também pussuem normalmente um alcance mínimo e máximo onde se pode medir. Os mínimos são muitas vezes alguns centímetros e o intervalo pode ser mais de um metro podendo chegar até dezenas ou centenas de metros. O modelo utilizado neste trabalho é o URG 04LX da empresa Hokuyo, suas características são descritas na Tabela B.2.

Tabela B.2. Especificações do Ladar URG 04LX

Especificações	
Voltage	5.0V +/- 5%
Corrente	0.5A nominal
Raio De Detecção	20mm a 5,6m
Comprimento de Onda	785nm, Classe 1
Angulo de Detecção	240°
Frequência	10Hz
Resolução	1mm
Precisão	De 20mm a 1m: +/- 30mm
	De 1m a 4m: +/- 3% da medição
Resolução Angular	0.36°
Interface	RS232 e USB
Peso	160g