SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito: 05.02.2004

Assinatura: Ana Paula Japão Japão

**Projeto de uma unidade aritmética de
ponto flutuante - Padrão IEEE 754 -
implementada em computação
reconfigurável**

Murilo Inácio Rodrigues

Orientador: *Prof. Dr. Eduardo Marques*

Dissertação apresentada ao Instituto de Ciências Matemática e de computação – ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências de Computação e Matemática Computacional.

USP – São Carlos
Fevereiro/2004

A Comissão Julgadora:

Prof. Dr. Eduardo Marques

Prof. Dr. Onofre Trindade Junior

Prof. Dr. Renato Camargo Giacomini

"Não preciso me drogar para ser um gênio.
Não preciso ser um gênio para ser humano.
Mas preciso do seu sorriso para ser feliz..."
(Charles Chaplin)

Dedico este trabalho a minha família e aos
meus amigos e companheiros do LCR.

AGRADECIMENTOS

Primeiramente agradeço a Deus pela força que me deu para vencer cada um dos obstáculos.

A meu orientador e amigo Eduardo Marques pela paciência, amizade e incentivos dados em todos os momentos.

Aos meus amigos do LCR pelo companheirismo, em especial ao Rovilson, Alexandre, Denis e Marco Antonio que me ajudaram nos momentos cruciais de definição deste trabalho.

A minha querida amiga Jacqueline Augusto de Oliveira pela sua dedicação demonstrada durante as correções gramaticais deste trabalho.

Ao Adriano Kaminski Sanches pela sua hospitalidade e amizade em me receber para o término deste trabalho.

A todos meus amigos do ICMC pelos momentos descontraídos que passamos nesta maravilhosa cidade de São Carlos.

A CAPES pelo auxílio financeiro.

RESUMO

Operações de ponto flutuante representam uma tarefa comum em uma grande variedade de aplicações. Porém, tais operações frequentemente resultam em um gargalo do sistema, devido a grande quantidade de números de ciclos de máquina necessários para executá-las. No caso específico dos sistemas embutidos, essas limitações geralmente proíbem as aplicações de tempo real devido às limitações dos processadores. Visando amenizar o problema, este trabalho apresenta uma solução de aritmética de ponto flutuante baseada em computação reconfigurável denominada FPMU-*Floating-Point Modular Unit*. A FPMU foi implementada no padrão IEEE 754 de maneira modular e com técnicas arquiteturais de exploração do paralelismo, cuja finalidade foi explorar os diversos tipos de requisitos das aplicações de tempo real, tais como velocidade, espaço ocupado, funcionalidade e etc. Os resultados obtidos foram considerados satisfatórios, atendendo uma grande classe de aplicações de tempo real.

ABSTRACT

Floating-point operations represent a common task in a great variety of applications. However, such operations frequently result in a bottleneck, due to the large number of machine cycles required to compute them. In the specific case of the real-time systems these limitations generally is a trouble due to the limitations of the processors. Aiming at to brighten up the problem, this work presents a solution of based Arithmetic of floating-point in reconfigurable computation called FPMU-Floating-Point Modular Unit. The FPMU was implemented in the IEEE 754 standard in modular way and with architectural techniques of parallelism exploration, whose purpose was to explore the diverse types of requirements of the applications of real time, such as speed, busy space, functionality. The gotten results had been considered satisfactory, fulfilling of a great classroom of applications of real time.

SUMÁRIO

AGRADECIMENTOS	IV
RESUMO.....	V
ABSTRACT	VI
SUMÁRIO.....	VII
ÍNDICE DE FIGURAS E TABELAS	IX
SIGLAS UTILIZADAS.....	XI
1 Introdução.....	1
1.1 Contextualização	1
1.2 Objetivos	2
1.3 Justificativa e Motivação	3
1.4 Organização do Trabalho	4
2 Conceitos básicos.....	5
2.1 Tecnologias de Hardware Reconfigurável.....	5
2.2 Características das Aplicações Numéricas.....	7
2.3 HDL.....	9
2.3.1 VHDL	9
2.4 Computação Configurável	11
2.5 Software X Hardware.....	11
2.5.1 Tecnologia FPGA	12
2.5.2 Dispositivos Reconfiguráveis.....	13
2.6 Soluções Disponíveis	14
2.7 Trabalhos Relacionados	16
3 Aritmética de Ponto Flutuante	17
3.1 Visão Geral da Aritmética de Ponto Flutuante.....	17
3.2 Padrão IEEE 754	22
3.2.1 Representação binária.....	22
4 Explorando o Paralelismo	25
4.1 Atraso Delta	26
4.2 Paralelismo Temporal	29
4.3 Paralelismo Espacial	31
4.4 Paralelismo Espacial e Temporal	32
4.5 Técnicas para Projeto de Pipeline.....	33
4.5.1 Projeto de Pipeline com Múltiplas Funções.....	34
4.5.2 Medidas de Desempenho para Pipeline.....	38
4.5.2.1 Speedup (fator de aceleração)	38
4.5.2.2 Eficiência	39
4.5.2.3 Throughput	39
5 Implementação.....	40
5.1 Bloco 'FPU_Design' (Componente Principal).....	40
5.1.1 Interface.....	41
5.1.2 Implementação	42
5.2 Adição de Inteiro	44

5.2.1	Interface.....	44
5.2.2	Implementação.....	44
5.3	Adição em Ponto Flutuante.....	45
5.3.1	Interface.....	45
5.3.2	Implementação.....	46
5.3.3	Componente 'FSIMPLE_ADD_NORMALIZE_PIPE'.....	48
5.3.4	Implementação do componente 'FSIMPLE_ADD_PIPE'.....	49
5.4	Multiplicação.....	51
5.4.1	Interface.....	51
5.4.2	Implementação.....	51
5.5	Multiplicação em Ponto Flutuante.....	53
5.5.1	Interface.....	53
5.5.2	Implementação.....	53
5.6	Componente 'FSIMPLE_MULT'.....	55
5.7	Divisão de Inteiro.....	56
5.8	Divisão em Ponto Flutuante.....	57
5.9	Normalização.....	59
5.9.1	Implementação.....	59
5.10	Listagem dos componentes.....	60
6	Validação e Exemplo de Aplicação.....	63
6.1	Exemplo de aplicação.....	64
6.2	Resultados obtidos.....	67
7	Conclusão e Trabalhos Futuros.....	69
7.1	Conclusão.....	69
7.2	Trabalhos Futuros.....	69
	Referência Bibliográficas.....	71

ÍNDICE DE FIGURAS E TABELAS

Figura 2-1(a) Meio somado descrito em VHDL (b) Meio somador representado esquematicamente.....	10
Figura 2-2 Estrutura de um FPGA.....	13
Figura 2-3: Operação de adição e multiplicação compartilhando o mesmo hardware.	14
Figura 3-1: Diferença dos valores em número real para a representação em ponto flutuante em sistemas digitais.....	18
Figura 3-2: Representação para números em ponto flutuante.....	18
Figura 3-3: Arredondando o resultado para obter maior precisão.....	19
Figura 3-4: Adição de números de ordem de grandeza próximos melhora a precisão.	20
Figura 3-5: Falsa precisão.	20
Figura 3-6: Operação de multiplicação multiplica eventuais erros nos operandos.	21
Figura 3-7: Rearranjando os operadores para evitar a multiplicação de possíveis erros.	21
Figura 3-8: Formato binário de um Número em Ponto Flutuante.....	23
Figura 3-9: Padrão de Ponto-Flutuante IEEE 754.....	23
Figura 3-10: Tipo definidos pelo padrão de Ponto-Flutuante IEEE-754.....	24
Figura 4-1: Atraso na propagação do sinal.	27
Figura 4-2: Acumulo de atrasos no caminho do carry de um somador.....	28
Figura 4-3: Pipeline com três estágios.....	29
Figura 4-4: Diagrama de um pipeline com três estágios.....	30
Figura 4-5: Implementação de dois acumuladores seriais.	31
Figura 4-6: Implementação de dois acumuladores em paralelo.....	32
Figura 4-7: Implementação de uma FPU paralela.....	33
Figura 4-8: (a) Pipeline linear para multiplicação em ponto flutuante. (b) Pipeline para multiplicação em ponto flutuante com reentrada de dados.....	35
Figura 4-9: Pipeline para adição em ponto flutuante com reentrada de dados.	36
Figura 4-10: Pipeline da adição e multiplicação em ponto flutuante.	37
Figura 4-11: Tabela de tempo para pipeline multifuncional. (A) Adição; (M) Multiplicação.	38
Figura 5-1: Interface do componente principal da FPU – ‘FPU_Design’.....	41
Figura 5-2: Esquemático do componente 'FPU_Dsign'.....	43
Figura 5-3: Diagrama hierárquico dos componentes para operação em ponto flutuante.	43
Figura 5-4: Interface do componente que executa adição de inteiro – ‘ADD’.....	44
Figura 5-5: implementação do componente ADD.....	45
Figura 5-6: Interface do componente para adição em ponto flutuante – ‘FADD_PIPE’.....	46
Figura 5-7: Implentação do componente 'FADD_PIPE'.....	47
Figura 5-8: Diagrama hierárquico dos componentes da adição em ponto flutuante.	48
Figura 5-9: Implementação do componente 'FSIMPLE_ADD_NORMALIZE_PIPE'.....	49
Figura 5-10: Implementação do componente 'FSIMPLE_ADD_PIPE’.....	50

Figura 5-11: Interface de multiplicação – ‘FMULT’	51
Figura 5-12: Implementação do componente para multiplicação de inteiro	52
Figura 5-13: Interface do componente de multiplicação em ponto flutuante - 'FMULT_PIPE'	53
Figura 5-14: Implementação do componente de multiplicação em ponto flutuante	54
Figura 5-15: Diagrama hierárquico dos componentes da multiplicação em ponto flutuante	55
Figura 5-16: Esquemático da implementação do componente 'FSIMPLE_MULT'	56
Figura 5-17: Implementação do componente de divisão de inteiro	57
Figura 5-18: Esquemático do componente para divisão em ponto flutuante - 'FDIVI_PIPE'	58
Figura 5-19: Diagrama hierárquico dos componentes da divisão em ponto flutuante	59
Figura 5-20: Diagrama esquemático do componente de normalização	60
Figura 6-1: Formato do arquivo gerado para testar o componente de adição da FPU	63
Figura 6-2: Código do testbench para FPU	64
Figura 6-3: Interligação entre o Nios e a FPU	66
Figura 7-1: Adicionando a funcionalidade da FPU na ALU do Nios	70

SIGLAS UTILIZADAS

<i>VLSI</i>	Very Large Scale Integration
<i>ASIC</i>	Application Specific Integrated Circuits
<i>DSP</i>	Digital Signal Processor
<i>EDA</i>	Eletronic Design Automation
<i>FPGA</i>	Field-Programmable Gate Arrays
<i>PLD</i>	Programmable Logic Device
<i>HDL</i>	Hardware Description Language
<i>EDA</i>	Eletronic Design Automation
<i>FPU</i>	Float Point Unit
<i>CLB</i>	Configurable Logic Block
<i>ALU</i>	Aritmetic Logic Unit
<i>VHDL</i>	VHSIC Hardware Description Language
<i>VHSIC</i>	<i>Very High Speed Integrated Circuit</i>
<i>NaN</i>	Not a Number
<i>ULA</i>	Unidade Lógica Aritmética
<i>RTR</i>	Run-Time Reconfiguration
<i>PAL</i>	Programmable Array Logic

1 INTRODUÇÃO

1.1 CONTEXTUALIZAÇÃO

Em busca de uma melhor performance, projetistas de hardware e software concentram seus esforços de projetos em busca de trechos em seus códigos que possam ser paralelizados. Porém, para um real aproveitamento do paralelismo, uma nova abordagem precisa ser adotada: além do desenvolvimento do software é necessário desenvolver, também, o hardware (CPU, ALU, etc.) que execute mais eficientemente um determinado trecho de software.

Em busca da eficiência e de um menor custo de projeto pode-se adotar soluções específicas para cada problema específico como, por exemplo, adotar uma variação da representação para número em ponto flutuante para um operador do qual se saiba de antemão a precisão necessária para cobrir toda a faixa de valores que este possa assumir durante os cálculos. Esta solução pode ser adotada graças às novas tecnologias de hardware reconfigurável e às linguagens de descrição de hardware (HDL) que aceleram o processo de desenvolvimento, implementação e manutenção. Este trabalho vem a ajudar aos projetistas que estão conduzindo suas pesquisas para esta nova tecnologia que une software e hardware com eficiência na execução e rapidez no desenvolvimento das aplicações em tempo real.

1.2 OBJETIVOS

O objetivo principal deste trabalho é desenvolver uma FPU¹ que possa aproveitar a reconfigurabilidade das FPGAs, possibilitando seu uso a uma grande gama de aplicações que hoje não são desenvolvidas em hardware devido a escassez de ferramentas no mercado que auxilie desenvolvedores migrar cálculos em número real, originalmente desenvolvidos para execução sequencial, para uma plataforma baseada no paralelismo. Para isso, foi buscado o estado da arte na construção de hardware para aplicações numéricas que requerem alto desempenho, que será explanado no decorrer desta dissertação.

Os resultados obtidos com este trabalho serão aplicados ao Projeto ARMOSH [Romero, 2000], [Aragão, Romero, Marques, 2000] (Aprendizado de Robôs Móveis via *Software e Hardware* - processo FAPESP n.º. 2000/02959-3), desenvolvido pelo Laboratório de Computação Reconfigurável - LCR do ICMC-USP, em conjunto com o LABIC (Laboratório de Inteligência Computacional) e com a CMU (*Carnegie Mellon University - USA*), e também servirá ao Projeto ARCHITECT [Gonçalves, Wolf, Coelho, Teixeira, Ribeiro, Marques, 2000], [Rodrigues, Gonçalves, Wolf, Osório, Ribeiro, Texeira, Marques, 2001]. Como também auxiliar a outros projetos que venham a ser desenvolvidos em computação reconfigurável que necessitem de precisão de cálculos em ponto flutuante. Para isso, a FPU foi desenvolvida com uma arquitetura modular e maleável de maneira que fique viável a especialização para uma determinada aplicação de tempo real. Em função da característica de sua modularidade ela foi denominada FPMU-*Floating-Point Modular Unit*.

¹FPU – Float Point Unit – esta abreviatura será utilizada a partir deste ponto para denominar Unidade de Ponto Flutuante.

1.3 JUSTIFICATIVA E MOTIVAÇÃO

Tradicionalmente, implementam-se sistemas computacionais em *hardware* (isto é, VLSI, ASICs, *gate-arrays*, etc.) ou implementa-se em *software* para ser executado em processadores (DSPs, microcontroladores, microprocessadores embutidos ou de finalidade geral). A evolução das metodologias de projeto de *hardware*, apoiadas em poderosas ferramentas EDA que aceleram o ciclo de desenvolvimento, e especialmente o surgimento de dispositivos reconfiguráveis como os FPGAs, abriram um novo horizonte entre os extremos da computação de finalidade geral e o *hardware* dedicado. Os FPGAs combinam a flexibilidade de dispositivos programáveis (como PALs e microprocessadores de finalidade geral) com o desempenho do *hardware* de finalidade específica [Vincentelli, 1993].

Um projeto de sistema digital pode ser desenvolvido rapidamente empregando-se linguagens de descrição de hardware e, com a utilização de circuitos reconfiguráveis, pode-se implementar em campo um protótipo ou a versão final do projeto desenvolvido, sem necessidade de processos de fabricação de *chips*.

Porém, quando desenvolvedores tentam migrar suas rotinas que utilizam ponto flutuante para este novo ambiente, encontram uma grande barreira técnica: seus circuitos ficam extremamente grandes ou simplesmente não conseguem tratar ponto flutuante. Isto se deve a pouca informação e a falta de ferramentas adequadas para fornecer uma unidade aritmética em ponto flutuante customizada e portátil.

Alguns fornecedores oferecem componentes comerciais prontos que realizam as operações básicas (soma, subtração, multiplicação e divisão). Apesar destes componentes respeitarem o padrão IEEE-754, não são portáteis para outros sintetizadores lógicos e são pouco flexíveis, pois não conduzem, muitas vezes, aos resultados esperados (custo X velocidade X espaço) além de prender o desenvolvedor a uma família específica de produtos de um fornecedor [Amphion, 2002].

A grande característica da FPU aqui desenvolvida é ter seu código totalmente escrito em VHDL, ser modular e configurável, o que facilita o desenvolvimento e implementação da FPU a um determinado projeto, atendendo a diversas aplicações de tempo real.

1.4 ORGANIZAÇÃO DO TRABALHO

Este capítulo apresentou o contexto no qual se insere este trabalho, a motivação para o seu desenvolvimento e os objetivos a serem alcançados.

O Capítulo 2 introduzirá conceitos relacionados às tecnologias de hardware reconfigurável, as características das aplicações que utilizam operações em ponto flutuante, as soluções disponíveis no mercado e trabalhos relacionados.

O capítulo 3 mostrará a problemática de utilizar as operações em ponto flutuante em sistemas digitais e detalha o Padrão IEEE 754 que teve como objetivo padronizar tais operações.

No capítulo 4 são apresentados conceitos de paralelismo tais como *pipeline* e processamento paralelo, além de mostrar como aplicar tais conceitos numa aplicação de multiplicação e soma em ponto flutuante com a finalidade de ganhar-se performance.

No capítulo 5 são apresentados detalhes da implementação deste trabalho.

No capítulo 6 é descrita a forma de validação utilizada além de ser apresentada uma aplicação prática que utiliza a FPU desenvolvida neste trabalho e uma análise dos resultados obtidos.

E, finalmente, no capítulo 7 é feita a conclusão, destacando-se as contribuições mais importantes e citando possíveis trabalhos futuros como forma de aprimoramento deste trabalho.

2 CONCEITOS BÁSICOS

2.1 TECNOLOGIAS DE *HARDWARE RECONFIGURÁVEL*

Os processos utilizados para adaptar sistemas projetados com tecnologias tradicionais para tecnologia de *hardware* configurável podem ser divididos em duas partes [Goldblatt, 1998]:

A primeira parte é responsável por traduzir o projeto para a nova tecnologia, cujas tarefas são descritas a seguir:

- ❑ Codificação do projeto em HDL (Linguagem de Descrição de *Hardware*). 90% dos projetos recentes são projetados utilizando HDL [Leung, Shanblatt, 1997];
- ❑ Resolução de diferenças de arquitetura. Componentes que utilizem características específicas de uma tecnologia precisam ser substituídos por equivalentes na tecnologia de *hardware* configurável;
- ❑ Substituição de partes do código original (multiplicadores, somadores, comparadores, etc.) por blocos de código otimizados para o hardware escolhido. Exemplo de ferramentas que geram blocos de código: *Xilinx LogiCOR* [Xilinx, 2002] e *LogiBLOX functions*;
- ❑ Otimização do código, adicionando *pipeline*, a fim de manter o desempenho e o tempo de resposta do modelo original;
- ❑ Ajuste do tempo de resposta do sinal pois devido à mudança de tecnologia, algumas características elétricas tais como tempo de subida e de descida do sinal podem ser diferentes. Projetos que se apoiem em características particulares de uma determinada arquitetura precisam ser revistos a fim de retirar esta limitação;
- ❑ Utilização de um estilo de codificação HDL apropriado.

A segunda parte consiste na implementação do projeto, cujas tarefas são descritas a seguir:

- ❑ Escolha da FPGA levando-se em conta custo, velocidade, quantidade de portas lógicas disponíveis e suporte do fabricante.
- ❑ Síntese e simulação em ferramentas EDA. Nesta fase, provavelmente aconteçam ajustes no código HDL.
- ❑ Configuração do hardware configurável.

A fim de melhorar o desempenho e o tempo de desenvolvimento do projeto em *hardware*, são utilizadas ferramentas automáticas que possibilitam a construção de blocos de código que funcionam como uma caixa preta para o projetista. Estes blocos de código são construídos a partir de dados fornecidos pelo projetista e pretendem resolver um problema específico que permite a construção de um código mais enxuto e livre de falhas.

A maioria dos sistemas desenvolvidos em *hardware* configurável pretende tratar um problema específico e requer componentes também específicos. Assim, uma ferramenta para auxiliar na construção de funções aritméticas em ponto flutuante é muito bem-vinda, resolvendo problemas de espaço ocupado *versus* tempo de processamento e simplificando o trabalho do projetista.

Para a implementação de uma rede neural *perceptron*, por exemplo, é muito utilizado a multiplicação e a soma na expressão $S = \sum (E_i w_i)$ [Wolf, 2001], que representa a somatória dos valores de entrada do neurônio multiplicado pelo seu peso. Neste caso, pode-se optar em realizar as multiplicações em paralelo e depois as somas ganhando tempo [Chapman, 1996], processo conhecido como Aritmética Distribuída ou serializar as operações a fim de diminuir a quantidade de portas lógicas utilizadas [Xilinx, 1996].

Existem também casos onde os valores de um operador já são conhecidos, como de um neurônio em uma rede neural que utilize uma função sigmóide de ativação. Neste caso, a saída está contida no domínio $[1,-1]$ e a exatidão nos cálculos é um fator crucial. Uma possível implementação que diminuiria o espaço em *hardware* seria a retirada de tratamentos especiais, como *overflow*, por exemplo. Por outro lado, há sistemas que funcionam bem utilizando FPU no padrão IEEE-754 [IEEE, 1985], sendo desejável que continuem utilizando FPU com o mesmo padrão quando implementadas em *hardware* configurável.

Um projeto real requer investigação de várias alternativas para alcançar a solução ideal de um problema sendo que, para isso, o projetista precisa ter em mãos alternativas maleáveis que permitam testar diversas variações do modelo. A FPU aqui apresentada tem a característica de poder ser arranjada de diversas maneiras através de variações dela mesma, o que faz com que o projetista possa testar todas as possibilidades de implementação em um tempo reduzido, obtendo assim uma melhor implementação. Isto é conseguido com a atribuição de diferentes valores às variáveis: quantidade de estágios de *pipeline*, *throughput*, latência e precisão nos cálculos.

2.2 CARACTERÍSTICAS DAS APLICAÇÕES NUMÉRICAS

Existem problemas que requerem resposta em tempo real e demandam uma grande quantidade de cálculos em ponto flutuante para que um resultado satisfatório seja alcançado com um bom nível de precisão, como o caso de algoritmos para o campo da robótica [Wolf, 2001].

Estes problemas podem ser resolvidos desenvolvendo-se uma arquitetura extremamente paralela e que explore as características particulares do problema específico forçando, assim, a construção de *hardware* voltado a uma área restrita e orientado a aplicação específica ao invés do uso de *hardware* de propósito geral. Porém, com a limitação da área de aplicação de um processador, diminui-se também seu conjunto de possíveis usuários, fator este que ocasiona um aumento no custo de desenvolvimento e produção por unidade.

A utilização de *hardware* reconfigurável normalmente é uma boa solução para conseguir-se desenvolver um *hardware* específico às necessidades do projeto, a um custo e tempo de desenvolvimento menor. Existe no mercado uma grande quantidade de ferramentas EDA para projetos descritos em HDL e projetos construídos com a utilização de diagramas esquemáticos. Estas ferramentas estão em constante evolução e cada vez mais auxiliam e facilitam o trabalho do projetista.

As características comuns das aplicações numéricas que demandam grande quantidade de cálculo são: [Stone, 1990]

- ❑ Utilização constante da aritmética de ponto flutuante, por esta permitir representar uma grande faixa de números;
- ❑ Os modelos de cálculos usam um conjunto discreto de pontos no espaço e tempo, o que fisicamente significa representar funções contínuas;
- ❑ O projetista tem uma razoável flexibilidade na determinação do tamanho do problema, usando uma representação discreta do modelo;
- ❑ O projetista pode escolher entre mais precisão dos resultados ou menor tempo de computação.

Tais características sugerem que o projetista tenha uma razoável flexibilidade para ajustar seu projeto às suas reais necessidades. Unindo-se a este fato a flexibilidade do *hardware* reconfigurável, a utilização de uma linguagem de descrição de *hardware* bem estruturada e as ferramentas de auxílio ao projeto cria-se uma nova filosofia para desenvolvimento de novos projetos, onde o tempo de desenvolvimento e o risco de projeto são menores do que no desenvolvimento tradicional.

2.3 HDL

A fim de conseguir vantagens nas tecnologias de semicondutores, têm sido desenvolvidos métodos de projeto e ferramentas de desenvolvimento mais complexas e poderosas. Um alto nível de abstração ao elaborar um projeto não é mais uma opção, mas sim, uma necessidade. Por esta razão, Linguagens de Descrição de Hardware (HDL) foram desenvolvidas. Ferramentas de síntese transformam entradas HDL em esquemas lógicos para a tecnologia física escolhida. Dessa maneira, um projeto pode ser sintetizado para FPGA e testado em campo em um tempo de produção rápido e, depois de validado, poderia ser sintetizado para ASIC, economizando tempo e diminuindo custos em todas as fases de projeto.

2.3.1 VHDL

VHDL é uma linguagem para descrição de *hardware* VHSIC (*Very High Speed Integrated Circuit*) padronizada pelo IEEE 1076.1, que oferece as seguintes vantagens para projetos digitais [Chang, 1997]:

- ❑ Padronização: reduz confusões e facilita interfaces entre ferramentas, fornecedores, e produtos diferentes.
- ❑ Suporte da indústria: com o lançamento de ferramentas de VHDL mais potentes e eficientes cresce também o compromisso das indústrias com esta tecnologia.
- ❑ Portabilidade: O mesmo código VHDL pode ser simulado e usado em ferramentas de diversos fornecedores e diferentes estágios de desenvolvimento.
- ❑ Capacidade de modelagem: VHDL foi desenvolvido para modelar todos estágios do desenvolvimento. VHDL pode descrever tanto comportamentos matemáticos como também rotinas que envolvem modelos complexos. Isso permite o uso de múltiplas arquiteturas e associá-las com o mesmo projeto durante o desenvolvimento.
- ❑ Reusabilidade: VHDL permite a criação de bibliotecas com códigos parametrizados desenvolvidos para uso futuro.

- Independência de tecnologia: As funcionalidades e comportamentos dos componentes podem ser criados em VHDL independente da tecnologia em que estes serão implementados.
- Documentação: VHDL é uma linguagem de alto nível cuja codificação pode ser considerada como uma documentação do projeto.
- Novas metodologias de desenvolvimento: VHDL cria uma nova metodologia que aumenta a produção e diminui o ciclo de desenvolvimento, abaixando os custos.

A título de ilustração, na Figura 2-1 é apresentado um trecho da descrição em VHDL de um meio somador com sinal de *enable* e, ao seu lado, o correspondente em diagrama esquemático, retirado de [IEEE, 1993].

Figura 2-1(a) Meio somado descrito em VHDL (b) Meio somador representado esquematicamente.

O trabalho de síntese do código VHDL para o nível de portas lógicas é feito pelo sintetizador, que procura explorar, ao máximo, as vantagens da arquitetura escolhida. Porém, o estilo de codificação influencia de forma decisiva o processo de síntese mostrando, assim, a importância da utilização de ferramentas de auxílio que forneçam códigos mais enxutos para o processo de síntese.

2.4 COMPUTAÇÃO CONFIGURÁVEL

O paradigma de construção de *hardware* vem sofrendo mudanças com o amadurecimento das tecnologias que levaram ao desenvolvimento de novas metodologias e da computação configurável. Isso estimulou o aparecimento de novas ferramentas EDA (*Electronic Design Automation*) que auxiliam todo o desenvolvimento do projeto. Hoje já é possível utilizar linguagens de alto nível para desenvolver *hardware*, possibilitando também o uso de metodologias antes aplicadas somente a desenvolvimento de *software*, tais como programação orientada a objeto e modularização.

2.5 SOFTWARE X HARDWARE

Numa implementação tradicional de um sistema computacional decide-se entre implementação em *hardware* ou *software*. Em alguns sistemas, toma-se esta decisão para cada sub-tarefa, implementando algumas em *hardware* e outras em *software*. Os projetos em *hardware* oferecem alto desempenho, pois:

- ❑ São especializados para o problema em particular – não há um *overhead* extra para interpretação de instruções ou circuitos extras para solucionar um problema mais genérico.
- ❑ São relativamente rápidos devido à execução espacial com um alto grau de paralelismo.

Já as implementações em *software* exploram um processador de finalidade geral que interpreta um determinado fluxo de dados como instruções que indicam quais operações devem ser realizadas. Como resultado o *software* é:

- ❑ Flexível – uma tarefa pode ser mudada simplesmente trocando-se o fluxo de instruções numa memória regravável.
- ❑ Relativamente lento – devido principalmente à execução temporal.

Cada nova geração de microprocessadores possibilita um aumento de desempenho na execução das aplicações. Entretanto, na maioria dos casos, a aplicação precisa ser novamente compilada para melhor usufruir as novas características desses processadores.

Em implementações espaciais, cada operador existe em um diferente ponto no espaço (área do *chip*), permitindo que a computação explore o paralelismo e alcance uma alta taxa de transferência de dados e baixa latência. Em implementações temporais, um pequeno número de recursos computacionais é reutilizado ao longo do tempo, permitindo que a computação seja implementada de forma compacta.

Um recente avanço nos esforços de obter maior poder de processamento e maior flexibilidade de projeto é o uso de dispositivos lógicos programáveis (PLDs). Estes dispositivos contêm *hardware* configurável que pode ser programado para a aplicação executada, tornando possível obter um desempenho comparável aos ASICs sem o custo e risco inerentes a esta tecnologia. Os dispositivos mais comuns de *hardware* configurável são os FPGAs.

2.5.1 TECNOLOGIA FPGA

O hardware configurável mais utilizado hoje é o FPGA (*Field Programmable Gate Array*). A arquitetura de um FPGA consiste de um arranjo 2-D de blocos lógicos. A comunicação entre os blocos lógicos é realizada através de recursos de interconexão. A borda externa deste arranjo consiste de blocos de entrada e saída (ver Figura 2-2). Blocos lógicos, também chamados de CLB (*Configurable Logic Block*) são, na maioria das vezes, constituídos por LUTs (*Lookup Tables*). LUTs são memórias endereçáveis que podem representar uma equação lógica. Os recursos de interconexão entre os CLBs também são configuráveis, possibilitando a representação de equações mais complexas pelo FPGA.

Figura 2-2 Estrutura de um FPGA

O principal benefício dos FPGAs e, mais amplamente, dos dispositivos lógicos programáveis, é que eles introduzem uma classe de dispositivos configuráveis pós-fabricação que suportam a realização de computação espacial. Além disso, destacam-se: a sua ótima flexibilidade em função da alta capacidade lógica; a facilidade de programação pelo usuário final, que reduz o tempo de fabricação para minutos; o ciclo curto de projeto que possibilita um rápido *time-to-market* (ciclo de tempo entre a identificação de um mercado e o desenvolvimento do produto para este mercado) e o baixo custo do dispositivo e dos custos fixos do projeto [Koopman, 1996].

2.5.2 DISPOSITIVOS RECONFIGURÁVEIS

A característica de reconfigurabilidade das FPGAs permite o compartilhamento de *hardware* entre projetos ou blocos distintos quando estes são acionados em tempos distintos [Ribeiro, 2002], [Teixeira, 2002].

Considerando-se, por exemplo, uma aplicação que utilize somente adição em um primeiro estágio e utilize somente multiplicação no estágio seguinte: as operações de adição e multiplicação podem compartilhar o mesmo espaço em *hardware* em tempos diferentes. Assim, ao passar do primeiro estágio para o segundo, o *hardware* é reconfigurado para operação de multiplicação no *hardware* que antes estava configurado para realizar a operação de adição (ver Figura 2-3). Soluções para esta implementação podem ser encontradas nas monografias [Teixeira, 2002] e [Ribeiro, 2002].

Figura 2-3: Operação de adição e multiplicação compartilhando o mesmo hardware

Um aspecto comum nessas aplicações é que a utilização de um sistema de computação reconfigurável pode prover um desempenho bem superior ao dos processadores de finalidade geral ou *workstations*, pois permitem simplificar rotinas em nível do *hardware*. O custo da lógica reconfigurável é tipicamente menor que o de aumentar a capacidade do microprocessador. É comum observar altos níveis de desempenho de supercomputadores em sistemas que custaram muito menos para serem montados[Luk, Cheung, Shirazi, 2002],[Jaenicke, LUK, 2001],[Ho, Leong, Leong, Becker, Glesner, 2002].

2.6 SOLUÇÕES DISPONÍVEIS

Muito está sendo investido na construção de ferramentas de auxílio ao projetista, tanto para desenvolvimento de *hardware* como *software*. Há disponíveis no mercado alguns componentes comerciais para serem usados em *hardware* configurável e algumas soluções com código aberto. As soluções comerciais têm a desvantagem já mencionada de não ser portátil para ferramentas de outros fabricantes e as soluções de código aberto não oferecem suficiente suporte técnico ao projetista. Algumas soluções para FPU disponíveis no mercado são mostradas a seguir:

A empresa *Gaisler Research* oferece uma FPU, de acordo com o padrão IEEE 754, chamada GRFPU – *High-Performance Floating-Point Unit*, que realiza operações de adição, multiplicação, divisão, raiz quadrada, conversão e comparação com precisão simples e precisão dupla. Este componente utiliza aproximadamente 8.000 LUTs quando sintetizado para família *Virtex II* da *Xilinx* e alcança um *throughput* de 65 MFLOPS sendo executada a 65 MHz. O preço aproximado do *netlist* do GRFPU é de R\$ 30.000. Para o código VHDL o preço pode chegar a R\$ 150.000 [GAISLER, 2003].

A empresa *Open Cores Org* disponibiliza uma FPU com precisão simples desenvolvida no padrão IEEE 754 com cinco ciclos de execução e quatro estágios de *pipeline*. Sintetizado no *Quartus II 3.0* da *Altera* este componente ocupa 67% do total de elementos lógicos de uma FPGA da família *Stratix*. Sua grande limitação é a baixa frequência de *clock* alcançada, apenas 4.87 MHz [OpenCores, 2003], [Asics, 2003].

A empresa *Digital Core Design* oferece uma FPU com precisão simples de acordo com o padrão IEEE 754 chamada DFPAU [CoreDesign, 2003]. A tabela a seguir informa a área e performance alcançada pela DFPAU sintetizada com as ferramentas da *Altera*.

Família	Elementos lógicos	Performance
Stratix	2.627	64 MHz
APEX 2 [^]	2.670	54 MHz
APEX 20KE	2.654	48 MHz

A empresa *Amphion Semiconductor Ltd* oferece uma biblioteca de operações em ponto flutuante com precisão simples de acordo com o padrão IEEE 754. Esta biblioteca é composta por três componentes: *Float-point Adder*, *Float-point Multiplier* e *Float-point Divider*, para operações de adição, multiplicação e divisão [Amphion, 2003]. A tabela a seguir informa a quantidade de células lógicas utilizada e a performance alcançada quando sintetizada com as ferramentas da *Altera*.

Família	Células lógicas	Performance	Parâmetros
EPF10K30A	618	35 MHz	Floating-point adder (M=16, E=6, P=3)
	785	37 MHz	Floating-point multiplier (M=16, E=6, P=2)
	2.051	41 MHz	Floating-point divider (M=16, E=6, P=6)

A própria *Altera* fornece uma *Megafunction* denominada ALTFP_MULT que implementa a operação de multiplicação com precisão simples (32 bits), precisão dupla (64 bits) e precisão estendida (43 a 64 bits). Sintetizado para a família da *Stratix*, na precisão simples, este componente alcança uma ótima frequência em se tratando de FPGA (102,91 MHz). Ao implementar a multiplicação em ponto flutuante utiliza um total de 1.633 elementos lógicos [Altera, 2003].

2.7 TRABALHOS RELACIONADOS

Em [Luk, Cheung, Shirazi, 2001] é descrito um método para customizar uma representação de um número em ponto flutuante tendo como alvo a tecnologia de hardware reconfigurável. Este método permite encontrar uma representação binária ideal para cada operador da aplicação levando em conta espaço em *hardware*, tempo de execução e precisão. A idéia básica desse método é simular as operações em ponto flutuante para todas as operações da aplicação e ir diminuindo a precisão enquanto seus resultados são comparados com os resultados inicialmente esperados. Este processo continua enquanto a diferença entre estes dois resultados estiverem dentro da faixa de erro especificado.

Em [Ho, Leong, Leong, Becker, Glesner, 2002] é apresentado um sistema para prototipação rápida de projetos para *hardware* que utiliza operações em ponto flutuante. Este trabalho fornece uma classe, desenvolvida na linguagem *Perl*, que simula os cálculos em ponto flutuante envolvidos numa operação específica, a fim de descobrir qual a mínima precisão requerida para os cálculos e operadores envolvidos. A precisão é fornecida em números de bits requeridos para o expoente e para a mantissa.

3 ARITMÉTICA DE PONTO FLUTUANTE

Embora números inteiros forneçam uma representação exata para valores numéricos, eles têm uma grande desvantagem: a inabilidade para representar valores fracionários. Aritmética em ponto flutuante resolve este problema. Programadores são advertidos da perda de velocidade associada com aritmética de ponto flutuante; contudo, os benefícios obtidos com o uso de ponto flutuante superam esta desvantagem.

3.1 VISÃO GERAL DA ARITMÉTICA DE PONTO FLUTUANTE

Um grande problema com a aritmética de ponto flutuante é que ela não segue as regras padrões da álgebra. Números inteiros também não seguem as regras padrões da álgebra, porque a representação deles é feita com números finitos de *bits*. Não se pode representar nenhum valor inteiro acima do ‘valor máximo’ ou abaixo do ‘valor mínimo’ suportado por determinada representação binária. Ponto flutuante, não só sofre deste mesmo problema, como também de outro: números inteiros são um subconjunto dos números reais. Sendo assim, os valores de ponto flutuante representam o mesmo conjunto de inteiros infinitos. Contudo, há um número infinito de valores entre quaisquer dois números reais; assim o problema é infinitamente pior (Figura 3-1). Entretanto, da mesma forma que foram limitados os valores na faixa entre máximos e mínimos, não se pode representar todos os valores entre essa faixa [Koren, 1993],[Hwang, 1976],[Flores, 1963],[Goldberg, 1991].

Figura 3-1: Diferença dos valores em número real para a representação em ponto flutuante em sistemas digitais

Para representar números reais, o mais comum é o uso da notação científica, usando a maior parte dos dígitos para representar a mantissa e a menor parte para representar o expoente(Figura 3-2). Assim, estes números de ponto flutuante podem somente ser representados através de um número específico de dígitos significativos. Isto tem um grande impacto na operação com aritmética de ponto flutuante. É fácil ver o impacto de utilizar precisão limitada na aritmética de ponto flutuante: adotando um formato simplificado de ponto flutuante decimal, por exemplo, com uma mantissa com três dígitos significativos e um expoente decimal com dois dígitos, sendo a mantissa e o expoente com valores sinalizados.

Figura 3-2: Representação para números em ponto flutuante.

Ao adicionar ou subtrair dois números em notação científica, é necessário o ajuste dos expoentes dos dois números para o mesmo valor. Por exemplo, antes de adicionar $1.23e1$ à $4.56e0$, ajustam-se os valores para que eles tenham o mesmo expoente. Um caminho a seguir é converter $4.56e0$ para $0.456e1$ e então adicionar à $1.23e1$, produzindo como resultado $1.686e1$. Infelizmente, o resultado não cabe em três dígitos significativos; assim, é preciso truncá-lo em três dígitos significativos. Arredondamento, geralmente produz uma melhor precisão, e assim obtém-se $1.69e1$ (Figura 3-3). Como se pode ver, a perda de precisão afeta a exatidão do valor.

Figura 3-3: Arredondando o resultado para obter maior precisão.

No exemplo anterior foi possível arredondar o resultado porque os cálculos foram feitos com cinco dígitos significativos. Se os cálculos estivessem limitados a três dígitos significativos, isto é, a FPU trabalhando internamente com três dígitos, ocorreria o truncamento do último dígito, resultando assim 1.68e1 ao invés de 1.69e1, que contém um erro maior do que o resultado arredondado. Com isso conclui-se que: dígitos extras durante a computação melhoram a exatidão dos resultados.

A perda de exatidão durante um simples cálculo não seria muito preocupante. No entanto, quando se calcula uma sequência de operações em ponto flutuante, o erro pode ir acumulando e produzir um grande efeito sobre o resultado final. Por exemplo, supondo a adição de 1.23e3 com 1.00e0: o primeiro passo seria o ajuste dos números para o mesmo expoente, obtendo assim: 1.23e3 e 0.001e3. Em seguida seria realizada a soma destes valores, resultando em 1.231e3. Este resultado parcial deve ser arredondado para que possa ser representado com três dígitos, isto é, 1.23e3 como resultado final. Contudo, supondo a adição de 1.00e0 com 1.23e3 dez vezes, o resultado da primeira adição seria 1.23e3, assim como o resultado da segunda adição, e da terceira, e da quarta,... e da décima adição de 1.00e3 com 1.23e3. De outra maneira, se fosse somado 1.00e0 com ele mesmo dez vezes, e então adicionado o resultado (1.00e1) à 1.23e3, o resultado obtido seria diferente: 1.24e3 (Figura 3-4).

$\begin{array}{rcl} 1.23e3 & \xrightarrow[\text{ajusta}]{\text{expoente}} & 1.230e3 \\ 1.00e0 & & 0.001e3 \\ 1.23e3 & & 1.231e3 \end{array}$	$\begin{array}{rcl} 1.00e0 & \xrightarrow[\text{repetindo}]{\text{dez vez}} & 9.000e0 \\ 1.00e0 & & 1.000e0 \\ 2.00e0 & & \end{array}$
$\begin{array}{rcl} 10.000e0 & \xrightarrow[\text{normaliza}]{\text{resultado}} & 1.000e1 \end{array}$	$\begin{array}{rcl} 1.23e3 & \xrightarrow[\text{ajusta}]{\text{expoente}} & 1.230e3 \\ 1.00e1 & & 0.010e3 \\ 1.24e3 & & 1.240e3 \end{array}$

Figura 3-4: Adição de números de ordem de grandeza próximos melhora a precisão.

Outro problema com adição e subtração é que essas operações podem gerar um resultado com uma falsa precisão. Considerando a subtração: 1.23e0 - 1.22e0. Isto produz 0.01e0. No entanto, isto é matematicamente equivalente a 1.00e-2, e escrito desta forma sugere que os dois últimos dígitos sejam exatamente zeros, mas infelizmente, tem-se somente um dígito significativo (Figura 3-5). De fato, algumas FPU e pacotes de *software* de ponto flutuante inserem dígitos (*bits*) aleatórios nas posições mais à direita depois da normalização, para que estatisticamente o resultado seja mais preciso.

$\begin{array}{rcl} 1.23e0 & \xrightarrow[\text{ajusta}]{\text{expoente}} & 1.230e0 \\ 1.22e0 & & 1.220e0 \\ 1.??e-2 & & 0.010e0 \end{array}$

Figura 3-5: Falsa precisão.

Multiplicação e divisão não sofrem dos mesmos problemas da adição e subtração já que não necessitam ajuste de expoente antes da operação; tudo que é preciso fazer é adicionar os expoentes e multiplicar a mantissa (ou subtrair o expoente e dividir a mantissa). Multiplicação e divisão não produzem particularmente maus resultados. No entanto, eles tendem a multiplicar algum possível erro já existente nos operadores. Por exemplo, ao multiplicar 1.23e0 por dois, quando se queria multiplicar 1.24e0 por dois, o erro que era de 0.01, antes da multiplicação, passa a ser 0.02 depois da multiplicação (Figura 3-6).

$$(A + \xi) * 10 = 10A + 10\xi$$

Figura 3-6: Operação de multiplicação multiplica eventuais erros nos operandos.

Freqüentemente, aplicando transformações normais da álgebra, pode-se rearranjar uma equação de modo que as multiplicações e divisões ocorram primeiro. Por exemplo, supondo a realização do cálculo: $x*(y+z)$. Para isso, soma-se y com z e depois multiplica esse resultado por x . No entanto, tem-se mais exatidão no resultado final transformando a equação $x*(y+z)$ em $x*y+x*z$, e proceder aos cálculos realizando as multiplicações primeiro, evitando, assim, multiplicar o erro produzido pela adição (Figura 3-7).

$$A*(B + C) \rightarrow (A*B) + (A*C)$$

Figura 3-7: Rearranjando os operadores para evitar a multiplicação de possíveis erros.

Multiplicações e divisões também têm seus próprios problemas. Ao multiplicar dois valores muito grandes ou muito pequenos, é provável que ocorra *overflow* ou *underflow* (um resultado maior ou menor que os valores máximos e mínimos suportados). A mesma situação acontece ao dividir um pequeno valor por um grande valor ou ao dividir um grande valor por um pequeno valor. Esses casos precisam ser tratados em particular.

Comparação em ponto flutuante é muito perigosa. Valores trazem imprecisões com eles devido a algum cálculo anterior (incluindo a conversão de *strings* de entradas para o formato de ponto flutuante). Assim, não deve-se comparar dois valores de ponto flutuante para testar sua igualdade. Diferentes cálculos podem produzir o mesmo resultado (matematicamente), porém, podem diferenciar em algum dos seus dígitos significativos. Por exemplo, a soma de $1.31e0$ com $1.69e0$ deve resultar em $3.00e0$. Da mesma maneira, a soma de $1.50e0$ com $1.50e0$ deve produzir $3.00e0$. No entanto, a comparação de $(1.31e0+1.69e0)$ com $(1.50e0+1.50e0)$ não deve, necessariamente, retornar verdadeiro. O teste de igualdade só retornará verdadeiro, se e somente se, todos bits (ou dígitos) dos dois operadores forem iguais. Isto não é necessariamente verdade devido a arredondamentos e truncamentos que ocorrem durante os cálculos.

O caminho padrão para efetuar comparações entre dois operadores de ponto flutuante e determinar qual o erro (tolerância) permitido na comparação e verificar se os valores estão dentro desta faixa de erro. Por exemplo, para testar se *Valor1* é igual a *Valor2* com um erro de *Err*, procede da seguinte maneira: **if abs(Valor1-Valor2) <= Err then ...**

Assim, sempre deve ser levado em conta o erro ao realizar qualquer tipo de comparação: maior, menor, igual e diferente.

Este valor do erro precisa ser cuidadosamente escolhido. Ele deve ser maior que a soma dos erros produzidos nos cálculos. O valor exato dependerá do formato particular do ponto flutuante e da maneira que os cálculos são realizados.

Existem muitos outros pequenos problemas que podem surgir em situações particulares, assim, não se pode tratar a aritmética de ponto flutuante em sistemas digitais como a aritmética real [Koren, 1993],[Hwang, 1976],[Flores, 1963],[Goldberd, 1991],[Crowder, 1960].

3.2 PADRÃO IEEE 754

A fim de padronizar a representação e a implementação de operações que envolvem cálculo em ponto flutuante um grupo de trabalho da *IEEE Computer Society* foi formado e após muito trabalho foi definido o padrão *ANSI/IEEE Std 754-1985*. Este novo padrão tinha como objetivo facilitar a portabilidade de programas entre diferentes computadores.

3.2.1 REPRESENTAÇÃO BINÁRIA

A representação de um número em ponto flutuante na forma binária é apresentada na Figura 3-8, onde “s” representa o sinal do ponto flutuante (negativo igual a “1”), “y” representa o expoente, e “x” representa a mantissa.

Figura 3-8: Formato binário de um Número em Ponto Flutuante.

Item	Precisão simples	Precisão
Dupla		
Bits de Sinal	1	1
Bits de expoente	8	11
Bits de mantissa	23	52
Total de bits	32	64
Sistema do expoente	Excesso de 127	Excesso de 1023
Intervalo do expoente	-126 a +127	-1022 a 1023
O menor, normalizado	2^{-126}	2^{-1022}
O maior, normalizado	Aprox. 2^{+128}	Aprox. 2^{+1024}
Intervalo decimal	Aprox. 10^{-38} a 10^{+38}	Aprox. 10^{-308} a 10^{+308}
O menor, não normalizado	Aprox. 10^{-45}	Aprox. 10^{-324}

Figura 3-9: Padrão de Ponto-Flutuante IEEE 754.

Em números não-normalizados, o *bit* “1” implícito à esquerda do ponto binário agora se torna um “0”. Números não-normalizados podem distinguir-se dos normalizados, pois não é permitido aos números normalizados ter o expoente apenas de zeros.

Normalizado		
±	0 < Exp < Max	Qualquer padrão de bits
Não-Normalizado		
±	0	Qualquer padrão de bits Diferente de zero
Zero		
±	0	0
Infinito		
±	11111...1	0
Not-a-Number (Não é um número)		
±	11111...1	Qualquer padrão de bits diferente de zero

Figura 3-10: Tipo definidos pelo padrão de Ponto-Flutuante IEEE-754.

Dois zeros estão presentes neste esquema, positivo e negativo determinado pelo *bit* de sinal. Ambos possuem o expoente e a mantissa igual a zero. Aqui também o *bit* à esquerda do ponto binário é implicitamente zero em vez de um.

Quando ocorre um *overflow* é fornecida uma representação especial para infinito, que consiste em um expoente com todos os bits em “1” (não permitido para números normalizados) e uma mantissa igual a zero.

Este número pode ser usado como operando que se comporta como as regras matemáticas usuais de infinito. Por exemplo, infinito mais qualquer coisa é infinito. Similarmente, qualquer número dividido por zero resulta em infinito.

E infinito dividido por infinito? O resultado é indefinido. Para tratar deste caso usa-se outro formato especial, denominado NaN (*Not a Number*). Este pode também ser tratado como operando com resultados previsíveis [Goldblatt, 1998] .

4 EXPLORANDO O PARALELISMO

As tecnologias usadas em sistemas de computação tradicional (serial) podem ser agrupadas em três grandes categorias[Almasi,Gottlieb, 1994]:

- Circuitos lógicos usados em CPU e associados a funções computacionais;
- Circuitos armazenadores (memórias) usados pela CPU durante os cálculos;
- Dispositivos de armazenamentos para armazenamento de programas e dados.

Para processamento paralelo, no entanto, é preciso um conjunto adicional de funções para permitir uma eficiente comunicação interprocessos.

FPGA é uma importante tecnologia para construção de sistemas paralelos, sendo uma solução poderosa e com uma relação custo *versus* benefício, em muitos aspectos, superiores a outras soluções (dependendo da quantidade de unidade fabricada, quantidade de desenvolvedores envolvidos, tempo para desenvolvimento e implantação e características do mercado alvo), permitindo ao projetista adotar soluções não convencionais e especializadas para construções de máquinas paralelas, a fim de obter resultados mais rapidamente ou uma tomada de decisão mais eficiente (fundamental em caso de sistemas real-time).

No entanto, há uma grande diferença entre velocidade dos eventos que acontecem no nível dos transistores e a velocidade dos eventos no nível da aplicação do usuário. O tempo para executar uma instrução relevante para o usuário certamente atravessará diversos transistores e seu tempo dependerá da arquitetura implementada como será visto na próxima seção.

4.1 ATRASO DELTA

Os computadores eletrônicos sofreram enormes mudanças desde sua concepção, porém, compartilham uma importante variável de projeto: o atraso para passar a mudança de sinal para o próximo dispositivo lógico (conhecido como atraso delta). Este custo representa o tempo necessário para propagar a mudança de nível lógico (zero para um ou um para zero) entre dispositivos consecutivos. Este tempo tem diminuído dramaticamente com a implantação das novas tecnologias de construção de transistores. Os transistores desenvolvidos em 1994 já eram dez milhões de vezes mais rápidos em relação aos transistores construídos em 1940 e com um custo na ordem de centenas de vezes menores[Alamasi, Gottlieb, 1994].

As características elétricas das FPGAs, principalmente o atraso na saída do sinal de uma porta lógica, produzem um grande efeito na velocidade do sistema. Uma porta lógica possui uma resistência de 100ohms e uma capacitância parasitária de 10pF. Se, por exemplo, for ligado à saída de uma porta lógica na entrada de outras 19 portas lógicas, tem-se um total de $19 \times 10\text{pF} = 190\text{pF}$ de capacitância sendo descarregada através de 100ohms da resistência de porta. A constante de tempo dessa associação será $100\text{ohm} \times 190\text{pF} = 19\text{ns}$. O sinal de saída leva aproximadamente de duas a três vezes a constante de tempo para ir de 5V à 0V ou de 0V à 5V, o que representa um tempo entre 38 e 57 nanosegundos. Comparando este tempo de resposta da porta lógica com um PC de 200MHz que executa um ciclo de *clock* em $1/(200 \times 10^6)\text{s} = 5\text{ns}$, percebe-se como é grande o tempo de resposta para uma porta lógica que tenha sua saída ligada a muitas entradas. Isto mostra que é necessário limitar a quantidade de entradas numa mesma saída para obter circuitos rápidos.

Usualmente, t_{PHL} e t_{PLH} (Figura 4-1) são valores próximos, e são referenciados como t_{pd} . Quanto menor o t_{pd} mais caro será o dispositivo.

Figura 4-1: Atraso na propagação do sinal.

Quanto mais portas o sinal tiver que atravessar, mais tempo levará para produzir a saída, pois ocorre o acúmulo de atrasos. Na Figura 4-2 é mostrado um caminho potencial para o sinal de *carry* em um somador de três *bits* (*ripple-carry*) [Vandenbut, 1999]. Note que o sinal de *carry* é transportado do *bit* menos significativo ao *bit* mais significativo, aumentando seu atraso a cada estágio do somador, pois atravessa duas portas lógicas em cada estágio.

Uma maneira de melhorar o desempenho deste circuito seria computar o bit de *carry* em paralelo (*carry-lookahead*). Assumindo a soma de dois números binários $a_{n-1}, a_{n-2}, \dots, a_1, a_0$ e $b_{n-1}, b_{n-2}, \dots, b_1, b_0$. O *carry* será gerado e propagado seguindo as seguintes equações:

Seja: $G_i = a_i \text{ and } b_i$ e $P_i = a_i \text{ XOR } b_i$

$$\begin{aligned} c_0 &= G_0 + P_0 c_{in} \\ c_1 &= G_1 + P_1 c_0 \\ c_1 &= G_1 + P_1 (G_0 + P_0 c_{in}) \\ c_1 &= G_1 + P_1 G_0 + P_1 P_0 c_{in} \\ c_2 &= G_2 + P_2 c_1 \\ c_2 &= G_2 + P_2 (G_1 + P_1 G_0 + P_1 P_0 c_{in}) \\ c_2 &= G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_{in} \\ c_3 &= G_3 + P_3 c_2 \\ c_3 &= G_3 + P_3 (G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_{in}) \\ c_3 &= G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_{in} \end{aligned}$$

Figura 4-2: Acúmulo de atrasos no caminho do carry de um somador.

Desta maneira o somador sofrerá um atraso de quatro portas somente, independente da quantidade de bits dos operadores. O primeiro atraso acontece ao calcular G_i e P_i em paralelo, depois, acontece mais dois atrasos ao calcular os termos *AND* e *OR*, e finalmente o *carry* é usado para computar a soma dos bits.

Em dispositivos FPGA a granulosidade é maior, ou seja, não é possível tratar com portas lógicas e sim com CLBs (Blocos Lógicos Configuráveis). Da mesma maneira, o sinal leva uma constante de tempo para sair de um CLB e entrar em outro ou ser apresentado num pino de saída da FPGA para ser usado externamente. Como se pode deduzir: circuitos que tenham um caminho crítico muito grande (maior caminho percorrido pelo sinal desde a entrada até a saída do circuito) têm um tempo de resposta proporcionalmente. O caminho crítico é limitante para a velocidade geral do circuito, pois, o sinal do relógio (*clock*) é o mesmo para todo o circuito, sendo assim, o período do sinal do relógio precisa ser maior que o caminho crítico do circuito.

Para aumentar a velocidade do sinal do relógio é usual dividir o circuito em pequenos circuitos, executados sequencialmente, sincronizados pelo sinal de relógio. Para aumentar a eficiência do sistema são adotados estágios de *pipeline* que se assemelham a uma linha de produção de uma fábrica. Esta técnica será discutida na próxima seção.

4.2 PARALELISMO TEMPORAL

Pipeline é uma técnica bem conhecida para acelerar a execução de operações sucessivas e idênticas. Ao invés de construir um circuito capaz de executar uma simples operação ou um conjunto de operações em um determinado tempo, dividi-se esta tarefa, criando-se sub-circuitos que podem executar, de forma independente, um conjunto de operações consecutivas [Stallings, 2000]. Assim, várias operações são executadas em paralelo, e a taxa em que os resultados são produzidos é consideravelmente maior que uma arquitetura que não utiliza *pipeline*.

Para permitir *pipeline*, o algoritmo é dividido em alguns passos e um circuito apropriado é construído para cada um desses passos [Koren, 1993],[Cavanagh, 1985]. Cada sub-circuito, que é chamado de um estágio de *pipeline*, precisa executar seu conjunto de instruções de forma independente. Para alcançar este objetivo é preciso adicionar um conjunto de registradores entre estágios adjacentes. Assim, enquanto um estágio trabalha com um conjunto de operadores, o estágio anterior pode processar o próximo conjunto de operadores.

Utilizando-se uma operação com três estágios, apresentada na Figura 4-3 para exemplificar o uso de *pipeline*, o *latch* (busca e retenção de dados de entrada) entre o estágio um e o estágio dois, armazena resultados intermediários do estágio um, que estão sendo usados para o processo do estágio dois, enquanto o estágio um está processando um novo conjunto de operadores.

Figura 4-3: Pipeline com três estágios.

Analisando-se o diagrama de tempo apresentado na Figura 4-4, tem-se os tempos para execução de quatro conjuntos de entrada: e1, e2, e3, e4 produzindo como resultado: s1, s2, s3, s4, respectivamente.

Figura 4-4: Diagrama de um pipeline com três estágios

Pode ser observado na Figura 4-4, que o tempo gasto para execução em cada estágio precisa ser o mesmo para todos os estágios. De outra maneira, os estágios de execução mais rápida precisariam esperar os de execução mais lenta terminarem seu processamento, para então ser alimentado o *latch* e processo se inicia novamente.

Uma importante decisão a ser tomada é o particionamento de um dado algoritmo em passos que poderão ser executados em estágios de *pipeline*. Estes estágios preferivelmente precisam ter o mesmo tempo de execução, tendo-se em vista que a taxa do *pipeline* é determinada pelo estágio mais lento. O número de estágios precisa ser determinado: quanto mais estágios houver maior será a taxa de saída do *pipeline*, e proporcionalmente crescerá também o custo de implementação e a latência do *pipeline*. Latência é o tempo gasto até que o primeiro resultado seja produzido pelo *pipeline*. Este é particularmente importante quando se quer executar apenas uma operação. Assim, tem-se de um lado da balança a latência e o custo de implementação e, do outro lado, a taxa de saída do *pipeline*.

4.3 PARALELISMO ESPACIAL

Pode-se alcançar o processamento paralelo na construção de processos em FPGA, simplesmente duplicando o circuito necessário tantas vezes quanto se queira, permitindo, desta maneira, que cada bloco contendo o mesmo processo execute de forma totalmente independente[Leung, Shanblatt, 1997]. A vantagem na utilização desta técnica é a facilidade com que se pode aumentar a quantidade de operações realizadas, e a desvantagem é que a complexidade de *hardware* cresce linearmente com o aumento das operações.

Figura 4-5: Implementação de dois acumuladores seriais.

Esta técnica é especialmente útil quando há um gargalo no processamento. Tomando-se como exemplo dois acumuladores que recebem dados assíncronos de duas fontes distintas e utilizando-se de apenas uma ULA para realizar a soma dos dois acumuladores, faz-se necessário a serialização dos dados para a ULA. E com isso podem ocorrer perdas de alguns dados que não conseguiram ser somados e foram sobrescritos por novos dados. Uma maneira de contornar este problema é construir um circuito auxiliar que armazene estes dados antes de mandá-los para a ULA (ver Figura 4-5). Dependendo da velocidade em que novos dados cheguem e a velocidade de soma da ULA, pode ocorrer um gargalo neste processo. Uma solução para resolver este gargalo seria a duplicação do processo de soma da ULA, deixando-a uma para cada acumulador (Figura 4-6).

Figura 4-6: Implementação de dois acumuladores em paralelo.

Pode-se também estudar a possibilidade de não duplicar parte do processo, especialmente partes que ocupam muito espaço em silício e que tenham processamento rápido em relação ao tempo total do processo. Assim, o processo perde um pouco em velocidade, pois os acessos às partes não duplicadas precisam ser serializados, mas em compensação, ganha-se em espaço de silício no *hardware*.

4.4 PARALELISMO ESPACIAL E TEMPORAL

Para se conseguir maior eficiência nos cálculos, utilizam-se os métodos descritos na seção anterior: paralelismo temporal e paralelismo espacial, na proporção em que o sistema que os utilizarão exigir. A Figura 4-7 exemplifica a utilização das duas abordagens na construção de um somador em ponto flutuante, extraído de [Stallings, 2000]. É possível observar o quanto se ganha em tempo de resposta quando se trabalha com uma sequência de operações. Porém, têm-se as desvantagens já mostradas neste capítulo. Deste modo, é importante ressaltar aqui a necessidade de parametrizar a construção da ULA, afim de não desperdiçar espaço ou tempo de processamento que sobrecarregaria o custo total do projeto.

Quando as operações aritméticas requeridas pela aplicação não correspondem exatamente às fornecidas pela ALU/FPU dos processadores convencionais ou quando as operações admitem um alto grau de paralelismo *bit-level*, há um amplo espaço para utilização de *hardware* reconfigurável para obter-se altos desempenhos.

Figura 4-7: Implementação de uma FPU paralela.

4.5 TÉCNICAS PARA PROJETO DE PIPELINE

Pipeline pode ser classificado como unifuncional, quando é dedicado a apenas uma operação, ou multifuncional, quando podem executar mais de uma operação compartilhando estágios comuns entre operações distintas. *Pipeline* multifuncional necessita de um controlador que deve direcionar os dados entre as interconexões dos estágios do *pipeline* de acordo com a função executada [Stone, 1990]. Esta seção resume uma técnica originalmente proposta por Davidson em 1971. A idéia principal desta técnica é construir um controlador simples que manipule as entradas de um *pipeline* complexo. Este controlador permite que o *pipeline* alcance o máximo de *throughput*, enquanto garante que não haverá colisões de dois ou mais operadores dentro de um estágio do *pipeline*.

A seguir será demonstrado o uso desta técnica na construção de uma unidade aritmética de ponto flutuante.

4.5.1 PROJETO DE PIPELINE COM MÚLTIPLAS FUNÇÕES

Primeiramente, serão examinadas rapidamente a multiplicação e adição em ponto flutuante, a fim de verificar uma aproximação razoável entre os estágios de *pipeline* destas duas operações. A operação mais simples de descrever é a multiplicação, e consiste em:

1. Assume-se que os valores de entrada são dois números em ponto flutuantes normalizados representados por $(\text{Mantissa}_1, \text{Expoente}_1)$ e $(\text{Mantissa}_2, \text{Expoente}_2)$, respectivamente. O primeiro passo consiste em somar os dois expoentes para formar o $\text{Expoente}_{\text{out}}$.
2. Multiplicam-se as duas mantissas para formar um resultado duas vezes maior do que o tamanho da mantissa. Este passo deve ser executado em paralelo com a adição dos expoentes, e pode demorar algumas unidades de tempo de execução se a multiplicação for projetada para ser executada em multiplicações sucessivas.
3. Examina-se o produto das mantissas e se não estiver normalizado, normaliza-se ajustando também o expoente.
4. Arredonda-se o produto das mantissas, e se houver *overflow*, ajusta-se o expoente.

Na Figura 4-8 é mostrado o diagrama de um *pipeline* linear e um *pipeline* com reentrada de dados entre estágios. A Figura 4-8b mostra a multiplicação das mantissas dividida em dois processos:

1. Produção de uma coleção de produtos parciais;
2. Adição dos produtos parciais.

Desta maneira, pode-se dividir a multiplicação das mantissas em dois estágios: o primeiro, produz todos os resultados parciais; e o segundo, soma estes produtos parciais. Porém, na prática esta forma de multiplicação requer muito hardware e, a fim de reduzir a complexidade, a multiplicação é dividida em mais estágios. Por exemplo, as mantissas são divididas em quatro partes, e cada uma destas partes forma um produto parcial que, combinados, geram o produto total dos dois operadores. Assim, a multiplicação é dividida em quatro estágios que envolvem a geração da coleção de produtos parciais e a adição destes para formar o resultado final. O tempo gasto para execução da multiplicação aumenta no fator quatro. Porém, o hardware requerido diminui num fator aproximadamente quatro. Obviamente, é uma decisão de projeto: diminuir o tempo de execução ou diminuir o hardware requerido.

Figura 4-8: (a) Pipeline linear para multiplicação em ponto flutuante. (b) Pipeline para multiplicação em ponto flutuante com reentrada de dados.

A adição em ponto flutuante pode ser descrita nos seguintes passos.

1. Assume-se que os valores de entrada são dois números em ponto flutuantes normalizados representados por $(\text{Mantissa}_1, \text{Expoente}_1)$ e $(\text{Mantissa}_2, \text{Expoente}_2)$, exatamente como na multiplicação descrita anteriormente. Subtrai-se o menor expoente do maior, a fim de encontrar a diferença entre eles ($\text{Expoente}_{\text{diff}}$).
2. Desloca-se a mantissa do operador com expoente menor para direita o número de posições dado por $\text{Expoente}_{\text{diff}}$.
3. Somam-se as mantissas e atribui o valor do expoente maior para o expoente da soma.
4. Normaliza-se a soma das mantissas, ajustando também o expoente.
5. Arredonda-se a soma das mantissas, e se houver *overflow*, ajusta-se o expoente.

Na Figura 4-9 está representada uma estrutura de *pipeline* para adição em ponto flutuante que utiliza reentrada de dados para a operação de deslocamento da mantissa. A operação de deslocamento requer muito hardware para ser executada em um único ciclo. Deste modo, esta operação foi dividida em alguns ciclos. Mais uma vez, é uma decisão de projeto - em quantos ciclos dividir esta operação? - menor tempo, requer mais hardware - mais ciclos, consomem mais tempo.

Figura 4-9: Pipeline para adição em ponto flutuante com reentrada de dados.

Comparando-se o modelo de *pipeline* para adição em ponto flutuante com o modelo de *pipeline* para multiplicação em ponto flutuante, podem ser destacadas algumas operações em comum: operação de adição/subtração/incremento para o expoente; operação de adição da mantissa; e finalmente as operações de normalização e arredondamento também podem ser compartilhadas.

A Figura 4-10 mostra a estrutura de *pipeline* depois de combinados as operações de adição e de multiplicação. Existem dois caminhos que podem ser seguidos pela operação, dependendo da natureza desta operação. Esta estrutura requer um cuidadoso controle de colisões. Para resolver o problema de colisão de dados, é necessário desenvolver um diagrama de tempo da informação percorrendo o *pipeline* da unidade aritmética. No exemplo de *pipeline* exposto aqui, admitindo-se que a multiplicação e o deslocamento são executados em dois ciclos, pode-se construir a tabela de tempo *versus* execução. Cada linha desta tabela representa uma mini-operação, ou um estágio do *pipeline*, e cada coluna representa uma unidade de tempo do *pipeline* (ver Figura 4-11). Esta tabela também é conhecida pelo nome de Tabela de Reserva.

Figura 4-10: Pipeline da adição e multiplicação em ponto flutuante.

Desta tabela é obtido o vetor de colisões que servirá para o controle de colisões, como mostrado a seguir. O vetor de colisões informa quantos ciclos de espera são necessários depois que se inicia uma determinada operação. Estes ciclos de espera garantem que não ocorrerão colisões de dados. O vetor de colisões é representado por “0” e “1”; “1” indica que irá ocorrer uma colisão e “0” indica que não haverá colisão. Por exemplo, o vetor “110000” indica que os dois próximos ciclos após o início de uma operação deve ser de espera, pois, ocorrerá colisão. Utilizando-se vetor de colisões é possível criar um controlador para o *pipeline* que impeça colisões de forma simples e de execução rápida. Este controlador consiste basicamente em deslocar o vetor de colisões para a esquerda a cada ciclo, e executar um “OU lógico” com um novo vetor de colisões quando se inicia uma operação. Uma nova operação só poderá ser iniciada quando o primeiro bit do vetor de controle de colisões for zero.

Estágio	T1	T2	T3	T4	T5	T6	T7
Subtrai/Adiciona expoentes	(A)(M)						
Deslocamento parcial		(A)	(A)				
Multiplicação parcial		(M)	(M)				
Soma das mantissas				(A)(M)	(A)(M)		
Normalização						A(M)	
Arredondamento							A(M)

Figura 4-11: Tabela de tempo para pipeline multifuncional. (A) Adição; (M) Multiplicação.

4.5.2 MEDIDAS DE DESEMPENHO PARA PIPELINE

Para se medir o desempenho obtido com o emprego das técnicas de paralelismo descritas anteriormente neste capítulo utiliza-se os seguintes parâmetros [Koren, 1993], [Stallings, 2000]: *speedup* (fator de aceleração), eficiência e *throughput* descritas a seguir.

4.5.2.1 SPEEDUP (FATOR DE ACELERAÇÃO)

Speedup expressa o ganho em velocidade que se obtém ao executar um grupo de tarefas em um arranjo paralelo comparada com sua execução sequencial. Este fator de aceleração é obtido dividindo-se o tempo necessário para execução sequencial de um algoritmo pelo tempo necessário para execução do mesmo algoritmo em um arranjo paralelo. Quanto maior for o fator de aceleração de uma implementação maior será sua eficiência.

4.5.2.2 EFICIÊNCIA

Eficiência relaciona o valor do *speedup* (fator de aceleração) com a quantidade de elementos implementados em paralelo, expressando assim, a porcentagem média de aproveitamento dos recursos disponíveis.

4.5.2.3 THROUGHPUT

Throughput expressa o número de operações executadas em uma implementação por unidade de tempo, ou seja, a quantidade de resultados gerados pelo circuito por unidade de tempo.

5 IMPLEMENTAÇÃO

A fim de aumentar a reusabilidade, facilitar o desenvolvimento, teste e manutenção, a FPU está dividida em blocos hierárquicos, onde cada um deles tem uma interface e uma função bem definida e são disponibilizados em forma de componente. A seguir são detalhados os blocos mais importantes para o entendimento da arquitetura interna da FPU.

A separação do projeto em blocos hierárquicos, além de possibilitar múltiplas visões da arquitetura interna da FPU, também facilita a customização da FPU para um domínio específico do problema. No capítulo 6 será mostrada uma implementação real da FPU em uma aplicação em robótica.

A seguir são detalhados os componentes principais na arquitetura da FPU e ao final deste capítulo tem uma breve explicação de todos os componentes desenvolvidos.

5.1 BLOCO 'FPU_DESIGN' (COMPONENTE PRINCIPAL)

Este bloco implementa as três funções básicas da FPU, adição, multiplicação e divisão em ponto flutuante, recebendo como entrada dois operadores em ponto flutuante, três bits de instrução e retornando o resultado da operação requisitada.

5.1.1 INTERFACE

Figura 5-1: Interface do componente principal da FPU – ‘FPU_Design’.

O componente ‘FPU_Design’ têm quatro parâmetros de configuração, a saber:

- LEN_SIGNAL - quando igual a um, indica que primeiro bit do operador representa o sinal (positivo/negativo). Quando igual a zero, indica que o sinal será representado;
- LENGHT_EXP – quantidade de bits do expoente dos operadores;
- LENGHT_MAN – quantidade de bits da mantissa dos operadores;
- LENGHT_FP – quantidade total de bit do operador.

Além dos parâmetros de configuração, existem seis sinais de entrada:

- i_clock – *clock* global da FPU;
- i_reset – sinal de *reset* assíncrono;
- i_clear_exception – quando em nível alto, força os sinais de exceção para nível baixo;
- i_fp_a – primeiro operador;
- i_fp_b – segundo operador;
- i_fpu_instr – três bits de instrução.

i_fpu_instr	Instrução executada
001	Adição
010	Subtração
011	Multiplicação
100	Divisão

5.1.2 IMPLEMENTAÇÃO

A implementação do componente ‘FPU_Design’, que representa toda a unidade aritmética, é composta por três grandes blocos: ‘FADD_PIPE’, componente responsável pela adição em ponto flutuante; ‘FMULT_PIPE’, componente responsável pela multiplicação em ponto flutuante; ‘FDIVI_PIPE’, componente responsável pela divisão em ponto flutuante. Os três componentes são desenvolvidos usando *pipelines* e serão detalhados a seguir. Na Figura 5-2 é mostrado detalhes da interligação entre os componentes e na Figura 5-3 é apresentado os componentes em forma hierárquica.

Figura 5-2: Esquemático do componente 'FPU_Dsign'.

Figura 5-3: Diagrama hierárquico dos componentes para operação em ponto flutuante.

5.2 ADIÇÃO DE INTEIRO

O componente responsável pela adição de inteiro sem sinal é muito utilizado como parte de diversos processos nos algoritmos de cálculo para números em ponto flutuante, principalmente para soma parcial das mantissas e no incremento do expoente. Por sua importância ele foi destacado aqui, seu nome é 'ADD' e sua interface e implementação é mostrada a seguir na Figura 5-4. Há uma variação deste componente chamado 'ADDC', que tem um sinal a mais de *carry*. A listagem completa dos componentes desenvolvidos é apresentada no final deste capítulo.

5.2.1 INTERFACE

Figura 5-4: Interface do componente que executa adição de inteiro – 'ADD'.

O parâmetro LENGHT_OP configura o tamanho dos operadores de entrada I_NUMBER_A e I_NUMBER_B e de saída O_RESULT.

5.2.2 IMPLEMENTAÇÃO

A adição de n bits é feita utilizando o componente 'ADD_BIT' para cada operador de entrada e encadeando o bit de *carry*. A Figura 5-5 mostra este arranjo em forma de diagrama esquemático e seu correspondente em código VHDL.

Figura 5-5: implementação do componente ADD.

5.3 ADIÇÃO EM PONTO FLUTUANTE

O componente de adição de ponto flutuante 'FADD_PIPE' pode ser utilizado diretamente pela aplicação. Este tem uma interface simplificada em relação ao componente 'FPU_Design' que o utiliza.

Uma variação deste componente tem como entrada os operadores já decompostos e com seu estado informado, ou seja, sua entrada seria a saída do componente 'FP_PARSE' e sua saída seria a entrada do componente 'FP_UNION'. Esta variação busca otimizar o espaço quando utilizado com o componente de multiplicação ou divisão de ponto flutuante.

5.3.1 INTERFACE

Requer parâmetros de configuração de número de bits. Tem como entrada um sinal de *clock* e dois operadores em ponto flutuante e como saída o resultado da adição e o sinal de *overflow*.

Figura 5-6: Interface do componente para adição em ponto flutuante – ‘FADD_PIPE’.

5.3.2 IMPLEMENTAÇÃO

O componente ‘FADD_PIPE’ utiliza o componente ‘FSIMPLE_ADD_NORMALIZE_PIPE’, detalhado a seguir, para obter o resultado da adição já normalizado.

O componente ‘FP_PARSE’ analisa o operador, verificando se este não contém uma representação especial da especificação IEEE-754, e decompõe este em sinal, expoente, dígito à esquerda da vírgula (bit implícito) e mantissa.

A Figura 5-7 mostra o diagrama esquemático do componente ‘FADD_PIPE’ e a Figura 5-8 mostra o diagrama hierárquico.

Figura 5-7: Implantação do componente 'FADD_PIPE'.

Figura 5-8: Diagrama hierárquico dos componentes da adição em ponto flutuante.

5.3.3 COMPONENTE 'FSIMPLE_ADD_NORMALIZE_PIPE'

Este componente realiza a operação de adição em dois estágios: no primeiro estágio utiliza o componente 'FSIMPLE_ADD_PIPE' para realizar a adição sem normalização de resultado; e no segundo estágio utiliza o componente 'NORMALIZE' para normalizar o resultado.

Figura 5-9: Implementação do componente 'FSIMPLE_ADD_NORMALIZE_PIPE'

5.3.4 IMPLEMENTAÇÃO DO COMPONENTE 'FSIMPLE_ADD_PIPE'

O componente 'FSIMPLE_ADD_PIPE' utiliza um estágio de *pipeline* para alinhar o expoente, ou seja, executar *shift* à direita na mantissa do menor operador enquanto incrementa o expoente até os expoentes dos operadores se igualarem ou a mantissa ou o expoente do menor operador se for a zero.

No segundo estágio de *pipeline* é executada a adição e a subtração da mantissa em paralelo, e se o sinal dos operadores forem iguais o resultado utilizado será da adição, caso contrario, o resultado da subtração da maior mantissa pela menor mantissa será utilizado. No final da adição é utilizado um componente para fazer o alinhamento do *carry* caso seja necessário. Este alinhamento é feito deslocando a mantissa com o *carry* um bit à direita e incrementando um ao expoente.

Figura 5-10: Implementação do componente 'FSIMPLE_ADD_PIPE'

5.4 MULTIPLICAÇÃO

Multiplificação de inteiros sem sinal é utilizada para multiplicar mantissas em operações de multiplificação em ponto flutuante.

5.4.1 INTERFACE

Figura 5-11: Interface de multiplificação – ‘FMULT’.

Recebe como entrada dois números inteiros positivos de tamanho indicado pelo parâmetro LENGHT_OP, e tem como saída um número inteiro positivo com tamanho especificado no parâmetro LENGHT_RESULT onde será retornado o resultado da multiplificação entre os operadores de entrada.

Normalmente o operador de saída tem o dobro de bits dos operadores de entrada para que não haja *overflow*, porem, em casos específicos, onde se conhece bem o domínio dos números utilizados, pode ser sabido de antemão que o resultado desta multiplificação intermediaria nunca resultará em *overflow* se for utilizado um sinal de saída com tamanho menor. Isso economizará espaço em hardware, que se tratando de FPGA ainda é um fator crucial para o sucesso de uma implementação.

5.4.2 IMPLEMENTAÇÃO

O algoritmo consiste em acrescentar zeros à esquerda (*shift*) do Multiplicador e do Produto inicialmente zerado, até que o ‘*carry*’ seja um. Quando isto acontecer soma-se o Multiplicador ao Produto colocando-se o resultado obtido no próprio Produto. Volta-se a rotacionar o Produto bem como o Multiplicador, até que o contador do número de bits (tamanho do Multiplicador) seja zero.

Esta implementação executa a multiplicação entre dois inteiros sem sinal utilizando componentes 'ADD' encadeado em forma de *ripple-carry-adder*. Fica com o sintetizador a responsabilidade de otimizar o código para *lock-carry-ahead*.

Figura 5-12: Implementação do componente para multiplicação de inteiro.

5.5 MULTIPLICAÇÃO EM PONTO FLUTUANTE

A multiplicação em ponto flutuante é realizada pelo componente 'FMULT_PIPE', que segue o mesmo conceito de implementação do componente de adição 'F_ADD_PIPE' já descrito.

5.5.1 INTERFACE

Figura 5-13: Interface do componente de multiplicação em ponto flutuante - 'FMULT_PIPE'.

O componente 'FMULT_PIPE' tem quatro parâmetros de configuração e quatro sinais de entrada, são eles:

- *i_clock* – *clock* global da FPU;
- *i_reset* – sinal de *reset* assíncrono;
- *i_fp_a* – primeiro operador;
- *i_fp_b* – segundo operador;

E como saída oferece o resultado e os sinais que indicam exceção.

5.5.2 IMPLEMENTAÇÃO

A implementação do componente 'FMULT_PIPE' é simples e deixa toda a responsabilidade para componentes secundários, como pode ser visto na figura a seguir. O componente 'FSIMPLE_MULT', tem o objetivo de multiplicar a mantissa, somar os expoentes e alinhar a mantissa caso a multiplicação resulte em duas casas a esquerda da vírgula. E o componente 'NORMALIZE' faz a normalização do resultado caso necessário.

Figura 5-14: Implementação do componente de multiplicação em ponto flutuante.

A Figura 5-14 mostra o diagrama esquemático e a Figura 5-15 mostra o diagrama hierárquico do componente 'FMULT_PIPE' responsável pela multiplicação em ponto flutuante.

Figura 5-15: Diagrama hierárquico dos componentes da multiplicação em ponto flutuante.

5.6 COMPONENTE 'FSIMPLE_MULT'

Este componente é responsável pela multiplicação da mantissa, soma dos expoentes e alinhamento da mantissa caso a multiplicação resulte em duas casas a esquerda da vírgula. Como o expoente está com excesso, no caso de número em ponto flutuante com 32 bits no padrão IEEE 754, o excesso é de 127. Assim, após a soma dos expoentes o resultado sairia com excesso de 254, precisando então uma subtração de 127.

Figura 5-16: Esquemático da implementação do componente 'FSIMPLE_MULT'.

5.7 DIVISÃO DE INTEIRO

O algoritmo consiste em girar, juntamente com o 'carry' o Dividendo dentro do Resto, estando este inicialmente zerado. Um contador do número de Bits (tamanho do Dividendo) deverá ser decrementado cada vez que for realizada uma rotação.

Se o resto for maior ou igual ao divisor, o 'carry' será igual a um, sendo este valor na próxima rotação posicionado no bit 2º do dividendo, toda vez que isto acontecer – o resto deve ser atualizado, subtraindo o valor do Divisor.

O final da próxima divisão irá ocorrer quando o contador de Bits for igual a zero, porém ainda é preciso posicionar o último 'carry' dentro do Dividendo, sem rotacionar o Resto.

Figura 5-17: Implementação do componente de divisão de inteiro.

5.8 DIVISÃO EM PONTO FLUTUANTE

O componente 'FDIVI_PIPE' utiliza o componente 'FSIMPLE_DIVI' para realizar a divisão dos operadores em ponto flutuante, e fica sobre sua responsabilidade checar condições de exceção.

Figura 5-18: Esquemático do componente para divisão em ponto flutuante - 'FDIVI_PIPE'.

Na Figura 5-18 é apresentado o diagrama esquemático onde pode-se ver as interconexões entre os componentes, e na Figura 5-19 é apresentado o diagrama hierárquico de componentes. Analisando os três diagramas Figura 5-8, Figura 5-15 e Figura 5-19 se percebe o quanto de código foi possível reaproveitar.

Figura 5-19: Diagrama hierárquico dos componentes da divisão em ponto flutuante.

5.9 NORMALIZAÇÃO

Este componente normaliza o número em ponto flutuante que tenha a mantissa menor que um, ou seja, tenha o bit à esquerda da virgula igual a zero. Neste caso, decrementa-se o expoente e desloca-se a mantissa à esquerda até que o primeiro bit setado alcance o bit à esquerda da virgula, ou o expoente chegue no zero.

No caso do expoente ser maior que zero a saída 'o_normalized' é setada, representando que o número foi normalizado.

5.9.1 IMPLEMENTAÇÃO

Esta implementação executa o decremento do expoente usando o componente 'SUBTRACT'. Para decidir qual será o valor total do decremento, utiliza-se um multiplexador que informará em sua saída a posição do primeiro bit na mantissa.

E para deslocar a mantissa, é utilizado outro multiplexador. Tomando o cuidado quando há *underflow* no expoente, neste caso, o expoente recebe zero e a mantissa é deslocada à esquerda no valor inicial do expoente (Figura 5-20).

Figura 5-20: Diagrama esquemático do componente de normalização.

5.10 LISTAGEM DOS COMPONENTES

Abaixo é apresentada a relação completa dos componentes desenvolvidos e utilizados na implementação da FPU.

Nome do componente	Descrição
ADD	Somador de n bit.
ADD_BIT	Somador de um bit.
ADDC	Somador de n bit com <i>carry</i> .
ALIGN_EXPONENT	Alinhamento do expoente.

ALIGNMENT_CARRY.	Alinhamento à direita da mantissa, incrementando o expoente se necessário.
AND_BIT_BY_BIT	Executa a função <i>and</i> entre todos os n bits do operador informado.
DIVI_FRACTION	Divisão entre operadores inteiros. Retorna resultado em ponto flutuante, apenas com um bit no expoente.
DIVI_T	Divisão entre operadores inteiros. Retorna resultado em ponto flutuante.
FADD_PIPE	Adição em ponto flutuante.
FDIVI_PIPE	Divisão em ponto flutuante.
FMULT_PIPE	Multiplicação em ponto flutuante.
FP_PARSE	Analisa um número em ponto flutuante e o decompõe em expoente, bit implícito, mantissa e indicadores de estado (<i>not-a-number</i> , zero, infinito, normalizado)
FP_UNION	Faz a tarefa contrária do componente 'FP_PARSE', recebendo as partes do número em ponto flutuante e seus indicadores de exceção, retornando um número em ponto flutuante no padrão adotado.
FPU_DESIGN	União de todas as características desejadas para a FPU. Funciona como um co-processador aritmético.
FSIMPLE_ADD_NORMALIZE_PIPE	Recebe os operadores já decompostos, efetua a adição em ponto flutuante e retorna o resultado normalizado, porém, também decomposto.
FSIMPLE_ADD_PIPE	Recebe os operadores já decompostos, efetua a adição em ponto flutuante e retorna o resultado.
FSIMPLE_DIVI	Recebe os operadores já decompostos, efetua a divisão em ponto flutuante e retorna o resultado.
FSIMPLE_MULT	Recebe os operadores já decompostos, efetua a multiplicação em ponto flutuante e retorna o resultado.
INCREMENT	Incrementa em um o operador inteiro.
MULT	Multiplicação de inteiro.
NORMALIZE	Normaliza um número em ponto flutuante.
OR_BIT_BY_BIT	Executa o OR bit a bit sobre o operador.
PART_DIVI	Usado no componente 'DIVI_T'. Divisão de inteiros, retorna o quociente e o resto separados.

	retorna o quociente e o resto separados.
SHIFT_RIGHT	Executa o <i>shift</i> à direita n posições sobre o operador.
SUB_BIT	Subtrator de um bit.
SUBTRACT	Subtrator de n bits.
SUBTRACT_TWO_WAY	Dois subtratores de n bits. Retorna A-B e B-A.

6 VALIDAÇÃO E EXEMPLO DE APLICAÇÃO

Para validar a FPU foi construído um *Testbench* para cada operação aritmética. Um programa escrito em C gerou a base de teste, um arquivo texto contendo na primeira coluna o primeiro operador, na segunda coluna o segundo operador e na terceira o resultado da operação. Estes números foram convertidos em uma string de caracteres de 32 posições contendo ‘0’ ou ‘1’, que representa um número em ponto flutuante no padrão IEEE 754 (Figura 7-1). Os operadores foram escolhidos de forma aleatória.

Operador A	Operador B	Resultado
01000010110001011100110011001101 $1.5453125 \times 2^6 = 98,9$	01000001010000011110101110000101 $1.5150000 \times 2^3 = 12,12$	01000010110111100000101000111110 $1.7346876 \times 2^6 = 111,02$
01000010110000111001100110011010 $1.5281250 \times 2^6 = 97,8$	01000001011000111101011100001010 $1.7800000 \times 2^3 = 14,24$	01000010111000000001010001111011 $1.7506250 \times 2^6 = 112,04$
01000010110000010110011001100111 $1.5109376 \times 2^6 = 96,7$	010000011000000101110000101001000 $1.0225000 \times 2^4 = 16,36$	01000010111000100001111010111001 $1.7665626 \times 2^6 = 113,06$
...	...	...

Figura 6-1: Formato do arquivo gerado para testar o componente de adição da FPU.

Utilizando o *software* da Aldec, *Active HDL 5.1*, foi simulado o código da FPU com os dados do arquivo de teste. O operador A e o Operador B foram usados como entrada da FPU e o resultado foi armazenado para posterior comparação. O código do *Testbench* é mostrado a seguir:

```

while NOT(endfile(INFILE)) loop
    readline(INFILE,IN_LINE);
    read(IN_LINE,A_FROM_FILE);
    read(IN_LINE,B_FROM_FILE);
    read(IN_LINE,S_FROM_FILE);
    i_fp_a <= To_StdLogicVector(A_FROM_FILE);
    i_fp_b <= To_StdLogicVector(B_FROM_FILE);
    S_FROM_FILE := To_bitvector(o_fp_s);
    write(IN_LINE, S_FROM_FILE, right, 32);
    writeline(RESULTS,IN_LINE);
end loop;

```

Figura 6-2: Código do testbench para FPU.

O arquivo de saída gerado pelo *testbench* tem o mesmo formato numérico do arquivo de entrada e contém somente o resultado da operação. Em uma segunda etapa é feita a comparação entre o resultado obtido no código em C, rodando em um Pentium III de 866 MHz e o resultado obtido com a FPU. Setenta por cento dos resultados se igualaram enquanto que em quarenta por cento dos resultados ocorreram diferenças de arredondamento. Essas diferenças no arredondamento podem ser cruciais para alguns tipos de aplicações, porém, podem ser minimizadas com as implementações de arredondamento discutidas no Capítulo III.

6.1 EXEMPLO DE APLICAÇÃO

Como aplicação prática foi utilizado um trabalho de mestrado do LCR que pretendia reescrever um algoritmo de campos potenciais para navegação de robô, originalmente escrito em C, para a linguagem VHDL e implementá-lo no FPGA da *Altera* (*Apex*) utilizando o kit *Excalibur* do mesmo fabricante.

Esta aplicação foi escolhida pelo fato de ter um elevado custo computacional na realização de duas equações trigonométricas em ponto flutuante. São elas:

$(1) F_x = F_x + (-f * (\cos(t_i) * r_i) - v * \sin(t_i)) / (r_i^2);$ $(2) F_y = F_y + (-f * (\sin(t_i) * r_i) + v * \cos(t_i)) / (r_i^2);$
<ul style="list-style-type: none"> ○ F_x, F_y – somatório das forças atrativas e repulsivas que dão o ângulo de movimento do robô. ○ f – Força de repulsão exercida por cada obstáculo. ○ t_i – ângulo do obstáculo com relação ao robô. ○ r_i – distância do obstáculo ao alvo. ○ v – vorticidade exercida por cada obstáculo.

Essas equações são executadas em laço para cada movimento do robô e resultaram no próximo ângulo de movimento do robô. As funções seno e co-seno serão resolvidas usando série de *Taylor*, da seguinte maneira:

$$(3) \text{ Sen } x = \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} \dots$$

$$(4) \text{ Cos } x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} \dots$$

A aplicação descrita acima foi implementada em *hardware* reconfigurável utilizando o processador *Softcore Nios* [Altera, 2003] e como co-processador aritmético a FPU construída neste mestrado, pois o processador *Nios* emula as operações em ponto flutuante em *software*, o que torna lento este tipo de operação. Foi gerada uma tabela de comparação de tempo de execução e ciclo de *clock* para o mesmo algoritmo e a mesma configuração de entrada sendo executada no Pentium III, no *Nios* com emulação de unidade de ponto flutuante e no *Nios* utilizando a FPU como co-processador aritmético.

Figura 6-3: Interligação entre o Nios e a FPU.

O processador *Softcore Nios* foi ligado a FPU por três canais de dados de 32 bits cada e um canal de quatro bits de instrução (Figura 6-3). Os canais de dados, que representavam o número em ponto flutuante no padrão IEEE 754, recebem a imagem de bits que representa o número em ponto flutuante no registrador do processador *Nios*, pois este também representa o número no padrão IEEE 754. Assim, nenhuma conversão se faz necessária.

A única preocupação para este tipo de aplicação é com o *delay* de resposta da FPU, que é controlada pelo *software*. Para que coubesse toda a FPU (adição, multiplicação e divisão) e todo o processador *Softcore Nios* (que até então só funcionava corretamente com todos os periféricos conectados) em uma única FPGA, foi necessário modificar a FPU a fim de se conseguir uma FPU que ocupasse um espaço menor de *hardware* e executasse com uma maior frequência.

Assim, foram retirados alguns controles de exceção da FPU, que para a implementação não se faziam necessários. Foram divididos alguns módulos de processamento, como a multiplicação de inteiro, por exemplo, em mais estágios de processamentos. Alguns *latches* de retenção de sinais foram suprimidos, pois os sinais permaneceriam na entrada até o resultado final ser obtido. Desta forma, conseguiu-se atingir a meta de tamanho e frequência necessários à aplicação.

Todo esse esforço de implementação obteve um resultado excelente como descrito por [Mezencio, 2002] em sua dissertação de mestrado.

6.2 RESULTADOS OBTIDOS

Foram colhidos resultados da aplicação descrita acima na implementação em software sendo executada em uma máquina Pentium III de 866Mhz e em duas versões implementadas em FPGA sendo executada no processador *Softcore Nios*. A primeira implementação no *Nios* utiliza o emulador de FPU para ponto flutuante e a segunda implementação utiliza a FPU desenvolvida neste mestrado para cálculos em ponto flutuante.

Os resultados obtidos são apresentados na Tabela 6.1 [Mezencio, 2002].

Nº	Tipo de Implementação	Passos até o alvo	Tempo para execução Completa	Ciclos de Clock em um Passo
1	PIII 866	38	0.8 Milisegundos	-
2	Nios 33.3	38	940 Milisegundos	29.537
3	Nios 33.3 + Hardware dedicado	41	400 Milisegundos	2.703

Tabela 6.2: Resultados obtidos na implementação do algoritmo de campos potenciais.

O objetivo maior da aplicação implementada como descrito acima foi comprovar a usabilidade da FPU desenvolvida e sua confiabilidade em uma aplicação real. Muito pode ser feito para melhorar o tempo de resposta da FPU desenvolvida especificamente para a aplicação de método de campos potenciais para navegação de robôs móveis implementada por [Mezencio, 2002]. A seguir está reproduzido o comentário dado por Rovilson Mezencio [Mezencio, 2002] em relação à utilização da FPU aqui desenvolvida para realização dos cálculos em ponto flutuante em seu trabalho de mestrado.

“A solução usando FPU, *hardware* dedicado desenvolvido pelo mestrando Murilo Rodrigues que teve como apoio os testes realizados por este pesquisador, mostrou-se um sucesso irrefutável, tanto em tempo de execução como em frequência de trabalho e precisão. Apesar de levar três passos a mais que a média ($41-38=3$) para chegar ao destino, o robô não colidiu com os obstáculos, o que demonstra uma precisão apurada.” [Mezencio, 2002].

7 CONCLUSÃO E TRABALHOS FUTUROS

7.1 CONCLUSÃO

A maior contribuição deste trabalho foi ter aberto a possibilidade de se usar aplicações que necessitam de alta precisão nos cálculos e requerem respostas em tempo real. Estas metas foram atingidas pelo fato da FPMU aqui desenvolvida ter utilizado a melhor tecnologia para projeto de hardware disponível atualmente no mercado. Além disto, a FPMU foi projetada obedecendo rigorosamente o padrão IEEE 754, o que possibilita o seu uso por muitos pesquisadores. Assim, acreditamos que a mesma será aceita na comunidade de *software* livre e disponibilizada futuramente no site opencores.org.

Muito ainda há para ser realizado, devido à complexidade de tratamento dos números em ponto flutuante em aritmética binária. Um caso que merece tratamento especial é o conjunto de funções trigonométricas e exponenciais.

7.2 TRABALHOS FUTUROS

O trabalho desenvolvido nesta dissertação poderá ter continuidade para melhoramentos ou trabalhos futuros, conforme descrito a seguir:

Colocar registradores de trabalho dentro do componente 'FPU_Design', agilizar a comunicação de dados entre a FPU e o aplicativo, principalmente quando o mesmo dado for usado várias vezes em uma sequência de cálculos. O resultado de um cálculo seria inicialmente armazenado em um registrador e por um comando externo de *Load/Store* estes registradores seriam acessados.

Aprimorar a arquitetura da FPU, otimizando o funcionamento do *pipeline* e trabalhando com cada bloco em particular para procurar obter a melhor relação espaço em *hardware* X frequência X latência X *throughput*.

No Capítulo 6 foi apresentado uma maneira de acoplar a FPU no processador *Softcore Nios*, interligando o *Nios* e a FPU através de portas de entrada e saída deixando, assim, o *software* com a responsabilidade de gerenciar a troca de dados. Esta solução pode trazer complexidade extra para o *software*. Uma segunda solução seria adicionar as funcionalidades da FPU diretamente na ALU do *Nios* como na Figura 7-1. A Altera disponibiliza toda a documentação necessária para que isso seja feito, pois ela incentiva que os vários *hardwares* dedicados desenvolvidos com a tecnologia *Quartus II* sejam agregados a seu processador *Nios Softcore*.

Figura 7-1: Adicionando a funcionalidade da FPU na ALU do Nios.

Como pôde ser notado, são vários os caminhos que podem ser seguidos para a melhoria e expansão do trabalho apresentado.

Referência Bibliográficas

[Romero, 2000] Romero, R. A., “*Aprendizado em Robôs Móveis Via Software e Hardware – ARMOSH*” – Projeto de Pesquisa do ICMC-USP, Março de 2000.

[Aragão, Romero, Marques, 2000] Aragão, A.; Romero, R.; Marques, E., “*Computação Reconfigurável Aplicada à Robótica*”, in *Computação Reconfigurável - Experiências e Perspectivas*, pp.184-188, CORE-2000, Marília-SP, Agosto 2000.

[Gonçalves, Wolf, Coelho, Teixeira, Ribeiro, Marques, 2000] Gonçalves, R. A.; Wolf, D. F.; Coelho, F. A. S.; Teixeira, M. A.; Ribeiro, A. A. L.; Marques, E., *ARCHITECT: Um Sistema de Computação Reconfigurável*, in *Computação Reconfigurável - Experiências e Perspectivas*, pp. 42-45, CORE-2000, Marília-SP, Agosto 2000.

[Rodrigues, Gonçalves, Wolf, Osório, Ribeiro, Teixeira, Marques, 2001] Gonçalves, R. A., Wolf, D.F., Rodrigues, M. L., Osório, L. F., Ribeiro, A. A. L., Teixeira M. A., Marques, E., *ARCHITECT-R: A System for Reconfigurable Robots Design - Overview and Initial Results*, IFIP VLSI - SOC 2001 - International Conference on Very Large Scale Integration, Montpellier - França, 2001

[Vincentelli, 1993] Vincentelli, A. S., *Some Considerations on Field-Programmable Gate Arrays and their Impact on System Design*, Lectures Notes in Computer Science, no. 705, p. 26-34, June 1993.

[Chang, 1997] Chang, K.C., *Digital Design and Modeling with VHDL and Synthesis*, IEEE Computer Society, 1997.

[Goldblatt, 1998] Goldblatt, K., *Adapting ASIC Designs for Use with Spartan FPGAs*, Xilinx Application Note, 1998.

[Wolf, 2001] Wolf D. F., *Projeto de Uma Rede Neural Usada no Reconhecimento de Gestos por Robôs Móveis Utilizando-se Computação Reconfigurável*, Dissertação de Mestrado, ICMC USP, São Carlos, Outubro de 2001.

[Stone, 1990] Stone, Harold S., *High-performance computer architecture*, 2ª edição, Addison-Wesley Publishing Company, 1990

- [Leung, Shanblatt, 1997] Leung, Steven S.; Shanblatt, Michael A., *ASIC system design with VHDL* 3ª Edição, Kluwer Academic Publisher, 1997
- [Chapman, 1996] Chapman, K., *Constant Coefficient Multipliers for the XC4000E*, Xilinx Application Note, 1996.
- [Ribeiro, 2002] Ribeiro, A.A.L., *Reconfigurabilidade dinâmica e remota de FPGAs*, Dissertação de Mestrado, ICMC USP, São Carlos, Outubro de 2002.
- [Teixeira, 2002] Teixeira, M.A., *Técnicas de reconfigurabilidade dos FPGAs da família APEX 20K – Altera*, Dissertação de Mestrado, ICMC USP, São Carlos, Outubro de 2002.
- [Luk, Cheung, Shirazi, 2002] Luk, A.G.; Luk W.; Cheung, P.Y.K.; Shirazi, N., “Customising Floating-Point Design”, 2002.
- [Jaenicke, Luk, 2001] Jaenicke, A.; Luk, W., “Parameterised floating-point arithmetic on FPGAs”, *Proc. IEEE Int. Conf. On Acoustics, Speech, and Signal Processing*, IEEE, 2001.
- [Ho, Leong, Leong, Becker, Glesner, 2002] Ho, C.H; Leong, M.P; Leong, P.H.W.; Becker, J.; Glesner, M., "Rapid Prototyping of FPGA based Float PointDSP Systems", Fevereiro de 2002.
- [Koren, 1993] Koren, I., *Computer Arithmetic Algorithms*, Prentice-Hall, 1993.
- [Hwang, 1976] Hwang, K., *Computer arithmetic: principles, architecture and design*, John Wiley & Sons, 1976.
- [Flores, 1963] Flores, *The logic of Computer Engineering*, Prentice-Hall, 1963.
- [Goldberg, 1991] Goldberg, D., *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, ACM Computing Surveys, Vol. 23, No 1, March 1991, pp. 5-48
- [Vandenbut, 1999] VanDenBut, D.E., *The Practical Designer Lab Book*, Prentice-Hall, 1999.

[IEEE, 1985] American National Standard, *ANSI/IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic*, Institute of Electrical and Electronics Engineers, 1985.

[IEEE, 1993] American National Standard, *IEEE Standard VHDL Language Reference Manual (IEEE Std 1076-1993)*, Institute of Electrical and Electronics Engineers, 1993.

[Koopman, 1996] Koopman, Philip, Paper. *Embedded System Design Issues (the Rest of the Story)*, International Conference on Computer Design (ICCD 96), 1996.

[Crowder, 1960] Crowder, Norman A., *The Arithmetic of Computers*, Doubled & Company, 1960.

[Almasi, Gottlieb, 1994] Almasi, G.S.; Gottlieb, A., “*Highly Parallel Computing*” 2^o edição, Benjamin/Cummings Company, Inc., 1994.

[Stallings, 2000] Stallings, W., *Computer Organization and Architecture - fifth edition*, Prentice-Hall, 2000.

[Cavanagh, 1985] Cavanagh, Joseph J. F., *Digital computer arithmetic*, McGraw-Hill, 1985.

[Xilinx, 1997] Xilinx, *The Role of Distributed Arithmetic in FPGA-based Signal Processing*, Xilinx DSP Applications page, <http://www.xilinx.com/products/logicore/dsp>, 1997.

[Altera, 2003] Altera, Corporation, *Nios Development Board Reference Manual, Stratix Edition*, <http://www.altera.com/literature/lit-nio.jsp>, 2003

[Amphion, 2002] Amphion Semiconductor Ltda., <http://www.amphion.com/>, Irlanda do Norte, 2002.

[Xilinx, 2002] Xilinx Corporation, <http://www.xilinx.com/>, 2002.

[Gaisler, 2003] GAISLER RESEARCH, <http://www.gaisler.com/>

[Opencores, 2003] OPENCORES.ORG, <http://www.opencores.org>

[Asics, 2003] ASICS.ws, <http://www.asics.ws>

[CoreDesign, 2003] Digital Core Design, <http://www.digitalcoredesign.com/>.

[Amphion, 2003] Amphion Semiconductor Ltd, <http://www.amphion.com/>.