UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CAMPUS CURITIBA

GERÊNCIA DE PESQUISA E PÓS-GRADUAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA E INFORMÁTICA INDUSTRIAL – CPGEI

EDSON PEDRO FERLIN

ARQUITETURA PARALELA RECONFIGURÁVEL BASEADA EM FLUXO DE DADOS IMPLEMENTADA EM FPGA

TESE DE DOUTORADO

CURITIBA
NOVEMBRO 2008

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial

---

TESE
apresentada ao UTFPR
para obtenção do título de

DOUTOR EM CIÊNCIAS

por

EDSON PEDRO FERLIN

ARQUITETURA PARALELA RECONFIGURÁVEL BASEADA EM FLUXO DE DADOS IMPLEMENTADA EM FPGA

Banca Examinadora:

Presidente e Orientador:

HEITOR SILVÉRIO LOPES (Prof. Dr)	**UTFPR**

Examinadores:

JAIRO PANETTA (Prof. Dr)	**INPE/CPTEC**
JOÃO MAURICIO ROSARIO (Prof. Dr)	**UNICAMP**
MARCOS AUGUSTO SHMEIL (Prof. Dr)	**PUCPR**
CARLOS RAIMUNDO ERIG LIMA (Prof. Dr)	**UTFPR**
JEAN MARCELO SIMÃO (Prof. Dr)	**UTFPR**

Curitiba, novembro de 2008.

EDSON PEDRO FERLIN

ARQUITETURA PARALELA RECONFIGURÁVEL BASEADA EM FLUXO DE DADOS IMPLEMENTADA EM FPGA

Tese apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial da Universidade Tecnológica Federal do Paraná, como requisito parcial para a obtenção do título de "Doutor em Ciências" – Área de Concentração: Informática Industrial.

Orientador: Prof. Dr. Heitor Silvério Lopes
Co-Orientador: Prof. Dr. Carlos Raimundo Erig Lima

Curitiba
2008

Ficha catalográfica elaborada pela Biblioteca da UTFPR – Campus Curitiba

F357a Ferlin, Edson Pedro
Arquitetura paralela reconfigurável baseada em fluxo de dados implementada em FPGA / Edson Pedro Ferlin. Curitiba, UTFPR, 2008
xx, 159 f. : Il. ; 30 cm

Orientador : Prof. Dr. Heitor Silvério Lopes
Co-orientador : Prof. Dr. Carlos Raimundo Erig Lima
Tese (Doutorado) Universidade Tecnológica Federal do Paraná. Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial. Curitiba, 2008
Bibliografia : 149-159

1. Engenharia de computadores. 2. Arquitetura de computadores. 3. Processamento paralelo (Computadores). I. Lopes, Heitor Silvério, orient. II. Lima, Carlos Raimundo Erig, co-orient. III. Universidade Tecnológica Federal do Paraná. Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial. IV. Título

CDD : 621.3

AGRADECIMENTOS

Agradeço primeiramente a Deus, pois somente Ele possibilitou que isto se tornasse realidade, dando vida, força, sabedoria e tranqüilidade.

Agradeço a todas as pessoas que me ajudaram neste período, em especial à minha esposa Sandra, pelo amor, apoio e compreensão; aos meus filhos Pedro Cesar e Gabriel Luis, pela alegria e amor incondicional; à minha mãe Onorina, pelo amor e carinho; ao meu pai Domingos (*in memoriam*); e aos amigos pelo incentivo.

Meus agradecimentos especiais aos professores Heitor Silvério Lopes e Carlos Raimundo Erig Lima que me orientaram e auxiliaram no Doutorado, além da motivação e paciência ao longo deste processo.

Em particular o meu agradecimento aos amigos da Universidade Positivo, em especial ao Prof. Marcos José Tozzi, pela motivação e ajuda, além da liberação de horários e infra-estrutura para que esta tese pudesse ser realizada.

Agradeço, também, a todos os professores e à seção de Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI), por toda a atenção a mim dispensada durante o Doutorado.

Gostaria ainda de agradecer aos membros da banca pela disponibilidade, colaboração e pronto atendimento.

SUMÁRIO

LISTA DE FIGURAS ix
LISTA DE TABELAS xiii
LISTA DE ABREVIATURAS E SIGLAS xv
RESUMO xix
ABSTRACT xx
1 INTRODUÇÃO 1
1.1 MOTIVAÇÕES 1
1.2 OBJETIVOS 4
1.3 CONTRIBUIÇÕES 5
1.4 ESTRUTURA DA TESE 6
2 FUNDAMENTAÇÃO TEÓRICA 9
2.1 COMPUTAÇÃO RECONFIGURÁVEL 9
2.1.1 Arquiteturas Reconfiguráveis 10
2.1.2 Programação e Reconfiguração dos Dispositivos 14
2.1.2.1 Programação 15
2.1.2.2 Compilação 15
2.1.2.3 Modelos de Reconfiguração 16
2.1.2.4 Configuração da Memória 17
2.1.3 Sistemas Computacionais Reconfiguráveis 18
2.1.3.1 Desenvolvimento 18
2.1.3.2 Metodologias e Ferramentas 19
2.2 PROCESSAMENTO PARALELO 21
2.2.1 Arquiteturas Paralelas 22
2.2.1.1 Classificação das Arquiteturas 22
2.2.2 Computadores Paralelos 26
2.2.2.1 Granularidade de Paralelismo 27
2.2.2.2 Escalabilidade e Escalonamento de Processos 28
2.2.2.3 Alocação de Recursos e Balanceamento de Cargas 28
2.2.3 Paralelização de Programas 30
2.2.3.1 Linguagens de Programação 30

2.2.3.2 A Paralelização..... 31
2.2.3.3 Compiladores Paralelizadores..... 34
2.2.4 Computador Fluxo de Dados..... 35
2.3 DESEMPENHO..... 38
2.3.1 Os Programas de Teste..... 38
2.3.2 As Métricas de Desempenho..... 39
2.3.3 Ganho Teórico do Processamento Paralelo..... 43
2.4 TRABALHOS RELACIONADOS..... 45
3 DEFINIÇÃO DA ARQUITETURA..... 49
3.1 CARACTERÍSTICAS..... 51
3.2 VISÃO GERAL DA ARQUITETURA..... 51
3.3 DESCRIÇÃO DA ARQUITETURA..... 54
3.3.1 *Templates*..... 56
3.3.2 Memória de *Templates*..... 57
3.3.3 Memória de Dados..... 60
3.3.4 Organização..... 60
3.3.5 Estrutura Lógica..... 62
3.3.6 Barramentos..... 63
3.3.7 Ciclo de Processamento..... 64
3.3.8 Unidade de Armazenamento..... 65
3.3.9 Unidade de Despacho..... 69
3.3.10 Elementos Processadores..... 73
3.3.11 *Pipeline*..... 76
3.4 DETALHES DA IMPLEMENTAÇÃO..... 77
3.5 CONJUNTO DE INSTRUÇÕES..... 78
3.6 EXEMPLO DE PROGRAMA..... 81
3.7 RECONFIGURAÇÃO DA ARQUITETURA..... 82
3.8 VANTAGENS DA ARQUITETURA..... 84
3.9 LIMITAÇÕES DA ARQUITETURA..... 85
4 VALIDAÇÃO E RESULTADOS..... 87
4.1 TEMPO DE OPERAÇÃO DA ULA..... 87
4.2 TEMPO DE EXECUÇÃO DAS INSTRUÇÕES..... 88
4.3 DIAGRAMA DE TEMPORIZAÇÃO..... 89

4.3.1 Situações de Estudo (Casos)..... 90
4.3.2 Arquitetura sem *Buffer*..... 90
4.3.3 Arquitetura com *Buffer*..... 93
4.3.4 Gasto de Ciclos de *Clock*..... 96
4.3.5 Comparação entre as Arquiteturas com e sem *Buffer*..... 98
4.4 ARQUITETURA - VERSÃO 1.1..... 98
4.5 ARQUITETURA - VERSÃO 1.2..... 101
4.5.1 Teste – Algoritmo Independente (32 Operações)..... 104
4.5.2 Teste – Algoritmo FIR (*Finite Impulse Response*)..... 105
4.6 ARQUITETURA - VERSÃO 1.3..... 109
4.6.1 Teste – Algoritmo Equação Diferencial..... 112
4.6.2 Teste – Algoritmo Independente (32 Operações) e Algoritmo FIR (5-*tap*)..... 114
4.6.3 Teste – Algoritmo de Criptografia IDEA..... 119
4.7 COMPARATIVO COM OUTRAS ARQUITETURAS..... 121
4.8 ANÁLISE DE DESEMPENHO..... 123
5 DISCUSSÃO E CONCLUSÕES..... 127
5.1 ANÁLISE DOS RESULTADOS..... 127
5.2 CONCLUSÕES..... 131
5.3 TRABALHOS FUTUROS..... 134
ANEXOS..... 135
1 DIAGRAMA EM BLOCOS – ARQUITETURA 16EPs (Memória Dividida)..... 135
2 DIAGRAMA EM BLOCOS – ARQUITETURA 4EPs (Memória Única)..... 136
3 DIAGRAMA EM BLOCOS – ELEMENTO PROCESSADOR..... 137
4 DIAGRAMA EM BLOCOS – MEMÓRIA DE *TEMPLATES*..... 138
5 DIAGRAMA EM BLOCOS – MEMÓRIA DE DADOS..... 139
6 LISTAGEM CÓDIGO VHDL – UNIDADE DE DESPACHO..... 140
7 LISTAGEM CÓDIGO VHDL – UNIDADE DE ARMAZENAMENTO..... 142
8 LISTAGEM CÓDIGO VHDL – UNIDADE LÓGICA-ARITMÉTICA..... 144
REFERÊNCIAS BIBLIOGRÁFICAS..... 149

LISTA DE FIGURAS

1 Posicionamento da computação reconfigurável..... 10
2 Níveis de acoplamento em um sistema reconfigurável..... 13
3 Plataformas computacionais..... 15
4 Taxonomia das arquiteturas de computadores..... 23
5 Multiprocessador..... 25
6 Multicomputador..... 26
7 Exemplo de grafo de fluxo de dados..... 36
8 Gráfico do *speedup versus* lei de *Amdahl*..... 42
9 Gráfico do *speedup* ideal *versus* real..... 43
10 Tempo de processamento *versus speedup* ideal em uma execução paralela..... 44
11 Diagrama em blocos da arquitetura paralela reconfigurável..... 52
12 Detalhamento do controle..... 53
12 Visão detalhada da arquitetura paralela reconfigurável..... 55
14 Estrutura básica do *template*..... 56
15 Formato do *template*..... 56
16 Memória de *templates*..... 58
17 Particionamento da memória de *templates*..... 59
18 Memória de dados..... 60
19 Diagrama lógico da arquitetura..... 61
20 Estrutura lógica da arquitetura..... 62
21 Máquina de estados geral da arquitetura..... 65
22 Estágio de armazenamento..... 66
23 Máquina de estados de controle da UA_armazenamento..... 67
24 Máquina de estados de controle da UA_*snoop*..... 69
25 Estágio de despacho..... 70
26 Máquina de estados de controle da UD_despacho..... 71
27 Máquina de estados de controle da UD_consulta..... 73
28 Diagrama em bloco do elemento processador..... 73
29 Estrutura do elemento processador..... 75
30 Máquina de estados de controle dos elementos processadores..... 76
31 *Pipeline* da arquitetura..... 77

32 Fluxograma e representação do comando SE (A<B)..... 80
33 Implementação do comando SE (A<B) em *templates*..... 80
34 Exemplo de programa..... 81
35 *Software* montador de *templates*..... 82
36 Diagrama de temporização (sem *buffer*) – pior caso..... 91
37 Diagrama de temporização (sem *buffer*) – pior intermediário..... 91
38 Diagrama de temporização (sem *buffer*) – melhor caso..... 92
39 Diagrama de temporização (com *buffer*) – pior caso..... 94
40 Diagrama de temporização (com *buffer*) – pior intermediário..... 95
41 Diagrama de temporização (com *buffer*) – melhor caso..... 95
42 Comparativo das curvas de ganho de ciclos de *clock*..... 97
43 Visão geral da arquitetura – versão 1.1..... 99
44 Tempo de execução com 10 *templates* (operações) – versão 1.1..... 100
45 Gráfico do *speedup* - versão 1.1..... 101
46 Visão geral da arquitetura – versão 1.2..... 103
47 Tempo de execução – algoritmo independente – versão 1.2..... 104
48 Gráfico do *speedup* – algoritmo independente – versão 1.2..... 105
49 Grafo de fluxo de dados do filtro FIR 5-tap..... 106
50 Tempo de execução - algoritmo FIR – versão 1.2..... 106
51 Gráfico do *speedup* – algoritmo FIR – versão 1.2..... 107
52 Comparativo de desempenho - algoritmo FIR - versão 1.2..... 108
53 Visão geral da arquitetura – versão 1.3..... 110
54 Detalhamento da arquitetura – versão 1.3..... 111
55 Grafo de fluxo de dados da equação diferencial..... 112
56 Comparativo de desempenho - algoritmo equação diferencial – versão 1.3..... 113
57 Tempo de execução - algoritmo equação diferencial – versão 1.3..... 113
58 Tempo – algoritmos independente e FIR – versão 1.3 (memória única)..... 114
59 Gráfico do *speedup* – independente e FIR – versão 1.3 (memória única)..... 115
60 Simulação da execução 4EPs – algoritmo FIR – versão 1.3 (memória única)..... 116
61 Tempo – algoritmos independente e FIR – versão 1.3 (memória dividida)..... 116
62 Gráfico do *speedup* – independente e FIR – versão 1.3 (memória dividida)..... 117
63 Simulação da execução 4EPs – algoritmo FIR – versão 1.3 (memória dividida).. 118
64 Simulação da execução 16EPs – algoritmo FIR – versão 1.3 (memória dividida). 118

65 Grafo de fluxo de dados do algoritmo IDEA..... 119
66 Ciclos para um bloco de cipher-texto – versão 1.3..... 120
67 Ciclos de *clock* por bloco de cipher-texto – versão 1.3..... 121
68 Curva de tempo normalizado das execuções FIR, independente e ideal..... 124
69 Curva do *speedup* das execuções FIR, independente e ideal..... 125
70 Comparativo do tempo de execução – algoritmo FIR – versões 1.2 e 1.3..... 126

LISTA DE TABELAS

1	Características da arquitetura	51
2	Características da arquitetura paralela genérica	78
3	Conjunto de instruções	79
4	Tempo gasto pela ULA nas operações com 32 bits	88
5	Tempo gasto em cada instrução com ponto fixo	88
6	Resultado das simulações para a arquitetura sem *buffer*	93
7	Resultado das simulações para a arquitetura com *buffer*	96
8	Características da implementação – versão 1.1	100
9	Características da implementação – versão 1.2	102
10	Comparativo de desempenho – algoritmo FIR – versão 1.2	108
11	Características da implementação – versão 1.3	110
12	Comparativo de desempenho – algoritmo equação diferencial – versão 1.3	112
13	Comparativo entre arquiteturas	122
14	Comparativo entre as implementações	129
15	Quadro comparativo entre as arquiteturas	131

LISTA DE ABREVIATURAS E SIGLAS

AHDL - *Altera Hardware Description Language*
ALU - *Aritmectic Logic Unit*
ANSI - *American National Standards Institute*
API - *Application Program Interface*
ASH - *Application – Specific Hardware*
ASIC - *Application – Specific Integrated Circuit*
CAD - *Computer Aided Design*
CFD - *Computational Fluid Dynamics*
CGRA - *Coarse-Grained Reconfigurable Array*
CISC - *Complex Instruction Set Computer*
CMP - *Chip MultiProcessor*
COMA - *Cache Only Memory Access*
COW - *Cluster Of Workstation*
CPI - *Cycles Per Instruction*
CRISC - *Complexed Reduced Instruction Set Computer*
DEFACTO - *Design Environment for Adaptive Computing Technology*
DPU - *DataPath Unit*
DRA - *Dynamically Reconfigurable Architecture*
DSP - *Digital Signal Processor*
EP - Elemento Processador
EPIC - *Explicity Parallel Instruction Computing*
FATOSS - *FAult Tolerant Scheduler System*
FD - Fluxo de Dados
FFT - *Fast Fourier Transform*
FIFO - *First in, First Out*
FIR - *Finite Impulse Response*
FLOPS - *Float Operations per Second*
FPGA - *Field-Programmable Gate Array*
FPU - *Float-Point Unit*
GMD - *German National Research Center*
HPF - *High Performance Fortran*

HPRC - *High-Performance Reconfigurable Computer*

IDEA - *International Data Encryption Algorithm*

IFU - *Interconnected Functional Unit*

IIR - *Infinite Impulse Response*

ILP - *Instruction-Level Parallelism*

IPC - *Instructions Per Cycle*

ISA - *Instruction Set Architecture*

JHDL - *Java Hardware Description Language*

LLP - *Loop_level Parallelism*

MD - Memória de Dados

MIMD - *Multiple Instruction Multiple Data*

MIPS - *Millions of Instructions per Second*

MMX - *MultiMedia eXtension*

MPI - *Message Passing Interface*

MPP - *Massively Parallel Processor*

MT - Memória de *Templates*

NoC - *Network-on-Chip*

NOW - *Network Of Workstation*

NUMA - *Non Uniform Memory Access*

OGMS - *Optimization Generation Memory Structure*

PAE - *Processing Array Elements*

PCA - *Polymorphous Computing Architecture*

PHCA - *Programmable Hardware Cellular Automaton*

PISC - *Packet Instruction Set Computer*

PLP - *Process-Level Parallelism*

PVM - *Parallel Virtual Machine*

RAM - *Random Access Memory*

RISC - *Reduced Instruction Set Computer*

ROCCC - *Reconfigurable Computing Compiler System*

SDF - *Scheduled Dataflow Processor*

SIMD - *Single Instruction Multiple Data*

SISAL - *Streams and Iterations in a Single Assignment Language*

SMP - *Symmetric Multiprocessor*

SoC - *System-on-Chip*

SPEC - *System Performance Evaluation Cooperative*

SRA - *Statically Reconfigurable Architecture*

SSE - *Streaming SIMD Extension*

SUIF - *Stanford University Intermediate Format*

TLP - *Thread-Level Parallelism*

UA - Unidade de Armazenamento

UD - Unidade de Despacho

ULA - Unidade Lógica e Aritmética

UMA - *Uniform Memory Access*

USB - *Universal Serial Bus*

VHDL - *VHSIC Hardware Description Language*

VHSIC - *Very High Speed Integrated Circuits*

VLIW - *Very Long Instruction Word*

VLSI - *Very Large Scale Integration*

XPP - *eXtreme Processing Plataform*

RESUMO

Os problemas de engenharia cada vez mais exigem grandes necessidades computacionais, principalmente em termos de capacidade de processamento, sendo que o tempo de execução é um dos pontos-chave em toda esta discussão. Neste sentido o processamento paralelo surge como um elemento decisivo, pois possibilita uma redução do tempo de processamento em decorrência da execução paralela das operações. Outro fator importante é a questão da computação reconfigurável que possibilita combinar o desempenho do *hardware* com a flexibilidade do *software*, permitindo o desenvolvimento de sistemas extremamente complexos e compactos. Este trabalho tem por objetivo apresentar uma proposta de uma arquitetura paralela reconfigurável baseada em fluxo de dados (*dataflow*), que aproveita a potencialidade tanto do processamento paralelo quanto da computação reconfigurável, e que proporciona uma rápida adequação da máquina paralela ao problema a ser resolvido, garantindo um alto desempenho e uma grande flexibilidade de adaptar o sistema paralelo à aplicação desejada. Esta arquitetura visa explorar o paralelismo existente entre as operações envolvidas nos cálculos numéricos, baseando-se no grafo de fluxo de dados do problema a ser solucionado. A arquitetura é composta por uma unidade de controle, responsável por todo o controle dos Elementos Processadores (EPs) e o fluxo de dados entre eles, e de vários EPs que efetivamente realizam a execução da operação. Ao contrário da computação sequencial, a computação paralela aproveita a disponibilidade dos EPs presentes na arquitetura, garantindo um maior desempenho. Alem disso, a arquitetura pode facilmente ser reorganizada, adaptado-se à aplicação, o que garante uma flexibilidade na classe de problemas computacionais que podem ser executados nesta arquitetura.

ABSTRACT

A Reconfigurable Parallel Architecture based on Dataflow implemented in FPGA

Many real-world engineering problems require high computational power, especially concerning to the processing speed. Modern parallel processing techniques play an important role in reducing the processing time as a consequence of the parallel execution of machine-level operations for a given application software, taking advantage of possible independence between data and operations during processing time. Recently, reconfigurable computation has gained large attention thanks to its ability to combine hardware performance and software flexibility, allowed the development of very complex, compact and powerful systems for custom applications. This work proposes a new architecture for parallel reconfigurable computation that associate the power of parallel processing and the flexibility of reconfigurable devices. This architecture allows quick customization of the system for many problems and, particularly, for numerical computation. For instance, this architecture can exploit the inherent parallelism of the numerical computation of differential equations, where several operations can be executed at the same time using a dataflow graph model of the problem. The proposed architecture is composed by a Control Unit, responsible for the control of all Processing Elements (PEs) and the data flow between them; and many application-customized PEs, responsible for the execution of operations. Differently from sequential computation, the parallel computation takes advantage of the available PEs and their specificity for the application. Therefore, the proposed architecture can offer high performance, scalability and customized solutions for engineering problems.

Keywords: Computer Architecture, Parallel Processing, Reconfigurable Computing.

CAPÍTULO 1

INTRODUÇÃO

1.1 MOTIVAÇÕES

Atualmente a computação está passando por um momento crítico, pois os computadores estão cada vez mais rápidos, por meio da utilização de diversos recursos, mas o *software* não usufrui integralmente deste potencial. Dentre estes recursos está a adoção de arquiteturas paralelas, em que várias unidades de processamento, comumente denominadas de processadores, trabalham cooperativamente para a solução de um problema. Este avanço, em termos de *hardware,* não está sendo acompanhado de forma apropriada pelo *software* que ainda é, em grande parte, desenvolvido no modelo seqüencial, o que inviabiliza o ganho de desempenho em computadores com *hardware* paralelo.

Com o crescente aumento da demanda de poder computacional, em função do grande volume de cálculos que certas aplicações exigem, surgiu necessidade de se investir em computadores com arquitetura com maior capacidade de processamento. Há sistemas computacionais, denominados Petascale, que possuem milhares de processadores e podem realizar aproximadamente um peta (10^{15}) operações por segundo (BELL *et al*, 2006).

É neste enfoque que se apresenta o Processamento Paralelo que objetiva reduzir significativamente o tempo de processamento, pela execução de diversas tarefas, simultaneamente. A esta possibilidade de execução simultânea dá-se o nome de Paralelismo (HENNESSY & PATTERSON, 2006).

Quando se refere ao processamento paralelo, pode-se ter certeza de que a variedade de computadores paralelos é grande. Cada computador paralelo apresenta uma arquitetura e o pouco *software* paralelo explora um tipo específico de paralelismo e ambos dependem da aplicação.

Os computadores paralelos podem ser classificados em duas categorias: com memória compartilhada e com memória distribuída (TANENBAUM, 2007). Os computadores com memória compartilhada se distinguem dos demais por apresentar vários processadores acessando fisicamente uma única memória, que é comum a todos (CASAVANT *et al*, 1996). Por sua vez, os computadores com memória distribuída apresentam uma configuração na qual

cada processador tem acesso à sua própria memória local e as informações compartilhadas devem ser trocadas entre os processadores por meio de rotinas de comunicação (PFISTER, 1998).

Ainda na categoria dos computadores paralelos se enquadram as arquiteturas Fluxo de Dados (*DataFlow*) e surgiram no final da década de 70 para explorar o paralelismo existente entre as instruções de um programa (SILC *et al*, 1999). Estes computadores possuem uma única memória para os dados e para as instruções, não possuem contador de programa, e também não possuem variáveis, pois os valores são representados por pacotes que são transmitidos entre os processadores. Associado a cada processador, existe um *template* que contém informação suficiente para permitir que cada processador funcione. Um programa fluxo de dados é organizado como um grafo. Neste grafo, os nós representam as instruções e os arcos representam o fluxo de dados entre os nós. O paralelismo entre as instruções do programa acontece de forma natural, à medida que a disponibilidade dos dados para um nó esteja satisfeita.

Uma outra abordagem no desenvolvimento de computadores paralelos é a computação reconfigurável. A idéia da utilização de *hardware* reconfigurável para o desenvolvimento de computadores surgiu na década de 60. Contudo, o impulso na utilização só ocorreu na década de 80, com o advento dos dispositivos programáveis, como no caso os FPGAs (*Field-Programmable Gate Array*). Uma arquitetura reconfigurável possibilita uma melhor adequação do *hardware* à aplicação, permitindo explorar estratégias diferentes em função da aplicação a ser executada. Apesar do grande avanço na área de arquiteturas reconfiguráveis, estas são ainda muito pouco exploradas.

A computação reconfigurável proporciona uma maior flexibilidade nas arquiteturas dos computadores, pois até há pouco tempo, apenas o *software* estava direcionado à aplicação, o que limitava o desempenho do sistema computacional. Agora, com a possibilidade do *hardware* também ser adaptável à aplicação, cria-se a possibilidade de se atingir um maior desempenho. Deste modo, um ponto importante é o desenvolvimento de uma arquitetura flexível, na qual o *hardware* seja implementado em lógica programável, utilizando os dispositivos programáveis como os FPGAs.

Nos últimos anos, tem-se observado um crescente aumento no uso da computação, em quase todas as atividades e áreas do conhecimento, principalmente nas relacionadas às ciências e às engenharias, onde se constata a existência de uma grande quantidade de problemas complexos, em que há um grande volume de dados e de operações. As soluções normalmente usadas para resolver estes problemas geram uma grande demanda de recursos

computacionais, necessários para armazenamento, recuperação, transmissão e processamento das informações.

Além disto, em muitos problemas, as soluções precisam ser obtidas em pequenos intervalos de tempo, cada vez menores, ou até mesmo em tempo real. Para uma grande parte destes problemas, as soluções implementadas em *software* seqüencial, executado em *hardware* monoprocessado não atendem às necessidades de tempo de resposta e/ou de desempenho.

Deste modo, novas soluções estão sendo desenvolvidas e utilizadas com o objetivo de melhorar o tempo de resposta, o desempenho e a precisão dos resultados, dentre as quais se destacam as que são baseadas no uso de *software* paralelo e arquiteturas paralelas de propósito geral (PFISTER, 1998); *software* distribuído e arquiteturas de propósito geral (TANENBAUM & VAN STEEN, 2007); *software* seqüencial ou paralelo e arquiteturas dedicadas (SIMA *et al*, 1997); *hardware* dedicado e fixo para aplicações específicas (CASAVANT *et al*, 1996); arquiteturas reconfiguráveis (VILLASENOR & MANGIONE-SMITH, 1997; TURLEY, 1998; SANCHEZ *et al*, 1999; BECKER *et al*, 2000; ORDONEZ & SILVA, 2000; SIPPER & SANCHEZ, 2000; COMPTON & HAUCK, 2002; BOUWENS *et al*, 2007; FERLIN *et al*, 2007; WU *et al*, 2007; SCROFANO *et al*, 2008) e arquiteturas de alto desempenho reconfiguráveis (HPRCs – *High-Performance Reconfigurable Computers*) (BUEL *et al*, 2007; GHAZAWI *et al*, 2008).

Existem algumas arquiteturas reconfiguráveis que exploram o conceito de fluxo de dados, tais como Computador Funcional (QUENOT *et al*, 1993), KressArray (HARTENSTEIN & KRESS, 1995), COLT (BITTNER *et al*, 1996), WASMII e HOSMII (SHIBATA *et al*, 1998), WaveScalar (SWANSON *et al*, 2003), *Asynchronous Dataflow* (TEIFEL & MANOHAR, 2004), Coprocessor (LIU & FURBER, 2005) e XPP (*eXtreme Processing Plataform*) (PACT, 2006) que, no entanto, têm limitações e que estimularam a proposta desta arquitetura.

A motivação para este trabalho é propor alternativas tecnológicas aos pontos fracos que estão presentes em algumas das arquiteturas mencionadas, tais como:

- Ser uma plataforma proprietária, como é o caso da XPP, pois gera uma dependência do fornecedor/fabricante.
- Ter as operações mapeadas diretamente nos elementos processadores como acontece em XPP, WaveScalar, KressArray, WASMII e HOSMII.
- Operar com um tipo único de dado, como KressArray.

- Depender de recursos tecnológicos externos, como no caso do Computador Funcional e Coprocessor.
- O *hardware* ser específico, como é o caso da arquitetura COLT.
- Utilizar blocos construtivos básicos que devem ser conectados como no caso das *Asynchronous Dataflow*.

1.2 OBJETIVOS

O principal objetivo deste trabalho é o estudo e o desenvolvimento de uma arquitetura paralela reconfigurável, baseada em fluxo de dados e implementada em dispositivos programáveis FPGA.

Esta arquitetura utiliza os conceitos de Computação Reconfigurável devido a possibilidade do computador paralelo poder ser reconfigurado, em tempo de compilação, para melhor adequar o *hardware* ao *software*, em função da aplicação a ser realizada. Estas ações são realizadas utilizando-se o modelo de reconfiguração denominado de semi-estático, no qual a reconfiguração ocorre antes do início do novo processamento.

A idéia básica é o desenvolvimento de uma arquitetura paralela na qual há um único elemento de controle (centralizado) e diversos elementos processadores. Neste modelo, a comunicação entre os elementos é por meio de barramentos paralelos usando o envio de requisições. Esta arquitetura recebe o processamento e as informações de configuração de um computador (*host*). O computador (*host*) contém as informações de configuração do computador paralelo, que são enviadas para o FPGA antes do processamento, configurando o dispositivo programável e ajustando os parâmetros da arquitetura à aplicação. Após este processo, o computador paralelo pode receber as informações para o processamento, que são provenientes do *host*, como o programa e os dados de entrada.

Esta arquitetura é baseada no conceito de Fluxo de Dados, no qual as operações podem ser executadas a partir do momento que tiverem todos os dados para serem processados, de modo que podem ser executadas em paralelo tantas operações quantos forem os elementos processadores disponíveis. Com isto, este computador paralelo pode ser utilizado em aplicações com eventos simultâneos, como no caso da computação numérica.

Além do objetivo geral, outras questões são relevantes para auxiliar no desenvolvimento deste trabalho. Estas questões foram formuladas como objetivos específicos:

a) Investigar as abordagens utilizadas nas arquiteturas relacionadas com a proposta analisando seus aspectos fortes e fracos;
b) Identificar e empregar as estratégias de computação reconfigurável e processamento paralelo no projeto de uma arquitetura paralela reconfigurável;
c) Definir os componentes básicos necessários para a arquitetura baseada em fluxo de dados com o foco na implementação em FPGA;
d) Estabelecer uma metodologia de programação para a arquitetura paralela;
e) Demonstrar a viabilidade da arquitetura mediante a execução de algumas aplicações científicas com eventos simultâneos, como filtro digital FIR (*Finite Impulse Response*), equação diferencial e criptografia IDEA (*International Data Encryption Algoritm*).

1.3 CONTRIBUIÇÕES

Há algumas arquiteturas paralelas que já apresentam o conceito de reconfigurabilidade e outras que apresentam o conceito de fluxo de dados. Contudo, poucas incorporam os conceitos mencionados, além de possuírem uma série de limitações, como mencionado anteriormente.

A proposta incorpora o conceito de máquina de fluxo de dados em conjunto com a tecnologia de lógica programável para o projeto de uma arquitetura paralela reconfigurável. A arquitetura proposta é implementada inteiramente em FPGA, aproveitando os recursos e capacidades destes dispositivos, usufruindo dos benefícios da computação reconfigurável para possibilitar que a arquitetura seja adaptada à aplicação.

Esta tese aborda questões importantes para o desenvolvimento de uma arquitetura de computador paralelo em arquiteturas reconfiguráveis. As contribuições desta tese incluem:

- *Definição de uma Arquitetura Paralela Reconfigurável baseada em Fluxo de Dados*

Esta arquitetura explora o conceito de fluxo de dados, que está presente principalmente nas aplicações científicas e de computação numérica. O controle desta arquitetura é centralizado, no qual há um único elemento de controle e diversos elementos processadores. A arquitetura é implementada inteiramente em dispositivos programáveis FPGA. As contribuições específicas nesta área incluem:

- Estudo e desenvolvimento de uma arquitetura paralela que permita a reconfigurabilidade, em tempo de compilação, tanto do *hardware* quanto do *software*.
- Utilização do conceito de fluxo de dados para o escalonamento das operações e para a programação desta arquitetura paralela.

- *Metodologia de Programação da Arquitetura Paralela*

Outra contribuição é o desenvolvimento de uma metodologia de programação paralela, envolvendo a paralelização da aplicação, a configuração da arquitetura e a execução paralela. A contribuição específica nesta área inclui:

- Adoção de uma paralelização que melhor se adapte a esta arquitetura paralela, tomando por base estudos de caso em computação numérica.

- *Aplicação na Arquitetura*

Em termos práticos a arquitetura pode ser utilizada para solucionar uma grande variedade de problemas computacionais de grande complexidade, principalmente os aplicados ao cálculo numérico. A contribuição específica nesta área inclui:

- Aplicação a problemas de cálculo numérico, adequando-os para o processamento na arquitetura proposta.

1.4 ESTRUTURA DA TESE

No Capítulo 2, são apresentados os conceitos e definições envolvendo a computação reconfigurável, como a sua evolução e o estado das arquiteturas reconfiguráveis. São, ainda, abordados tópicos de processamento paralelo, cujo enfoque é diferenciar as diversas arquiteturas paralelas e suas características. Além disto, discute-se algumas métricas úteis para a análise e comparação das arquiteturas para se poder verificar o efetivo ganho de desempenho.

O Capítulo 3 apresenta a definição da arquitetura proposta. O objetivo é descrever as características e parâmetros que foram estabelecidos e levados em consideração na implementação da arquitetura em lógica programável.

No Capítulo 4, relata os experimentos efetuados com a arquitetura paralela reconfigurável proposta e os seus resultados.

No Capítulo 5, faz-se uma discussão dos resultados e apresenta as conclusões obtidas.

CAPÍTULO 2

FUNDAMENTAÇÃO TEÓRICA

Neste capítulo, abordam-se os principais assuntos que balizaram o desenvolvimento da arquitetura paralela reconfigurável. Primeiramente, destaca-se a Computação Reconfigurável que é um dos pontos centrais deste projeto, pois a arquitetura proposta é desenvolvida usando os fundamentos de reconfigurabilidade e a sua implementação é feita em dispositivos programáveis FPGA. O segundo ponto é a questão do Processamento Paralelo que contém os fundamentos da computação paralela visando explorar a execução simultânea das instruções de um programa. Ainda, revisa-se o tópico relativo aos aspectos do desempenho de sistemas computacionais como as métricas mais apropriadas para se avaliar o desempenho das arquiteturas. Por último, apresentam-se as arquiteturas relacionadas, destacando as características que divergem da arquitetura proposta.

2.1 COMPUTAÇÃO RECONFIGURÁVEL

Há dois métodos na computação tradicional para a execução de algoritmos, segundo COMPTON & HAUCK (2002). O primeiro é o que usa um circuito integrado específico para a aplicação, ou ASIC (*Application-Specific Integrated Circuit*), e realiza as operações em *hardware*. Os ASICs são muito rápidos e eficientes quando executam a computação para a qual foram projetados. Contudo, após a fabricação este circuito não pode mais ser modificado. O segundo método utiliza Microprocessadores, que são de longe uma das soluções mais flexíveis em relação aos ASICs. Os microprocessadores executam um conjunto de instruções para realizar a computação. Pela alteração das instruções do *software*, a funcionalidade do sistema é modificada sem alteração do *hardware*. Entretanto, o aspecto negativo desta flexibilidade é que o desempenho acaba sendo afetado e pode ficar muito aquém do que um ASIC. O microprocessador deve ler cada instrução da memória, decodificá-la e, somente então, executá-la. Isto resulta em um alto *overhead* de execução para cada operação individualmente.

A Computação Reconfigurável é um modelo computacional que pretende combinar o desempenho do *hardware* e a flexibilidade do *software* (MARTINS *et al*, 2003), utilizando-se para isto os dispositivos programáveis FPGA (LIMA & GUNTZEL, 2000). SILC *et al* (1999)

definem que a Computação Reconfigurável é o meio para explorar o fato de que muito do tempo de processamento é gasto em uma pequena parte do *software* e a velocidade do *hardware* pode melhorar significativamente o desempenho do sistema computacional. Outra definição é apresentada por COMPTON (2003), na qual a Computação Reconfigurável é a tendência de preencher a lacuna entre o *hardware* e o *software*, potencialmente obtendo mais alto desempenho do que somente com o *software*, enquanto mantém um alto nível de flexibilidade com o *hardware*.

A computação reconfigurável foi inicialmente proposta na década de 60, mas continua sendo um campo de pesquisa novo e com os novos dispositivos programáveis FPGA tornou-se uma solução atrativa e viável economicamente. Na década de 90, a disponibilidade de grande densidade de interconexão e de transistores nos FPGAs propiciou uma nova forma de elementos reconfiguráveis denominados de *Reconfigurable Data Path* (VEMURI & HARR, 2000).

As plataformas reconfiguráveis são pontos-chave para esta tendência atual no projeto de sistemas digitais (HARTENSTEIN, 2001), transpondo a lacuna entre ASICs e microprocessadores, como mostrado na Figura 1.

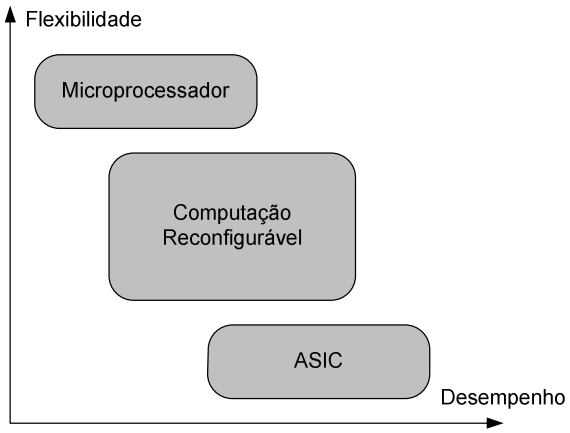


Figura 1: Posicionamento da computação reconfigurável
(Fonte: HARTENSTEIN, 2001)

2.1.1 Arquiteturas Reconfiguráveis

As arquiteturas tradicionais de computadores podem resolver qualquer tipo de computação, desde que lhes sejam submetidos diferentes programas, cada qual com um propósito específico. Para a maioria das computações, esta abordagem centrada na utilização de microprocessadores é mais barata e rápida.

O principal foco do projeto de microprocessadores reside no desempenho e na funcionalidade a ser proporcionada. Contudo, os custos de fabricação e de projeto de arquiteturas baseadas em microprocessador estão aumentando com rapidez (ADÁRIO *et al*, 1999). Tais custos compreendem três aspectos:

- **Custos de *hardware***: Os microprocessadores são maiores e mais complexos do que o necessário para resolver uma tarefa específica.
- **Custos de projeto**: Unidades funcionais, raramente usadas em determinadas aplicações, podem estar presentes em microprocessadores, tendendo a consumir uma parte considerável do esforço de projeto.
- **Custos de energia**: Muita energia é desperdiçada por unidades funcionais ou blocos que não são usados durante grande parte do tempo de processamento.

Considerando-se as aplicações específicas ou requisitos em termos de energia, velocidade e custos, podem ser adotados tipos especiais de microprocessadores, voltados para a aplicação ou otimizados com vistas ao conjunto de requisitos específicos.

Até há pouco tempo, a implementação de microprocessadores específicos para aplicações, por meio de dispositivos programáveis, não era factível, dado o baixo nível de integração evidenciado por estes. Nos dias atuais, entretanto, os dispositivos programáveis FPGA têm atingido densidades superiores a 3 milhões de portas lógicas e, além disso, possuem núcleo RISC (*Reduced Instruction Set Computer*) e memória RAM (*Random Access Memory*) agrupados nos arranjos reconfiguráveis.

Atualmente está ocorrendo um aumento no desenvolvimento de processadores implementados em FPGAs, denominados *Soft Processors* (YIANNACOURAS *et al*, 2007). Um exemplo dessa categoria de processadores é o NIOS II da Altera. Esses processadores são amplamente utilizados em aplicações embarcadas, como a descrita em KATZ & SOME (2003). Ainda nessa categoria tem-se os processadores configuráveis que alcançam uma velocidade em até 6 vezes maior do que os processadores tradicionais nas aplicações específicas (DUTT & CHOI, 2003).

Em assim sendo, acoplando um dispositivo programável FPGA, a um processador, torna-se possível a exploração eficiente do potencial das chamadas **Arquiteturas Reconfiguráveis**.

Arquiteturas reconfiguráveis permitem ao projetista a criação de novas funções e possibilitam a execução de operações, com um número consideravelmente menor de ciclos de *clock* do que o necessário nos microprocessadores. Em uma arquitetura reconfigurável, são desnecessárias muitas das unidades funcionais usualmente encontradas em

microprocessadores e, por conseguinte, há uma substancial economia de energia e tempo de desenvolvimento.

As unidades funcionais podem ser implementadas no dispositivo programável, à medida que a aplicação necessitar, ou ainda, pode-se configurar um subconjunto de unidades funcionais específicas à aplicação, a partir de um conjunto maior, ativando-as durante a execução.

Quando se deseja executar uma computação, tradicionalmente escolhem-se implementações em *hardware* ou *software*. Em alguns sistemas, esta decisão é tomada a partir das tarefas, atribuindo-se algumas ao *hardware* específico e, outras, ao *software* que é executado nos microprocessadores.

As implementações baseadas em *hardware* provêem alto desempenho por dois motivos (DEHON & WAWRZYNEK, 1999):

- **Especificidade**: Implementações em *hardware* são específicas ao problema a ser resolvido, não ocasionando *overhead* adicional para tarefas de interpretação ou circuito extra para a resolução de problemas mais genéricos.
- **Rapidez inerente**: Duas características marcantes nas implementações em *hardware* são a natureza altamente paralela e a execução *espacial*, em que não há uma ordem explícita de execução.

As implementações em *software*, por sua vez, exploram *hardware* de propósito mais geral, o qual interpreta um fluxo de dados como um conjunto de instruções que indicam as operações a serem efetuadas. Conseqüentemente, as implementações baseadas em *software,* segundo DEHON & WAWRZYNEK (1999), são:

- **Flexíveis**: As tarefas podem ser alteradas pela simples substituição das instruções;
- **Relativamente lentas**: Ao contrário de implementações em *hardware*, a execução é eminentemente *temporal* e não *espacial*.
- **Relativamente ineficientes**: Porque os operadores podem ser inadequadamente relacionados à tarefa computacional.

O benefício obtido com o uso de FPGAs — e dispositivos reconfiguráveis em geral — é a atenuação dos dois extremos ilustrados acima: evidencia-se a possibilidade de reconfigurar o *hardware* em período de pós-fabricação, adaptando-o às necessidades da aplicação e, ao mesmo tempo, explorando as vantagens oferecidas pela computação *espacial*, dentre as quais obtém-se a execução de mais de uma operação no mesmo período de tempo.

Há muitas formas diferentes de projetos de arquiteturas para uso na computação reconfigurável. Uma das primeiras variações é o grau de acoplamento (se houver) com o processador principal. A fim de executar mais eficientemente uma aplicação em um sistema de computação reconfigurável, as partes do programa que não forem mapeadas para a lógica reconfigurável são executadas pelo processador principal.

Segundo COMPTON (2003), para os sistemas que utilizam um processador principal em conjunto com a lógica reconfigurável, há diversas maneiras nas quais essas duas estruturas de computação podem ser acopladas, como mostrado na Figura 2.

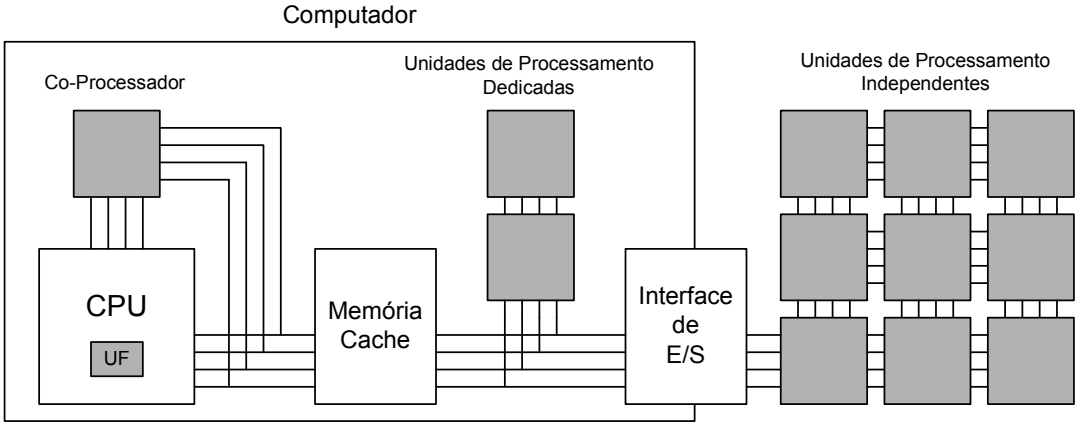


Figura 2: Níveis de acoplamento em um sistema reconfigurável. Obs.: A lógica reconfigurável está sombreada (Fonte: COMPTON & HAUCK, 2002)

Primeiro, o *hardware* reconfigurável pode ser usado somente para prover unidades funcionais reconfiguráveis para o processador principal. Isto possibilita a utilização de um ambiente de programação tradicional com a adição de instruções específicas que podem ser alteradas. Neste caso, as unidades reconfiguráveis executam como unidades funcionais dentro do fluxo de dados do processador principal, com o uso dos registradores para armazenar os dados.

Segundo, uma unidade reconfigurável pode ser usada como um co-processador. Um co-processador é em geral maior que uma unidade funcional e é possível realizar processamento sem a supervisão constante do processador principal. Neste caso, o processador inicializa o *hardware* reconfigurável e também envia os dados necessários para o circuito reconfigurável ou, então, provê informações sobre onde os dados podem ser encontrados na memória.

Terceiro, uma unidade de processamento reconfigurável dedicada comporta-se como se fosse um processador adicional em um sistema multiprocessado. A memória cachê de

dados do processador principal não é visível para esta unidade de processamento reconfigurável dedicada. Há um grande atraso na comunicação entre o processador principal e o *hardware* reconfigurável, como quando comunicam informações de configuração, dados e resultados. Contudo, este tipo de *hardware* reconfigurável permite uma considerável independência na computação, devido à transferência de uma grande parte do processamento para este.

Finalmente, o modelo de acoplamento mais fraco é o de uma unidade de processamento independente externa. Este tipo de *hardware* reconfigurável raramente se comunica com o processador principal (se existir). Este modelo é similar a um *cluster*, no qual o processamento pode ocorrer por períodos de tempo muito longos, sem grande quantidade de comunicações.

Uma compacta integração do *hardware* reconfigurável (BERGER, 2005), que é o mais freqüente, pode ser usada com uma aplicação ou um conjunto de aplicações para um baixo *overhead* de comunicação. Contudo, o *hardware* está habilitado para operar sobre uma parte significativa do tempo sem a intervenção do processador principal e a quantidade de lógica reconfigurável disponível é freqüentemente muito limitada. Os modelos de acoplamento mais fracos permitem um grande paralelismo na execução do programa, mas estão sujeitos a um alto *overhead* de comunicação. Nas aplicações que necessitam de uma grande quantidade de comunicação, isto pode reduzir ou mesmo eliminar algumas vantagens obtidas com esta organização.

2.1.2 Programação e Reconfiguração dos Dispositivos

O desenvolvimento de um *hardware* reconfigurável passa obrigatoriamente pela programação e reconfiguração do dispositivo programável. Segundo MARTINS *et al* (2003), considera-se que:

a) Pode-se programar, por meio de um processo denominado de programação, usando um objeto denominado de programa. No caso de um microprocessador, o programa executável é composto de instruções do conjunto de instruções do nível ISA (*Instruction Set Architecture*).

b) Pode-se configurar, por meio de um processo denominado configuração, usando um objeto denominado padrão de configuração (conjunto de padrões de configuração). No caso de um dispositivo reconfigurável, que pode ser um FPGA, o padrão de configuração é

composto de *bits* de programação de configuração dos blocos lógicos reconfiguráveis e dos elementos de interconexão e/ou roteamento reconfiguráveis.

2.1.2.1 Programação

A programação/configuração das arquiteturas pode ocorrer em três plataformas computacionais (HARTENSTEIN, 2001), como mostrado na Figura 3. A primeira é utilizada para arquiteturas convencionais, e é baseada no modelo de *Von Neumann* para a programação dos microprocessadores. A segunda é para arquiteturas usando lógica programável, em que a programação ocorre utilizando-se duas ferramentas distintas: CAD (*Computer Aided Design*) para a lógica programável e o Compilador para o microprocessador. A terceira é para arquiteturas reconfiguráveis em que utiliza-se um co-compilador, que particiona a aplicação tanto para o microprocessador quanto para a arranjo reconfigurável.

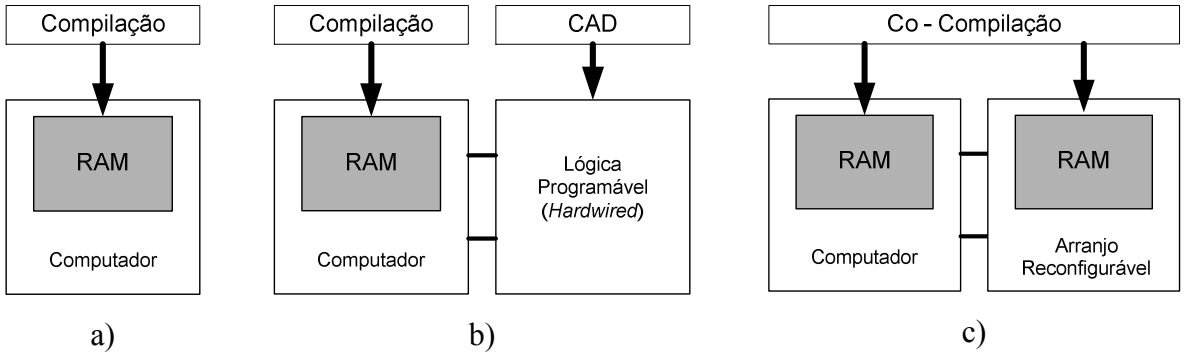


Figura 3: Plataformas computacionais: a) "*Von Neumann*", b) atual, c) reconfigurável (Fonte: HARTENSTEIN, 2001)

2.1.2.2 Compilação

Os compiladores para arquiteturas reconfiguráveis devem considerar a reconfigurabilidade dos componentes em tempo de execução quando geram os diferentes mapeamentos, não apenas observar o aumento da capacidade de multiplexar no tempo as funções nos dispositivos programáveis, mas também agendar reconfigurações para minimizar o *overhead* de configuração (COMPTON & HAUCK, 1999).

Um objetivo importante a ser alcançado pelos modernos compiladores para arquiteturas reconfiguráveis é realizar o particionamento automático da aplicação (BECKER, 1997). Para tal, deve-se ter alguns critérios de controle: i) até que ponto um algoritmo pode

ser paralelizado?, ii) quais as capacidades e as potencialidades disponíveis do dispositivo programável?, e iii) qual é a carga de trabalho suportada pela lógica reconfigurável?.

Um Co-Compilador é um compilador que realiza o particionamento automático da aplicação, escrita em uma linguagem de alto nível, tanto para o microprocessador quanto para o dispositivo programável, como no caso do compilador apresentado em CARDOSO & VÉSTIAS (1999). Em BECKER *et al* (1998) tem-se um trabalho completo a respeito dos métodos de Paralelização e dos modelos de Co-Compilação usados para reconfiguração de arquiteturas reconfiguráveis.

2.1.2.3 Modelos de Reconfiguração

As arquiteturas reconfiguráveis podem seguir modelos distintos de reconfiguração das suas funções. ADARIO *et al* (1999) apresentam, sob a ótica da capacidade de reconfiguração proporcionada pelo projeto da arquitetura reconfigurável, uma generalização dos modelos de execução definidos por PAGE (1996). Esta abordagem divide os modelos de projeto em três classes de reconfiguração, considerando o número de configurações e o instante em que ocorre cada reconfiguração.

- **Estática**

Neste modelo, o circuito possui uma única configuração que nunca é modificada. O dispositivo programável é totalmente programado para executar uma única função que permanece inalterada durante toda vida útil do sistema. Este modelo não explora a flexibilidade provida pela reconfiguração; a única vantagem aproveitada diz respeito às facilidades de projeto e prototipação conferidas pela reconfiguração (ADÁRIO *et al*, 1999). É usada pelos desenvolvedores para avaliar as implementações de protótipos (SILC *et al*, 1999).

- **Semi-estática**

Neste caso, o circuito apresenta várias configurações e as reconfigurações acontecem apenas ao término de cada execução. Dependendo da granulometria das tarefas executadas entre sucessivas reconfigurações, pode-se dizer que este modelo efetua reconfiguração em tempo de execução. Neste modelo, os dispositivos programáveis são usados de forma mais proveitosa. Arquiteturas desta classe são chamadas de SRA (*Statically Reconfigurable Architecture*) (ADÁRIO *et al*, 1999).

- **Dinâmica**

O circuito apresenta várias configurações e as reconfigurações acontecem, de fato, em tempo de execução. Este modelo utiliza eficientemente as arquiteturas reconfiguráveis. As arquiteturas deste tipo são denominadas DRA (*Dynamically Reconfigurable Architecture*) (ADÁRIO *et al*, 1999). A reconfiguração dinâmica dos dispositivos programáveis é a mais poderosa forma de computação reconfigurável (SILC *et al*, 1999), e envolve a reconfiguração do *hardware* ao mesmo tempo em que executa as tarefas.

2.1.2.4 Configuração da Memória

Há uns poucos estilos diferentes de configuração da memória que podem ser usados com sistemas reconfiguráveis: contexto único, contexto múltiplo e parcial. O contexto representa a configuração da memória ou mapeamento dos arranjos lógicos internos do dispositivo programável. Um dispositivo programável de contexto único é programado seqüencialmente e necessita uma reconfiguração total, quando alguma mudança ocorrer na programação. Muitos dispositivos FPGA comerciais são deste tipo. A reconfiguração em tempo de execução, neste tipo de dispositivo de contexto único, deve ser agrupada em contextos que são inteiramente substituídos no *hardware*, quando necessário.

Um dispositivo de contexto múltiplo tem múltiplas camadas de programação, e cada camada pode ser ativada em diferentes tempos. Uma vantagem do dispositivo de contexto múltiplo sobre um de contexto único é a possibilidade de um chaveamento das funções extremamente rápido. O projeto de contexto múltiplo possibilita o carregamento em *background*, permitindo que um contexto seja configurado enquanto outro ainda está em execução.

Os dispositivos, que podem ser programados seletivamente sem uma reconfiguração total, são chamados de parcialmente reconfiguráveis (COMPTON & HAUCK, 1999). O dispositivo parcialmente reconfigurável é também mais apropriado para a reconfiguração em tempo de execução que o de contexto único, pois áreas pequenas do arranjo reconfigurável podem ser modificadas sem necessitar de que o arranjo inteiro seja reprogramado. Isto permite configurações às quais ocupam apenas uma parte da área total para ser configurada no arranjo sem a remoção completa das configurações já presentes.

2.1.3 Sistemas Computacionais Reconfiguráveis

Os sistemas computacionais reconfiguráveis pretendem alcançar ou se aproximar do desempenho das soluções implementadas em *hardware* e da flexibilidade das soluções implementadas em *software* (MARTINS *et al*, 2003).

Segundo COMPTON & HAUCK (2002), quando os sistemas computacionais que contêm tanto *hardware* reconfigurável quanto um microprocessador, o programa deve primeiro ser particionado em seções para serem executadas no *hardware* reconfigurável e seções que devem ser executadas em *software* no microprocessador.

A execução de um sistema reconfigurável ocorre em duas fases distintas (COMPTON & HAUCK, 1999): configuração e execução (propriamente dita). A configuração do *hardware* está sob o controle do microprocessador. As configurações podem ser carregadas exclusivamente na inicialização do programa, ou periodicamente durante o tempo de execução, dependendo do projeto do sistema.

2.1.3.1 Desenvolvimento

Inicialmente, o desenvolvedor de soluções para sistemas computacionais reconfiguráveis fazia o particionamento, quando necessário e, depois, configurava o *hardware* reconfigurável diretamente, considerando os blocos lógicos e de conexão que o compõem. Como essa técnica exige grande conhecimento do *hardware* por parte do desenvolvedor, foram criados modos de desenvolvimento, ambientes, ferramentas e linguagens para facilitar esse trabalho.

Uma ferramenta para o projeto por meio de esquemático e da linguagem de descrição de *hardware* é o QuartusII da Altera (ALTERA, 2008). Essa ferramenta permite a utilização de VHDL (*Very-High-Speed-Integrated-Circuits Hardware Description Language*) (www.vhdl.org), AHDL (*Altera Hardware Description Language)* e módulos de esquemático. Depois da descrição da solução implementada em *hardware*, a ferramenta faz a verificação da sintaxe do projeto e é capaz de fazer simulações funcionais no nível de portas lógicas e comportamentais para linguagens de descrição de *hardware*.

Um sistema que visa não somente o mapeamento na linguagem C para FPGA, mas também a exploração do paralelismo da aplicação é o DEFACTO (*Design Environment for Adaptive Computing Technology*), descrito em detalhe em DINIZ *et al* (2005). Este sistema é

baseado no sistema SUIF (*Stanford University Intermediate Format*) para a conversão para VHDL, como descrito em VERMA & SINGHAL (1999).

Outra ferramenta é a BYU JHDL Open Source CAD Tools, que foi desenvolvida na Brigham Young University (HUTCHINGS *et al*, 1999). Essa ferramenta utiliza uma linguagem chamada JHDL (*Java Hardware Description Language*) (JHDL, 2008). Como a linguagem é de alto nível e orientada a objetos, é extremamente fácil a integração da parte da solução implementada em *hardware* reconfigurável (bibliotecas JHDL expandidas) com a parte implementada em *software*.

A ferramenta da empresa Celoxica chamada DK Design Suíte permite migrar programas escritos na linguagem C (DESIGN, 2008), no caso Handel-C e SystemC (CELOXICA, 2008) e Trident (TRIPP *et al*, 2007), diretamente para os dispositivos programáveis, sem necessidade de utilizar linguagem de descrição de *hardware*. No entanto, é necessário incluir alguns comandos e bibliotecas especiais. É possível, também, gerar o código VHDL correspondente ao programa. Outra possibilidade é utilizar o Matlab para isto por meio da ferramenta AccelChip DSP Synthesis (ACCELCHIP, 2008).

Uma ferramenta desenvolvida para computação reconfigurável é a VFORCE Framework (MOORE *et al*, 2007) baseada em uma extensão da linguagem C++ denominada VSIPL++ (*the Vector/Signal/Image Processing Library*), que encapsula as implementações de *hardware* em uma API (*Application Programming Interface*), possibilitando que a aplicação seja portável.

Existem ferramentas que não geram *a* configuração para os dispositivos programáveis, mas que são capazes de converter linguagens de alto nível para linguagens de descrição de *hardware*. Essa descrição pode então ser usada por uma ferramenta específica que gera a configuração. Exemplos desse tipo de ferramenta são *Atmosphere Simulation Compiler* e *Target Compiler* da empresa Adelante (ADELANTE, 2008). Estas ferramentas são capazes de converter ANSI (*American National Standards Institute*) C para VHDL. Esse tipo de ferramenta é muito útil para os projetistas que desejam desenvolver suas soluções em linguagens de alto nível, de modo que não fiquem presos aos dispositivos suportados pelas ferramentas que transformam a linguagem de alto nível na configuração para os dispositivos.

2.1.3.2. Metodologias e Ferramentas

Em LALL & CIGAN (2005), tem-se uma visão geral sobre as diversas metodologias de desenvolvimento de projetos utilizando-se dispositivos programáveis. ANDREWS *et al*

(2004) comentam que os componentes que combinam um microprocessador e circuitos reconfiguráveis necessitam um modelo de programação que abstraia o *hardware* computacional.

Na solução de diagramas esquemáticos, o particionamento, se houver, é feito manualmente. Uma parte da solução implementada em *hardware* reconfigurável é desenvolvida utilizando-se ferramentas que, a partir da captura do esquemático, geram os *bits* de configuração para um determinado dispositivo programável. O esquemático é uma representação visual de portas e componentes lógicos combinacionais e seqüenciais do *hardware* que faz parte da solução implementada no sistema reconfigurável. Há a necessidade do conhecimento de componentes lógicos e de desenvolvimento de *hardware* por parte do projetista. Geralmente, nesse tipo de ferramenta, é possível usar componentes mais complexos já prontos como somadores, ULAs (Unidades Lógica-Aritmética) e memórias. Essa abordagem geralmente não é aplicada a projetos grandes por causa da dificuldade que existe em se fazer uma representação gráfica de muitos componentes. Além disso, o projetista ainda precisa ter um bom conhecimento de projeto de sistemas digitais.

Outra abordagem é a utilização de linguagens de descrição de *hardware* como VHDL para a parte da solução que é implementada em *hardware*. A VHDL é uma linguagem estruturada que oferece a possibilidade de descrever o *hardware* e este ser simulado antes de sua síntese, facilitando a verificação, tanto em termos de funcionamento quanto em termos de tempos de atraso dos componentes e desempenho, sem a necessidade da prototipação do sistema. Um programa em VHDL pode ser escrito basicamente usando dois tipos (modelos) de descrição: estrutural e comportamental. Na descrição estrutural, a organização física e topológica do sistema é descrita. Isso quer dizer que são especificadas as entradas e saídas, os componentes lógicos, as interligações e os sinais que compõem o sistema. No entanto, na abordagem usando VHDL estrutural, é necessário conhecimento de projeto de *hardware*. Na descrição comportamental, não é necessário descrever a organização física e topológica do sistema, somente o comportamento. Nessa abordagem, são descritas as funções (comportamento) do sistema. Essa abordagem diminui a necessidade de conhecimento de projeto de *hardware*, aumentando a facilidade de desenvolvimento do sistema. No entanto, os sistemas gerados podem não ser tão otimizados em questões de desempenho e área física de dispositivo ocupada quanto os descritos em VHDL estrutural.

A última abordagem é a descrição do sistema por meio de uma linguagem de programação de alto nível como a Linguagem C (SYSTEMC, 2008), Linguagem C++ (OCAPI, 2008), Impulse C (PELLERIN & THIBAULT, 2005), ou mesmo usando o Matlab

(ACCELCHIP, 2008). Nessa abordagem, o projetista não precisa saber como é feito o projeto de *hardware* ou conhecer linguagens específicas de descrição de *hardware*. Além disso, o desenvolvimento do *software* e do *hardware* pode ser feito em conjunto. Apesar do possível desenvolvimento em conjunto, alguns compiladores não conseguem fazer o particionamento automático. Nesses casos, se existir particionamento, o desenvolvedor deve indicar para o compilador quais partes do sistema serão implementadas em *hardware* reconfigurável e quais serão implementadas em *software,* por meio de algumas diretivas de compilação. Em NAJJAR *et al* (2003) é realizado um detalhamento das linguagens de alto nível para a computação reconfigurável.

Quando o compilador é capaz de fazer o particionamento automático, o programa de um sistema computacional tradicional pode ser compilado pela ferramenta, que faz o particionamento da solução, se necessário, utilizado em um sistema reconfigurável. Essa abordagem garante enorme flexibilidade e facilidade de programação do sistema. No entanto, as soluções geradas podem não ter um desempenho tão bom quanto se a arquitetura do sistema fosse descrita manualmente.

A obtenção de alto desempenho com sistemas reconfiguráveis pode ser atingida utilizando um dos métodos para desenvolvimento de aplicações detalhados em HERBORDT *et al* (2007).

2.2 PROCESSAMENTO PARALELO

O Processamento Paralelo é uma realidade devido à difusão dos computadores paralelos, principalmente em decorrência dos baixos custos atuais (STALLINGS, 2006). O carro-chefe desta evolução são os circuitos integrados VLSI (*Very Large Scale Integration*) (TOCCI *et al*, 2006) que, a cada dia, estão mais compactos e potentes com os avanços proporcionados pela microeletrônica, em particular pela utilização de novos materiais semicondutores aliada às novas técnicas de fabricação.

Em teoria, o processamento paralelo pode aumentar ilimitadamente o desempenho dos computadores paralelos. Na prática, o desempenho destes computadores é fortemente limitado, tanto pelo *hardware* quanto pelo *software* (LE & HUU, 1997).

Os computadores paralelos são sistemas computacionais consistindo de múltiplas unidades de processamento conectadas por alguma rede de interconexão (REWINI & BARR,

2005). Além disso, há também o *software* necessário para fazer com que as unidades de processamento trabalhem de forma cooperativa e simultaneamente.

Os computadores paralelos têm sido construídos usando componentes disponíveis no mercado, em especial os processadores. As potencialidades e limitações desses componentes têm forte influência no projeto (TANENBAUM, 2007).

O principal objetivo do projeto de um computador paralelo é fazê-lo processar mais rápido que um computador com um único processador. Contudo, sabe-se que isto não depende somente da arquitetura do computador (*hardware*), mas também de um outro componente, que muitas vezes é negligenciado nesta análise, que é o *software*. Com isto, pouco adianta ter um computador paralelo (com diversos processadores) se o *software* não puder aproveitar este paralelismo do *hardware*.

Em FERLIN (2004) é realizada uma discussão sobre o avanço tecnológico dos processadores e a sua utilização pelo *software*.

2.2.1 Arquiteturas Paralelas

A exploração do paralelismo possibilita o aumento do desempenho dos computadores, mas isto tem como fator limitador a utilização de arquiteturas com apenas um processador. Com isso, adota-se uma configuração que possa suprir este requisito e desta forma utiliza-se o modelo de arquiteturas paralelas, no qual há vários processadores dispostos em uma dada configuração, trabalhando simultaneamente e de forma cooperativa em uma única aplicação.

2.2.1.1 Classificação das Arquiteturas

A taxonomia das arquiteturas de computadores apresentada por *Gordon Bell* (TANENBAUM, 2007), como mostrada na Figura 4 é um aprimoramento da taxonomia de *Michael J. Flynn*, proposta em 1966 (CASAVANT *et al*, 1996) e que classifica as arquiteturas de computadores, da seguinte forma:

- **Fluxo Único de Instruções**
  - Fluxo único de dados
    - CISC (*Complex Instruction Set Computer*)
    - RISC (*Reduced Instruction Set Computer*)
    - Superescalar

        - VLIW (*Very Large Instruction Word*)
        - EPIC (*Explicity Parallel Instruction Computing*)
        - DSP (*Digital Signal Processor*)
        - *Multi-threaded*
    - Fluxos múltiplos de dados
        - Vetoriais
        - SIMD (*Single Instruction Multiple Data*)
- **Fluxo Múltiplo de Instruções**
    - Multiprocessador (Memória Compartilhada)
        - UMA (*Uniform Memory Access*)
        - NUMA (*Non-Uniform Memory Access*)
        - COMA (*Cache Only Memory Access*)
    - Multicomputador (Memória Distribuída)
        - NOW (*Network Of Workstation*)
        - COW (*Cluster Of Workstation*)
        - MPP (*Massively Parallel Processor*)

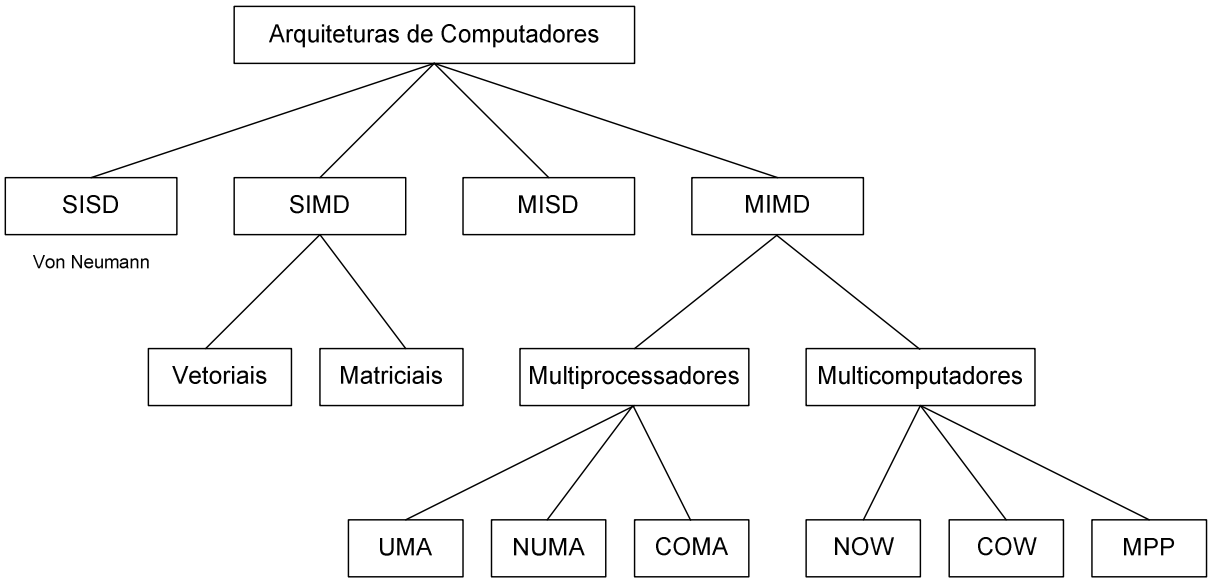


Figura 4: Taxonomia das arquiteturas de computadores

(Fonte: TANENBAUM, 2007)

A seguir destaca-se sucintamente as duas divisões propostas por *Flynn*, e referendadas por *Bell*, mostrando a diferença básica entre fluxo único e múltiplo de instruções.

- **Fluxo Único de Instruções**

Esta categoria é caracterizada pela execução de um único fluxo de instruções em cada processador como os microcomputadores. Isto significa que em um dado instante, a arquitetura executa um único programa e sobre um único processador também podendo trabalhar sobre vários conjuntos de dados.

Nos computadores VLIW utiliza-se um gabarito longo no qual as instruções, que podem ser executadas simultaneamente, são explicitamente especificadas pelo compilador, durante a etapa de compilação do programa. Este gabarito é lido pelo processador durante a execução do programa. Uma vantagem deste modelo é a possibilidade de adaptação automática com o aumento do número de unidades funcionais, pois o gabarito possui informações a respeito do paralelismo das instruções e quais instruções podem ser executadas simultaneamente.

Uma arquitetura que tem diversos *pipelines* paralelos é chamada de arquitetura superescalar (MOTOROLA, 1993), e é o coração da maioria dos processadores atuais, principalmente dos que seguem a filosofia RISC e VLIW, alguns que são CISC ou mesmo híbridos como os CRISC (*Complexed Reduced Instruction Set Computer*). O principal benefício desse recurso é permitir aos processadores executarem mais de uma instrução por ciclo de máquina, por meio dos vários *pipelines* paralelos, que estão atrelados às unidades funcionais.

O modelo SIMD é uma técnica para obtenção do paralelismo em nível de dados sem comprometer o código do programa. Uma implementação SIMD requer relativamente poucas instruções para realizar cálculos complexos com o mínimo de acessos à memória (GOODACRE & SLOSS, 2005). Este modelo está presente nos modernos microprocessadores nas unidades MMX (*MultiMedia eXtension*) e SSE (*Streaming SIMD Extension*).

- **Fluxos Múltiplos de Instruções**

Esta categoria, denominada de MIMD (*Multiple Instruction, Multiple Data*), é caracterizada pela execução de múltiplos fluxos de instruções, sobre fluxos distintos de dados, pelos seus múltiplos processadores. Há duas classes, de acordo com a forma de comunicação entre os processadores: os multiprocessadores e os multicomputadores. Desta maneira, a MIMD é caracterizada por ter vários processadores, cada um executando uma parte do programa, simultaneamente, de forma cooperativa, para a solução do problema.

Multiprocessadores

Nos multiprocessadores todos os processadores compartilham uma mesma memória física, como mostrado na Figura 5. Nesta categoria os computadores paralelos são subdivididos de acordo com o tipo de acesso à memória compartilhada: Acesso à Memória Uniforme (*Uniform Memory Access* - UMA) e Acesso à Memória Não Uniforme (*Non-Uniform Memory Access* - NUMA) (CASAVANT *et al,* 1996). O multiprocessador de acesso à memória uniforme (UMA) tem uma arquitetura na qual o tempo de acesso à memória compartilhada é o mesmo para todos os processadores. Em uma arquitetura NUMA a memória física do sistema é particionada em módulos, cada um dos quais é local e está associado a um processador específico. Como resultado, o tempo de acesso à memória local é menor que o da memória não local.

Os programas desenvolvidos para os computadores UMA podem ser executados sem qualquer modificação em computadores NUMA. Contudo, o desempenho nos computadores NUMA pode ser potencialmente inferior ao obtido nos computadores UMA, considerando a mesma freqüência de *clock* (TANENBAUM, 2007).

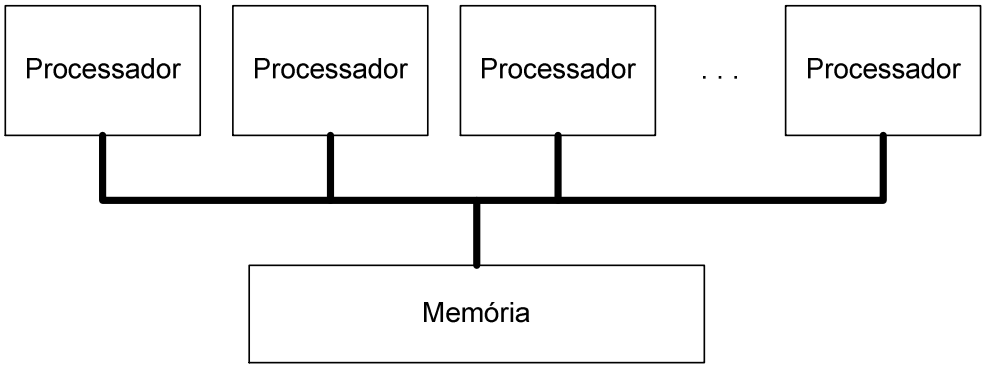


Figura 5: Multiprocessador

Atualmente o enfoque nessa área são as arquitetura com múltiplos núcleos (*multicore*) denominados CMPs (*Chips Multiprocessors*). Os CMPs são formados quando múltiplos processadores são integrados em um mesmo *chip* (CHAPARO *et al*, 2007). Entretanto, há alguns problemas decorrentes do desenvolvimento dos CMPs, como os descritos em AGGARWAL *et al* (2007). Esses CMPs necessitam de novos elementos de conexão como os NoCs (*Network-on-Chip*) (MURALI *et al*, 2007).

Multicomputadores

Um multicomputador ou *cluster* é um sistema único de computadores interconectados que se comunicam via passagem de mensagem (BELL & GRAY, 2002), como mostrado na Figura 6.

Os multicomputadores podem ser divididos em computadores MPP (*Massively Parallel Processor*) e computadores COW (*Cluster of Workstation*) ou NOW (*Network of Workstation*). Um computador MPP é formado por um conjunto de nós mais ou menos padrão, ligados por uma rede de interconexão extremamente rápida. Por exemplo, a família Cray T3E usa os processadores Alpha e o computador Option Red por sua vez utiliza os processadores PentiumPro. Um sistema COW/NOW é composto por diversos microcomputadores ou estações de trabalho ligadas por intermédio de uma rede de comunicação (TANENBAUM, 2007). Um exemplo deste sistema é a arquitetura Beowulf, que é uma abordagem eficiente e de baixo custo (MERKEY, 2008).

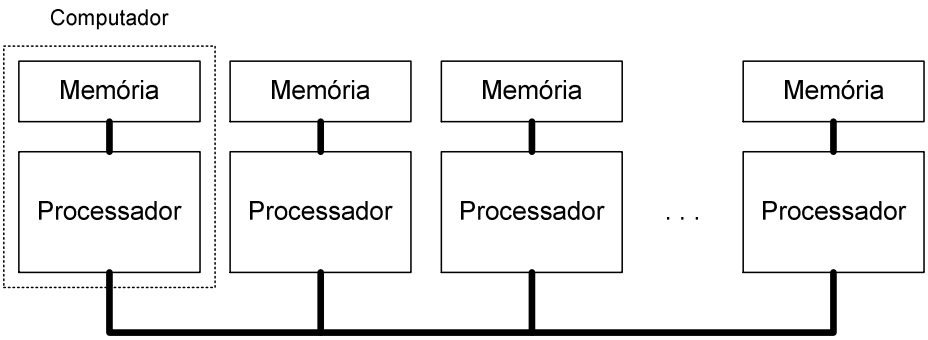


Figura 6: Multicomputador

2.2.2 Computadores Paralelos

Os computadores paralelos se enquadram na categoria denominada MIMD, conforme descrita por Flynn (CASAVANT *et al*, 1996) e podem ser classificados em duas categorias: Memória Compartilhada, como é o caso de Multiprocessadores; e Memória Distribuída, como é o caso dos Multicomputadores ou *Clusters*.

Os computadores paralelos são amplamente utilizados nas aplicações que requerem alto poder de processamento, normalmente envolvendo cálculos complexos, como na área de geoprocessamento, computação gráfica, simulações e outras. O ponto comum nestas aplicações é o grande volume de dados a serem processados, em um curto intervalo de tempo,

e a complexidade da operação a ser realizada. Neste perfil, entram os computadores paralelos, pois podem realizar este processamento em um tempo menor que o tempo esperado na execução pelos computadores com um único processador.

Os computadores com memória compartilhada se distinguem dos demais por apresentarem vários processadores acessando fisicamente uma única memória, que é comum a todos (TANENBAUM, 2007). Por sua vez, os computadores com memória distribuída apresentam uma configuração na qual cada processador tem acesso à sua própria memória local (PFISTER, 1998). Neste caso, as informações compartilhadas devem ser trocadas entre os processadores por meio de rotinas de comunicação, tais como MPI (*Message Passing Interface*) (MPI, 2008) e PVM (*Parallel Virtual Machine*) (PVM, 2008).

O fato dos Multiprocessadores, como descrito em (CARTER, 2003), terem memória compartilhada e seus processadores serem iguais (geralmente) e sincronizados, facilita bastante o trabalho de divisão de tarefas. Pode-se afirmar que nestes computadores, simplesmente distribuindo igualmente as tarefas pelo número de processadores já é um bom começo. A mesma abordagem não funciona de forma eficiente nos Multicomputadores ou *clusters* (PFISTER, 1998). Estes computadores podem ser compostos por processadores distintos, o que pode resultar em desempenhos diferentes. Devido às características dos processadores, é muito provável que o com maior capacidade de processamento (mais rápido, com mais memória) conclui sua(s) tarefa(s) antes do que os mais modestos, ficando ocioso em um determinado período de tempo. É fácil perceber que, nesta situação, pode-se não ter o resultado esperado e pior, pode-se até ter um desempenho inferior ao que se teria caso todas as tarefas fossem alocadas somente ao processador com maior capacidade de processamento.

A interconexão dos diversos componentes de um computador paralelo, tais como processador e memória, pode ser feita de duas maneiras: Estática, na qual os componentes são interligados de forma física durante a construção do computador paralelo ou Dinâmica, na qual há diversos elementos de comutação que fazem o encaminhamento das mensagens entre os componentes, utilizando canais de comunicação ou uma memória compartilhada.

2.2.2.1 Granulometria de Paralelismo

A Granulometria de Paralelismo está intimamente relacionada com a arquitetura ou, mais precisamente, com o *hardware* do computador. Isto decorre do fato de que o tamanho do grão define o tamanho do processamento dos algoritmos e do *software*, e isto tem uma analogia direta com o *hardware*, pois é este *hardware* que efetivamente executa os

programas. A granulometria pode ser expressa em dois modos: **Grossa**, quando se tem porções grandes de *software* em que há pouca ou quase nenhuma comunicação, ou seja, uma alta coesão e baixo acoplamento; e **Fina**, quando ocorre uma grande quantidade de comunicação, entre as porções pequenas de *software*, ou seja, baixa coesão e alto acoplamento, para que o processamento ocorra efetivamente (TANENBAUM, 2001).

As arquiteturas, com um número pequeno de processadores com grande poder de processamento e que possuem conexões de baixa velocidade entre os processadores, são chamadas de fracamente acopladas e, normalmente, trabalharão com uma granulometria de paralelismo grossa. Em um outro extremo, tem-se as arquiteturas compostas por um grande número de processadores pequenos, com pouco poder de processamento, mas interagindo por meio de redes de comunicação de alta velocidade, que constituem arquiteturas ditas fortemente acopladas e que utilizarão uma granulometria de paralelismo fina (TANENBAUM, 2007).

2.2.2.2 Escalabilidade e Escalonamento de Processos

Um sistema ao qual pode-se adicionar mais processadores e aumentar de modo correspondente a sua potência computacional é dito escalável. Em termos ideais, um sistema escalável deve manter a mesma banda passante média por processador e uma latência média constante conforme o número de processadores do sistema aumentar (TANENBAUM, 2007).

STONE (1993) apresenta um modelo estocástico para o escalonamento das tarefas, no qual as tarefas não têm o mesmo tempo de execução e o objetivo é distribuir as tarefas para os processadores de modo que estes sejam utilizados com tempos iguais. Contudo, se isto não for possível, o máximo tempo de execução nos processadores deve ser o menor tempo possível. Uma alternativa para o escalonamento de processos é a utilização do ambiente Mosix (BARAK, 2008) que distribui automaticamente de forma transparente nos nós do *cluster* os processos. Em HONG & PRASANNA (2007) é apresentado um modelo para o escalonamento de processos em um ambiente heterogêneo.

2.2.2.3 Alocação de Recursos e Balanceamento de Carga

A etapa de alocação de recursos é vital no processamento paralelo (SCHWAN & MUKHERJEE, 1996). Se não for feito um escalonamento adequado das tarefas que aproveite

satisfatoriamente os recursos, mais precisamente os processadores, o resultado pode ser desastroso para o desempenho geral do computador paralelo, desprezando todo o trabalho realizado na fase de paralelização do programa.

Entretanto, não basta somente segmentar o programa em vários trechos que possam ser executados em paralelo, utilizando-se um dos métodos de paralelização, pois os multicomputadores podem ser compostos por processadores ou computadores de diferentes desempenhos. Isto leva a um grande problema: como dividir as tarefas da melhor forma possível neste ambiente heterogêneo?

Esta situação traz problemas no balanceamento de cargas e também na própria execução do programa. Isto decorre do fato de ser difícil decidir que trecho do programa é destinado para qual processador. Pode-se estar sobrecarregando o processador, levando a um gasto excessivo de tempo de processamento ou ainda sub-utilizando um processador com tarefas pouco adequadas para o mesmo, ocasionando também baixo desempenho no sistema.

Desta forma, é necessário que haja uma maneira de se avaliar o desempenho de cada um dos processadores do sistema antes das tarefas serem alocadas. Isto possibilita que este processo de alocação, como descrito em SILBERSCHARTZ *et al* (2008) aproveite a potencialidade de cada processador, resultando em um desempenho melhor do sistema.

Pode-se perceber, então, que não basta ter a divisão das tarefas a serem executadas. É necessário também dividir da melhor maneira possível estas tarefas levando em consideração as características do sistema que é utilizado para a sua execução. Para isto, alguns detalhes importantes precisam ser analisados. Primeiro, procurar a melhor maneira de medir o "poder" de processamento dos computadores e o grau de ociosidade. Depois, tentar medir o "peso" (carga) das tarefas a serem distribuídas e, então, estimar o tempo de processamento necessário. Com estas informações, é possível então efetuar uma distribuição mais adequada destas tarefas nos processadores.

Para um escalonamento adequado em multicomputadores, o ponto inicial é conhecer detalhadamente cada um dos processadores. Conhecer as características físicas (como velocidade, memória, tipo) é fácil e não oferece grandes problemas. A parte mais delicada é saber o grau de ociosidade de um processador em um dado momento. Em PERRETTO & FERLIN (2005), descreve-se um sistema escalonador para *cluster* de computadores, denominado de FATOSS (*FAult TOlerant Scheduler System*), que utiliza informações estáticas e dinâmicas para o escalonamento das tarefas.

2.2.3 Paralelização de Programas

A programação dos computadores em alto nível envolve dois itens básicos: a linguagem de programação, na qual o programa é escrito, e o compilador, o qual é um *software* que converte o programa de uma linguagem para o formato executável.

Por meio de uma linguagem de programação e do compilador apropriado é que se obtém o programa a ser executado ou o conjunto de instruções ordenadas logicamente, que fazem com que o computador realize a tarefa desejada.

Isto também se aplica para os computadores paralelos, pois utiliza-se linguagem de programação específica ou convencional com extensão para o desenvolvimento dos programas, e que precisam ser transformados para o formato específico do computador.

2.2.3.1 Linguagens de Programação

Um programa descreve a execução de uma lista de comandos, que são definidos por uma determinada linguagem de programação e que, por sua vez, especifica um conjunto de ações a serem efetuadas pelo *hardware*, de modo que a tarefa seja efetivamente realizada.

A grande maioria dos computadores seguem o modelo de arquitetura *von Neumann*, definida por *John Louis von Neumann*, que segundo TANENBAUM (2007) consiste da associação de um processador e de uma memória, os quais são fisicamente separados e, nesta memória, ficam armazenados tanto o programa, quanto os dados a serem processados.

Com isso, as linguagens de programação sofreram forte influência dos conceitos definidos por *von Neumann*, devido a sua forte orientação para o *hardware* do computador, e estas linguagens diferem substancialmente da forma com que o ser humano pensa e expressa seus conhecimentos. As linguagens de programação que se enquadram nesta perspectiva são denominadas linguagens convencionais (imperativa – procedimentais), como é o caso do C, Fortran e outras. Desta forma, a busca por linguagens de programação mais adequadas ao modo de pensar do ser humano assumiu uma posição de destaque dentro da computação. Assim, as linguagens de programação não convencionais (BROOKSHEAR, 2005), tais como a programação declarativa (funcional e lógica) e a programação imperativa orientada a objetos, tiveram um grande impulso.

- **Linguagens voltadas para paralelismo**

Uma das possibilidades para a programação dos computadores paralelos é a utilização das linguagens ordinárias com uma dada extensão ou mesmo linguagens paralelas. Neste modelo, indica-se a forma ou a maneira pela qual o compilador obtém o paralelismo. O compilador de posse destas informações pode efetuar a paralelização do programa, de acordo com o que foi especificado.

Desta forma, os trechos do programa que podem ser executados em paralelo são informados pelo próprio programador, durante o desenvolvimento do programa. Com isso, fica a cargo do sistema computacional o efetivo particionamento do programa, durante a compilação ou mesmo na execução para os processadores.

A HPF (*High Performance Fortran*) é um exemplo de uma linguagem com extensão (CASAVANT *et al,* 1996), e corresponde a um conjunto de extensões para a Fortran 90, designada para permitir a especificação de algoritmos paralelos. O programador escreve o programa por meio de diretivas de distribuição para especificar a disposição desejável para os dados.

Pode-se também utilizar as linguagens convencionais como C e FORTRAN para escrever os programas paralelos. Para isto basta usar as rotinas específicas para a comunicação entre os processadores por meio da troca de mensagens, utilizando-se as bibliotecas de passagem de mensagens, tais como PVM e MPI.

2.2.3.2 A Paralelização

Com a crescente utilização dos computadores paralelos e o grande parque de *software* atualmente instalado desenvolvido em Fortran, HPF e outras linguagens, tem-se a necessidade do desenvolvimento de compiladores paralelizadores que transformam os programas seqüenciais em paralelos (WOLFE, 1996).

Uma alternativa para esse fato é transformar os programas seqüenciais para o formato paralelo de forma automática ou, pelo menos, semi-automática pelo próprio compilador, possibilitando utilizá-los nos computadores paralelos. Para que isto aconteça, é necessário que se tenha a paralelização dos programas, por meio de um compilador que transforme um programa escrito em uma linguagem convencional em um programa paralelo equivalente. O compilador paralelizador capaz de efetuar esta transformação tem algumas fases adicionais

que um compilador normal não contém, dentre as quais: uma de detecção de paralelismo e outra de alocação de recursos (WOLFE, 1996).

A paralelização dos programas envolve duas etapas distintas, mas interligadas. A primeira é a etapa de detecção dos trechos do programa que podem ser executados simultaneamente pelos processadores. A segunda etapa é de paralelização propriamente dita, na qual o sistema efetivamente realiza a divisão das tarefas que podem ser executadas em paralelo, bem como o seu encaminhamento para os processadores (FERLIN, 1998).

Atualmente, a detecção e a paralelização são efetuadas em tempo de compilação, na qual são extraídas as dependências do programa, analisando-as para se conseguir determinar que partes podem ser executadas em paralelo.

Os métodos de paralelização estudados em FERLIN (1997) e JIMENEZ *et al* (2002) têm uma característica muito peculiar, pois são apropriados para a paralelização de laços em diversos níveis de aninhamento ou profundidade.

As duas abordagens utilizadas para a exploração do paralelismo existente nos programas são: a detecção do paralelismo pelo programador (paralelismo explícito), e a detecção automática do paralelismo (paralelismo implícito).

No primeiro caso tem-se o paralelismo explicito, no qual o programador utiliza as linguagens paralelas (como a HPF), ou mesmo de linguagens seqüenciais estendidas (como a C com rotinas MPI ou PVM) que suportam o paralelismo, para o desenvolvimento dos programas. Desta forma, o programador pode utilizar o seu conhecimento empírico para explorar ao máximo o potencial de paralelização de suas aplicações. No entanto, isto pode conduzir a uma exploração inadequada do paralelismo, devido à falta de experiência por parte do programador.

Por outro lado, no paralelismo implícito, que é o segundo caso, utilizam-se as linguagens, como no caso da linguagem SISAL (*Streams and Iterations in a Single Assignment Language*) (SISAL, 2008), para o desenvolvimento dos programas e a exploração do paralelismo fica sob a responsabilidade do próprio sistema computacional ou, mais precisamente, do compilador paralelizador. A principal vantagem desse método é que o programador não se envolve com a paralelização de suas aplicações, evitando que este interfira e indique uma exploração que seja inadequada ou pouco eficiente.

- **Dependência de dados**

Há dois tipos de dependências que limitam o paralelismo em um programa: as dependências de controle e as de dados. As dependências de controle resultam do fluxo de execução do programa. A dependência de dados por sua vez resulta do fluxo de dados durante a execução do programa e estas dependências podem ser classificadas como: dependência de Fluxo, de Saída e Anti-dependência (LILJA *et al*, 1994).

Existem algoritmos implementados via *software,* e também em *hardware*, que são capazes de eliminar as anti-dependências e as dependências de saída. Estas últimas dependências são consideradas dependências falsas, pois envolvem mais de uma operação de atribuição ao mesmo operando de destino. Tal fato não ocorre com a dependência de fluxo, e por isso é denominada de dependência verdadeira.

Segundo DONGARRA *et al* (2003), a avaliação das dependências é indispensável para a detecção automática de paralelismo em programas seqüenciais. Eliminando-as ao máximo obtém-se um alto grau de paralelismo e as várias operações podem ser executadas simultaneamente, garantindo um melhor desempenho.

Em virtude de nem todas as dependências poderem ser determinadas em tempo de compilação, a paralelização fica comprometida. Isto decorre do fato de haver certas dependências que somente são conhecidas durante a execução do programa como em comandos condicionais, tais como IFs, CASEs.

- **Níveis de Paralelismo**

O paralelismo existe em múltiplos níveis nos computadores paralelos, como descrito por HAMMOND *et al* (1997):

- *Instruction-Level Parallelism* (ILP): paralelismo entre instruções individuais e independentes em uma aplicação.
- *Loop-Level Parallelism* (LLP): quando as instruções de um laço são independentes.
- *Thread-Level Parallelism* (TLP): paralelismo decorrente da execução simultânea de conjuntos de instruções independentes, denominadas de *tasks* ou *threads*.
- *Process-Level Parallelism* (PLP): envolve a execução da aplicação em processos independentes.

O nível mais elevado é o algorítmico (processos e trechos de programas) e o inferior é o de instruções que é implementado em *hardware*. Quanto mais baixo for o nível, mais fina é a granulometria do processamento. O processamento paralelo pode explorar um destes níveis ou até mesmo uma combinação destes, dependendo do computador paralelo e do compilador paralelizador.

Contudo, o grau de paralelismo influencia toda a execução, pois está diretamente relacionado com o tamanho do problema e com a capacidade de processamento do processador. Segundo STONE (1993), a granulometria é apenas um dos fatores que os programadores têm que considerar na paralelização e depende exclusivamente das dependências de dados contidas no programa.

Quanto maior o grau de paralelismo - normalmente processos – menor a quantidade de comunicações entre os componentes do computador paralelo haverá e, conseqüentemente, menos tempo gasto para esta finalidade. Por outro lado, se os grãos forem pequenos, há muitos pedaços - normalmente instruções - que, por sua vez, requererão uma quantidade elevada de comunicações entre os processadores, para a troca de informações e sincronismo, que ocasiona um tempo de comunicação maior, influenciando o tempo total de processamento.

2.2.3.3 Compiladores Paralelizadores

O paralelismo em um programa pode ser explorado implicitamente por meio do uso de compiladores paralelizadores, como o caso do *Parafrase2* (da Universidade de Illinois) (PARAFRASE, 2008), o *SUIF* (*Stanford University Intermediate Format* – da Universidade de Stanford) (STANFORD, 2008), *ParaScope* (Ambiente de Programação Paralela da Universidade de Rice) (HALL *et al*, 1993) e *Adaptor* (da GMD – *German National Research Center*) (ADAPTOR, 2008).

A exploração eficiente do paralelismo existente em um algoritmo depende do seu particionamento em módulos e do escalonamento destes módulos para os processadores do computador paralelo. Com isto, pode-se dividir o problema da paralelização em três aspectos:

- **Identificação do paralelismo do programa**: Identificar o paralelismo presente no programa com o intuito de aproveitar a arquitetura paralela, executando o programa no menor tempo por meio do seu particionamento e do respectivo escalonamento dos módulos.

- **Particionamento do programa**: Particionar o programa em módulos a serem executados em paralelo, visando o menor tempo de execução, e conseqüentemente, determinar o tamanho adequado para cada um destes módulos.
- **Escalonamento dos módulos**: Distribuir os módulos pelos processadores do computador paralelo, aproveitando ao máximo a arquitetura disponível.

A identificação do paralelismo é independente do computador paralelo. Entretanto, o particionamento e o escalonamento são determinados para minimizar a execução do programa em uma arquitetura paralela e normalmente dependem de parâmetros tais como número de processadores, desempenho dos processadores, custo de comunicação, custo de escalonamento e outros.

2.2.4 Computador Fluxo de Dados

As arquiteturas Fluxo de Dados (*dataflow*) surgiram no final da década de 70 para explorar o paralelismo entre as instruções de um programa (SILC *et al*, 1999). Uma arquitetura *dataflow* é uma arquitetura que explora o paralelismo na sua forma natural. Estes computadores possuem uma única memória para os dados e para as instruções e não possuem um contador de instruções como no modelo *von Neumann*.

Os sistemas Fluxo de Dados não possuem variáveis, pois os valores são representados por pacotes que são transmitidos entre os processadores. Cada processador possui a tarefa de executar alguma operação com sua(s) entrada(s) e produzir uma saída contendo o resultado daquela operação. Desta forma, não há variáveis globais ou qualquer outra informação externa. Cada processador pode iniciar a execução assim que seus dados estejam disponíveis. A seqüência das operações é implícita à aplicação e depende exclusivamente do fluxo de dados.

Associado a cada processador existe um *template* que contém informação a respeito da operação a ser realizada, os *buffers* de entrada e uma lista dos destinos de saída. Um ciclo de execução consiste em buscar e despachar todos os *templates* prontos para execução, executá-los, e então armazenar os resultados nos destinos apropriados. Sob estas condições, se o processamento for iniciado com um único processador e forem adicionados mais processadores, o desempenho do computador aumenta até que todo paralelismo implícito tenha sido explorado, aproveitando-se a escalabilidade do sistema.

Um programa fluxo de dados é organizado como um grafo (ARVIND & NIKHIL, 1990). No grafo, os nós representam as instruções e os arcos representam o fluxo de dados entre os nós. No modelo *data driven*, no instante que um nó do grafo detectar que todos os seus arcos de entrada estão habilitados, a instrução é executada, produzindo um resultado na saída. A saída pode habilitar outros nós do grafo de fluxo de dados. Desta forma, o paralelismo entre as instruções acontece de forma natural, na medida em que a disponibilidade dos dados para o nó esteja satisfeita.

Este modelo pode explorar o paralelismo existente nas operações envolvidas como no cálculo numérico de equações diferenciais. Neste tipo de aplicação há várias operações que podem ser executadas simultaneamente no modelo do grafo de fluxo de dados do problema. Por exemplo, a solução numérica da equação diferencial ordinária (Equação 1) sujeita às condições de contorno *y(0)=1* e *y´(0)=(0)* é mostrada na Figura 7.

$$\frac{d^2y}{dx^2} + 2x\frac{dy}{dx} + 2y = 0 \qquad (1)$$

A solução numérica desta equação é apresentada na Figura 7 e possui 16 operações: multiplicação (6), adição (2), subtração (2), duplicação "D" (3), condicional "SE" (1), relacional "<" (1) e parada "S" (1). Inicialmente têm-se cinco nós independentes que podem ser executados simultaneamente, depois outros cinco e assim sucessivamente, seguindo-se o grafo de fluxo de dados.

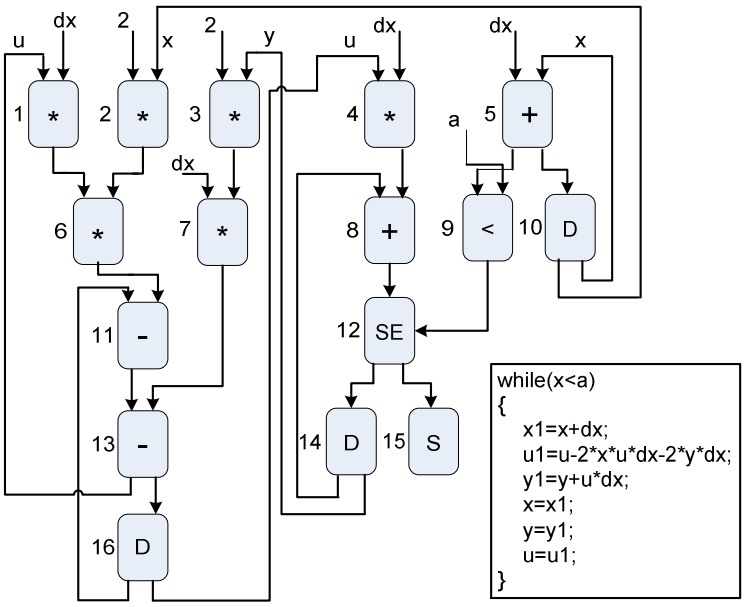


Figura 7: Exemplo de grafo de fluxo de dados

As expressões aritméticas possuem intrinsicamente em suas operações um paralelismo funcional em nível de instrução (ILP), e são propícias para a execução em computadores paralelos. A execução obedece exclusivamente o fluxo natural das operações, apenas com a restrição da disponibilidade dos dados para o processamento.

As arquiteturas fluxo de dados podem explorar tanto o paralelismo ILP quanto o TLP dependendo da complexidade das funções associadas com cada nó no grafo de fluxo de dados. No modelo fluxo de dados o paralelismo ILP aparece quando os nós do grafo representam instruções únicas no caso de ULAs. O paralelismo TLP é explorado quando os nós do grafo são associados com funções complexas no caso de microprocessadores.

O modelo fluxo de dados possui as seguintes vantagens e desvantagens:

- Vantagens:
  - Alto paralelismo – dependendo das operações da aplicação.
  - Alto *throughput* – vários *templates* podem ser processados simultaneamente.
  - Não há restrições – pois não há problemas com a coerência dos dados na memória.
- Desvantagens:
  - Tempo esperando argumentos – os *templates* só são despachados quando todos os dados estiverem disponíveis.
  - Alto *overhead* de controle – uma quantidade significativa dos *bits* do *template* é para informações de controle.

As arquiteturas fluxo de dados foram classificadas por VEEN (1986) como sendo da categoria MIMD. Esta classificação é condizente, pois cada processador trabalha com as suas instruções e com os seus dados. Isto porque cada *template* possui tanto a operação, quanto os dados para o processamento.

Ainda dentro deste aspecto, um computador baseado na arquitetura fluxo de dados pode ser considerado um *Symmetric Multiprocessor* (SMP), pois todos os processadores são idênticos e acessam a mesma memória compartilhada. Em um computador fluxo de dados pode-se realizar tantas operações escalares quanto operações vetoriais. Contudo, uma operação vetorial é desmembrada em várias operações escalares, já que cada processador realiza uma operação a cada momento.

O primeiro computador paralelo baseado em fluxo de dados é o Manchester Dataflow Computer (GURD *et al*, 1985). O protótipo deste computador foi desenvolvido na Universidade de Manchester (UK) em 1981 e tinha 12 unidades funcionais.

2.3 DESEMPENHO

Uma das razões mais importantes para se medir e avaliar o desempenho dos computadores paralelos é discernir se o desempenho atual pode ser melhorado, quando o ambiente paralelo for alterado, como com a inclusão de mais processadores.

As métricas mais usuais para se comparar as arquiteturas paralelas são *Speedup*, Eficiência, FLOPS **(***Floating Point Operations per Second***),** MIPS (*Millions of Instructions per Second*), além de outras. Para avaliar o desempenho das arquiteturas de computadores utilizam-se os programas de teste, denominados de *Benchmarks* (BERGER, 2005), como é o caso dos programas da SPEC (*System Performance Evaluation Cooperative*) (*www.spec.org*), que fornecem informações a respeito de vários elementos do sistema.

Contudo, a utilização de uma única métrica para a avaliação do desempenho dos sistemas está longe de acontecer, pois conforme for o ponto de vista, ter-se-á uma que melhor se adapta, mas não para todos os casos.

2.3.1 Os Programas de Teste

Um *Benchmark* é um programa que é usado para efetuar testes de desempenho do sistema computacional, e tem como objetivo avaliar a influência da arquitetura no desempenho (MURDOCCA & HEURING, 2007).

Os *Benchmarks,* como os SPEC (*www.spec.org*), são compostos por programas que fazem testes envolvendo operações com números inteiros, como o caso do SPECint2000 (onze programas de inteiros) e também para números com ponto flutuante, como o SPECfp2000 (quatorze programas de ponto flutuante), como descritos em HENNING (2000). Outro exemplo é o ZDFPUWinMark (ZDNET, 2008), que consiste de cinco algoritmos para resolver as equações de *Poisson*, *Fast Fourier Transform* (FFT), cálculo de órbitas planetárias, cálculo de áreas de polígonos e uma matriz de eliminação de coeficiente de equações lineares de *Gauss-Jordan*. Estes algoritmos são elaborados para refletir o uso real e produzir um índice único para a comparação dos sistemas. Um outro pacote de *benchmarks* útil para se avaliar computadores paralelos é o Fluent (FLUENTE, 2008a).

A utilização de um *benchmark* implica que os sistemas devem ter a mesma configuração, tanto em termos de *hardware* quanto de *software*. O único componente que altera é o qual se está fazendo a análise do desempenho. Normalmente, o recurso a ser

avaliado é o processador e, neste caso, os demais componentes devem ser os mesmos, para poder comparar o desempenho dos processadores.

2.3.2 As Métricas de Desempenho

A seguir descrevem-se algumas métricas utilizadas para se medir o desempenho dos sistemas computacionais em particular os computadores paralelos.

- **Tempo de Execução**

O tempo é a medida básica de desempenho dos computadores, pois o que realizar a mesma quantidade de trabalho, no menor tempo, é o mais rápido (HENNESSY & PATTERSON, 2006). O Tempo de execução é medido em segundos por programa e é dado pela Equação 2.

$$Tempo\ de\ Execução = \frac{Contagem\ de\ Ciclos\ de\ Clock}{Frequência\ do\ Clock} \quad (2)$$

O tempo é um dos fatores afetados tanto pelo *hardware* quanto pelo *software*. Ele tem ligação direta com os componentes físicos do computador como freqüência de *clock*, velocidade dos componentes e também pela quantidade de instruções necessárias para executar o programa.

- ***Throughput***

O *Throughput* é a soma total dos trabalhos executados em um determinado intervalo de tempo (HENNESSY & PATTERSON, 2006).

- **MIPS**

Uma das maneiras de se evitar a utilização do tempo para medir o desempenho é o uso da métrica MIPS, que é dada pela Equação 3. Esta métrica utiliza o CPI (*Cycles per Instruction*) que é igual à média do número de ciclos de *clock* gastos por instrução executada.

$$MIPS = \frac{Contagem\ de\ Instruções}{Tempo\ de\ Execução \times 10^6} = \frac{Freqüência\ do\ Clock}{CPI \times 10^6} \quad (3)$$

A MIPS é uma taxa de execução de instruções e especifica o desempenho inversamente ao tempo de execução. Desta maneira, os computadores rápidos têm uma alta taxa de MIPS.

Há três problemas decorrentes da utilização da MIPS como uma medida para a comparação de computadores. Primeiro, a MIPS especifica a taxa de execução, mas não leva em conta a capacidade das instruções. Usando a MIPS, não se pode comparar computadores, com diferentes conjuntos de instruções, pois a quantidade de instruções, certamente, é diferente. Segundo, a MIPS varia entre os programas em um mesmo computador; assim o computador pode não ter uma mesma taxa de MIPS igual para todos os programas. Finalmente, e o mais grave, a MIPS pode variar inversamente com o desempenho do computador, de modo que quanto maior a MIPS menor o desempenho (HENNESSY & PATTERSON, 2006).

A métrica MIPS é fortemente afetada pela arquitetura do processador. Sabe-se que um programa para computadores RISC demanda uma maior quantidade de instruções do que para CISC, apesar do CISC demandar um tempo maior para cada instrução.

- **FLOPS**

Uma alternativa é a utilização das operações de ponto flutuante por segundo, abreviadamente FLOPS (PATTERSON & HENNESSY, 2005). A fórmula para FLOPS é dada pela Equação 4.

$$FLOPS = \frac{Contagem\ de\ Operações\ de\ Ponto\ Flutuante}{Tempo\ de\ Execução \times 10^6} \quad (4)$$

Uma operação de ponto flutuante pode ser uma operação de adição, subtração, multiplicação ou divisão, que utiliza operandos de ponto flutuante com precisão simples ou dupla.

- ***Speedup***

O *Speedup* é essencial para expressar quanto mais rápida é a execução de um programa em um computador paralelo do que em uma execução seqüencial (STONE, 1993).

O *Speedup* (***S***) é a razão entre o tempo para executar o algoritmo seqüencialmente e o tempo executando o algoritmo paralelo, usando ***N*** processadores, dado pela Equação 5.

$$S \geq \frac{Tempo\ Seqüencial}{Tempo\ Paralelo} \tag{5}$$

Segundo CASAVANT *et al* (1996), o *Speedup* é obtido pela Equação 6, por meio da aplicação dos limites, em que ***f*** a fração paralelizável do código. Em CENTODUCATTE (2000) tem-se uma explanação mais detalhada.

$$\lim_{N\to\infty} S \leq \lim_{N\to\infty} \frac{1}{(1-f)+\frac{f}{N}} \tag{6}$$

O *Speedup* é um fator que depende de dois itens: o número de processadores da arquitetura e do *software* paralelizado. Esta métrica só tem sentido a partir do momento em que a arquitetura possua uma quantidade de processadores superior ou igual a dois. Caso contrário, estar-se-á apenas fazendo uma simulação e não uma execução em paralelo. Entretanto, de pouca utilidade é esta métrica se o *software* não tiver um código paralelo compatível com o número de processadores. Isto porque alguns processadores podem ficar ociosos durante o processamento e não contribuirão na redução do tempo de processamento.

- ***Lei de Amdahl***

Segundo a *Lei de Amdahl* (CASAVANT *et al*, 1996), proposta por Gene Amdahl, quanto maior a parcela de paralelismo e de partes do programa que podem ser executadas em paralelo, maior é a taxa de *Speedup*. A lei estabelece que "Se uma computação tem uma parte seqüencial *X* e um componente paralelo *Y*, então o máximo *Speedup* é *X*/(*X*+*Y*)" (GRAY *et al*, 2006).

Desta forma, pode-se dizer que o tempo total necessário para a execução de um programa, em uma arquitetura com ***N*** processadores, é dado pela Equação 7.

$$Tempo\ Total = \frac{Tempo\ Paralelo}{N} + Tempo\ Sequencial \tag{7}$$

Na Figura 8, tem-se a visualização das curvas do *Speedup* Ideal (paralelismo completo) e da *Lei de Amdahl* considerando que a parcela seqüencial do código é de 10%. Nesta figura, percebe-se claramente que o *Speedup* máximo, que o sistema pode alcançar, é definido pela curva da *Lei de Amdahl*. Isto porque na maioria dos programas o código não é inteiramente paralelo, devido a uma parcela seqüencial. Quanto maior a parcela paralela do programa, mais o valor do *Speedup* real se aproxima do *Speedup* ideal.

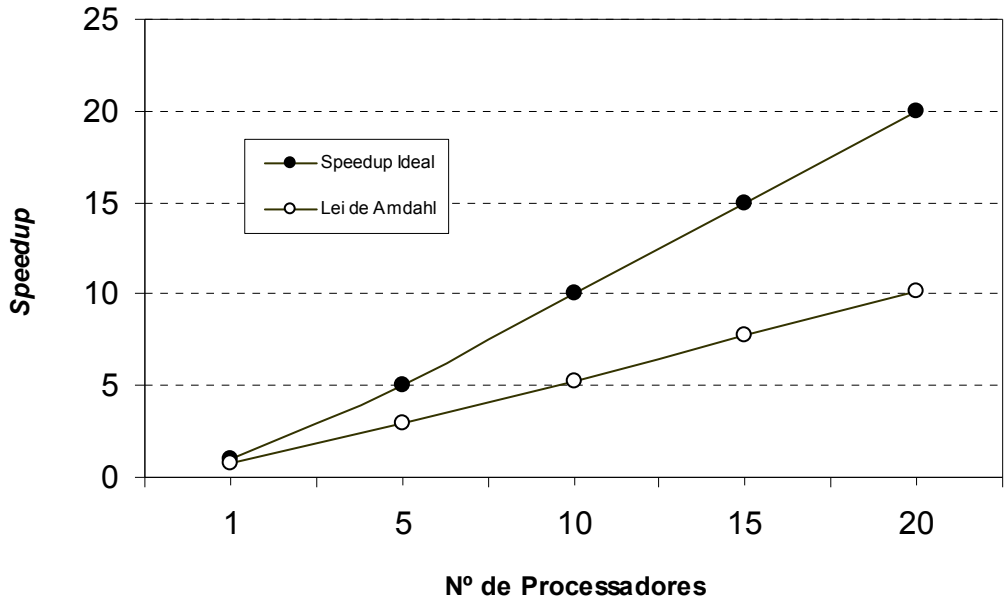


Figura 8: Gráfico do *speedup versus* lei de *Amdahl*

Com base nos resultados obtidos pela execução do *benchmark* Fluent FL5S3 (FLUENT, 2008b), que é um pacote de testes que utiliza o método dos volumes finitos para resolução de problemas CFDs (*Computational Fluid Dynamics*), constata-se que o *Speedup* real é linear ao número de processadores até o total de oito processadores. O valor do *Speedup* real se aproxima do valor ideal atingindo em média 86,91% de eficiência. Contudo, após este valor o *Speedup* real fica muito abaixo do valor esperado. Isto pode ser verificado no gráfico da Figura 9, que foi obtido pelos testes realizados em um *cluster* (com oito processadores) composto por três máquinas: uma HP e uma IBM, ambas com dois processadores Pentium II, e uma Compaq com quatro processadores Pentium Pro, executando o *software* Fluente/uns (FLUENT, 2008c).

A *Lei de Amdahl* é um bom balizador do *Speedup* máximo que pode ser alcançado pelo sistema computacional, pois tem relação direta com a parcela de paralelismo disponível

no *software*. Este valor é afetado pela eficiência do compilador paralelizador e pelas dependências presentes no *software*.

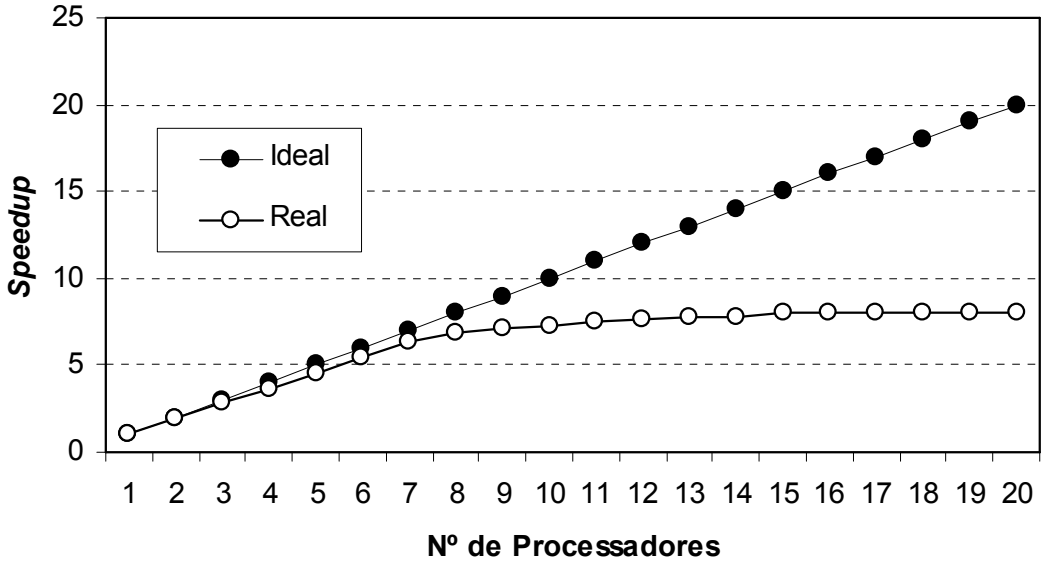


Figura 9: Gráfico do *speedup* ideal *versus* real

- **Eficiência**

A eficiência (***E***) é a razão entre o *Speedup* (***S***) e o número de processadores (***N***). Esta métrica indica quanto do paralelismo foi explorado no algoritmo. Quanto maior for a parcela seqüencial, menor é a eficiência da arquitetura na execução do programa. Com isto, a eficiência é dada pela Equação 8 e é uma medida que indica a proporção do tempo que os processadores estão ocupados (STONE, 1993).

$$E = \frac{S}{N} \tag{8}$$

2.3.3 Ganho Teórico do Processamento Paralelo

O ganho teórico do processamento paralelo nem sempre é alcançado na prática. Este ganho máximo tem como limitantes os aspectos tanto da arquitetura, em particular do *hardware*, quanto do *software*. Os aspectos da arquitetura envolvem o número de EPs (Elementos Processadores) e a organização da arquitetura. Na questão *software* os dois

aspectos fundamentais são o paralelismo da aplicação e as dependências intrínsecas. Nenhum computador paralelo está livre deste peso, mas esta redução de desempenho pode ser minimizada.

No processamento paralelo idealizado, o tempo de execução mínimo é obtido em função da razão entre o tempo seqüencial (com um EP) e o número de EPs. Por exemplo, se uma arquitetura possui quatro EPs, então o tempo de processamento mínimo é ¼ do tempo.

Utilizou-se como tempo normalizado para a execução com um EP o valor 100 (arbitrário) e os tempos de execução para as outras quantidades de EPs e *Speedup* foram com referência a este valor. Os tempos de execução ideal foram obtidos pela razão entre o tempo normalizado e o número de EPs, por exemplo para 2 EPs o tempo é de 50. O *Speedup*, por sua vez, é obtido pela razão entre o tempo normalizado (na execução com um EP) e o tempo de execução para cada quantidade de EPs, como exemplo para 10 EPs o *Speedup* ideal é 10.

Observando-se o gráfico do tempo de processamento e *Speedup* ideais *versus* número de EPs apresentado na Figura 10, constata-se que esta curva do tempo de processamento tem um comportamento exponencial decrescente, enquanto a curva do *Speedup* possui um comportamento linear crescente. Isto implica em dizer que o tempo de processamento ideal decai em uma taxa variável e o *Speedup ideal* aumenta em uma taxa constante em conseqüência do número de EPs.

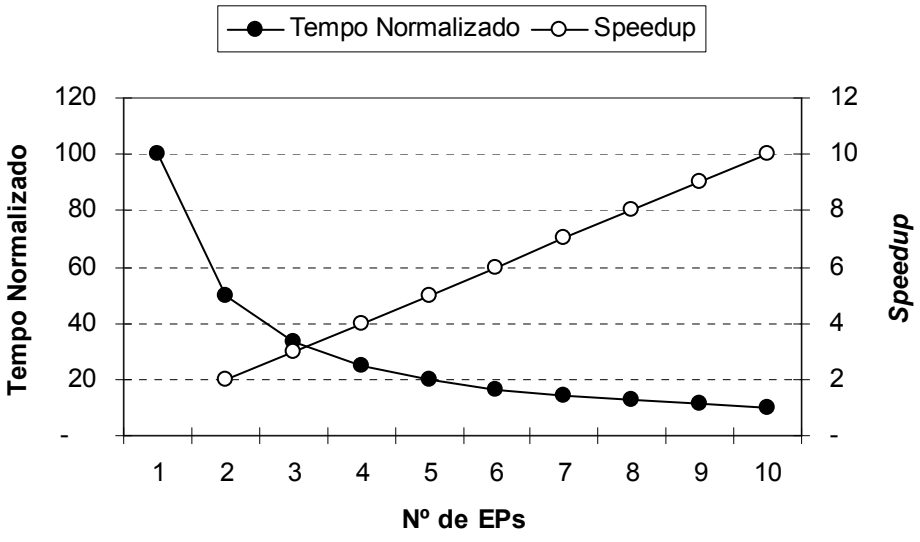


Figura 10: Tempo de processamento *versus speedup* ideal em uma execução paralela

2.4 TRABALHOS RELACIONADOS

Existem algumas arquiteturas reconfiguráveis como as citadas em HARTENSTEIN (2001), NAGELDINGER (2001) e CARDOSO (2004). Dentre estas, algumas arquiteturas merecem destaque por explorarem o conceito de fluxo de dados:

- Computador Funcional (QUENOT *et al*, 1993) possui uma arquitetura mista baseada em *transputer*, que é um microprocessador voltado para processamento paralelo da década de 80, e de EPs denominados de *DataFlow Processor*. Esta arquitetura é direcionada para aplicações de processamento de imagem. Como desvantagens, além do fato de utilizar um componente tecnológico que foi abandonado pelo mercado, ainda apresenta a restrição quanto ao tipo de aplicação.
- KressArray (HARTENSTEIN & KRESS, 1995) também é uma arquitetura que possui uma estrutura matricial, na qual as operações são mapeadas no *cluster* de EPs denominados de DPUs (*DataPath Units*). Entretanto, o controle é centralizado e cada DPU é composto por uma ULA (Unidade Lógica e Aritmética) operando com ponto fixo. Desta forma, as operações podem ser somente com um tipo de dado, limitando as aplicações.
- COLT (BITTNER *et al*, 1996) é uma arquitetura da década de 90 que possui uma matriz de EPs, denominados de IFUs (*Interconnected Functional Units*), que operam como estágios de *pipelines*. Para formar os *pipelines*, de modo a implementar as funções desejadas, tem-se que configurar a matriz de IFUs, utilizando um comutador *crossbar* e isto consome muitos recursos do dispositivo.
- As arquiteturas WASMII e HOSMII, descritas em (SHIBATA *et al*, 1998), são sistemas de *hardware* virtual *data driven* baseado em múltiplos conjuntos de configurações. Os nós e os arcos de um grafo de fluxo de dados são representados diretamente em uma configuração de FPGA, como se fosse um *hardware* específico para aquela aplicação, limitando a flexibilidade e a quantidade de configurações por dispositivo.
- WaveScalar (SWANSON *et al*, 2003) é uma arquitetura matricial de EPs denominados de *WaveCache*, organizados em *clusters*. As operações são mapeadas nos *clusters* e isto dificulta o processo de distribuição das tarefas para os *WaveCache*, pois necessita-se de um estudo prévio sobre a melhor forma de

alocação O controle é descentralizado, ocasionando um consumo maior de recursos do dispositivo para a implementar a lógica de controle.

- *Asynchronous Dataflow* (TEIFEL & MANOHAR, 2004) são dois projetos de arquiteturas baseadas em blocos lógicos, que podem ser acoplados seguindo o fluxo de dados para obter a função desejada. Os blocos lógicos, que são padronizados, operam de forma assíncrona. A comunicação entre os blocos ocorre por meio de canais explícitos de passagem de mensagem. O fator restritivo destas arquiteturas é que os diversos blocos lógicos devem ser configurados e interconectados, usando os canais de comunicação, de modo a implementar o fluxo de dados do processamento. Isto afeta a versatilidade da arquitetura, pois não há uma reutilização do bloco lógico em novas operações, além do que há um expressivo consumo de recursos para a implementação dos canais de comunicação entre os blocos lógicos.
- *Coprocessor* (LIU & FURBER, 2005) é uma arquitetura de um co-processador que opera em conjunto com um processador RISC. Desta forma, a arquitetura apresenta o inconveniente de depender de um processador adicional para o processamento geral, enquanto o co-processador realiza as tarefas no modelo *dataflow*. Outro fator é que as operações são implementadas utilizando-se blocos funcionais padrões, e que são interconectados de modo a produzirem a solução para o problema, limitando a flexibilidade da arquitetura devido ao não reaproveitamento do bloco funcional
- XPP (*eXtreme Processing Plataform*) (PACT, 2006) é uma arquitetura comercial da empresa PACT XPP Technologies e é dependente de uma plataforma proprietária. Esta arquitetura foi introduzida na década de 90 e opera em uma estrutura matricial de EPs denominados de PAE (*Processing Array Elements*). Um agravante é que os PAEs são configurados e explicitamente mapeados para expressarem o processamento desejado. Isto limita a flexibilidade de utilizar a mesma configuração para outras tarefas subseqüentes.

Outras arquiteturas que exploram o modelo *dataflow*, mas que não são implementadas em FPGA ou então são específicas, são:

- SDF (*Scheduled Dataflow Processor*) (KAVI *et al*, 2001) é uma arquitetura multitarefa, que executa tarefas não bloqueantes, composta por três unidades funcionais principais: *execution pipeline* – responsável pela execução da tarefa;

synchronization pipeline – responsável pelo acesso à memória; e *scheduler unit* – responsável pelo escalonamento das tarefas para as demais unidades. É uma arquitetura mista *von Neumann-dataflow* que possui a restrição de ter um contador de programa para determinar a ordem de execução.

- PISC (*Packet Instruction Set Computer*) (CARLSTROM & BODÉN, 2004) é uma arquitetura *dataflow* síncrona para processadores de rede de comunicação composta por um *pipeline* de processadores PISC. Cada processador PISC possui uma estrutura com memória (pacote e de instruções), registradores e quatro unidades funcionais (ULA, *branch*, *copy* e *load*). Esta arquitetura também apresenta a restrição de que a ordem de execução em cada processador deve seguir um contador de programa.
- ASH (*Application – Specific Hardware*) (BUDIU *et al*, 2005) é uma máquina *dataflow* com *hardware* estático, sintetizado como uma coleção de circuitos assíncronos. Estes circuitos operam independente sem utilizar sinais de controle global ou de *clock*. Nesta arquitetura há um mapeamento das operações diretamente nas unidades funcionais, o que limita fortemente o reaproveitamento dessas unidades.
- *Transparent Dataflow Execution* (RUTZIG *et al*, 2007) é uma arquitetura híbrida composta por uma máquina *dataflow*, um processador simples para executar as instruções de controle e, adicionalmente, um mecanismo de *hardware* para detectar e transformar, em tempo de execução, as instruções para serem executadas na máquina *dataflow*. O inconveniente desta arquitetura é que deve haver a transformação dinâmica das instruções convencionais para *dataflow*, e isto acarreta em constantes reconfigurações.
- *Dynamically Reconfigurable Dataflow Architecture* (VOIGT & TEUFEL, 2007) é uma arquitetura para processamento de sinal composta por dois componentes: *Computational Cluster (CC)* – responsável pelas operações e é específico para o problema a ser processado e *distribution unit* – responsável pela conexão entre as CCs. A restrição neste caso está relacionada à especificidade do tipo de aplicação e também à complexidade do mapeamento das operações nos CCs.

Como suporte para as arquiteturas reconfiguráveis *dataflow,* em EKPANYAPONG *et al* (2005) são apresentados três algoritmos de interconexão dos nós e em WU & PENG (2003) tem-se o processo para o desenvolvimento de sistemas SoC (*System-on-Chip*) para

arquiteturas *dataflow*. JIANG & WANG (2007) apresentam um algoritmo para minimizar o custo de comunicação entre tarefas baseado no problema de particionamento temporal de um grafo de fluxo de dados.

Outro tipo de arquitetura reconfigurável são as denominadas de CGRA (*Coarse-Grained Reconfigurable Architecture*). Estas arquiteturas geralmente consistem de uma matriz com um grande número de unidades funcionais interconectadas no modelo de uma malha (GALANIS *et al*, 2007; PARK *et al*, 2008). São arquiteturas em que a reconfiguração ocorre ao nível funcional, garantindo uma grande flexibilidade enquanto reduz o *overhead* de reconfiguração. Alguns exemplos destas arquiteturas são: MorphoSys (SING *et al*, 2000), PipeRench (GOLDESTEIN *et al*, 2000), XiRisc (LODI *et al*, 2003), PHCA (*Programmable Hardware Cellular Automaton*) (CHARBOUILLOT *et al*, 2008) e DART (PILLEMENT *et al*, 2008).

Uma variação das arquiteturas reconfiguráveis CGRA são as PCAs (*Polymorphous Computing Architectures*) voltadas para sistemas embarcados e que podem ser configuradas dinamicamente ou estaticamente para se adaptarem às necessidades computacionais da aplicação (HARDNETT *et al*, 2003; CAVIN *et al*, 2008). Nesta categoria estão as arquiteturas TRIPS (SANKARALINGAM *et al*, 2003) e MONARCH (VAHEY *et al*, 2006).

CAPÍTULO 3

DEFINIÇÃO DA ARQUITETURA

Este trabalho propõe uma arquitetura paralela reconfigurável, baseada em fluxo de dados, implementada em FPGA (*Field-Programmable Gate Array*). Esta arquitetura utiliza os conceitos da computação reconfigurável, na qual o computador paralelo pode ser reconfigurado para melhor adequar o *hardware* e o *software* em função da aplicação.

Esta arquitetura apresenta quatro pontos básicos: (1) permite explorar o paralelismo nas aplicações; (2) possibilita que vários elementos processadores (EPs) operem em paralelo com controle centralizado; (3) o controle é baseado no fluxo de dados da aplicação; e, (4) readaptação para outras aplicações.

Ao longo dos anos, tem-se elaborado várias propostas de Arquiteturas de Computadores, como as citadas em HARTENSTEIN (2001), CARDOSO (2004) e BUEL *et al* (2007). Umas paralelas, outras reconfiguráveis, outras ainda utilizando fluxo de dados, ou mesmo combinações destas. É justamente a conjunção destas características que motivou o desenvolvimento desta arquitetura.

Uma arquitetura com estas características permite atender melhor às necessidades computacionais que algumas aplicações exigem, garantindo uma maior flexibilidade em termos de processamento. Um procedimento comum nos sistemas microprocessados é a reprogramação do *software*, adaptando-os a uma determinada arquitetura. Com a possibilidade da reconfiguração, o *hardware* também passou a ser reprogramado. Isto tira proveito de ambos os lados de um sistema computacional: físico (*hardware*) e lógico (*software*).

A maioria dos computadores paralelos têm sido construídos usando componentes disponíveis no mercado, em especial os EPs, como os microprocessadores. As potencialidades e limitações destes componentes têm forte influência no projeto dos computadores (TANENBAUM, 2007). O desenvolvimento por meio da lógica programável minimiza tais limitações, possibilitando alcançar um potencial de processamento elevado e um aproveitamento melhor dos recursos disponíveis. Isto tudo tem o objetivo de aumentar o desempenho de um sistema computacional, rompendo com o paradigma de *hardware* tradicional *versus software* tradicional.

Esta arquitetura explora o paralelismo *espacial*, no qual a simultaneidade ocorre em função da independência das operações. Pode-se associar o paralelismo espacial com o ILP (*Instruction-Level Parallelism*) no qual o paralelismo ocorre em função das instruções. Contudo, a obtenção do paralelismo é mais complexa, devendo-se utilizar um método de paralelização como os descritos por FERLIN (1997).

Com esta arquitetura, pode-se explorar os conceitos de *macro* e *microtasking* (SATO *et al*, 1996). A *macrotasking* explora o paralelismo TLP (*Thread-Level Parallelism*) e baseia-se na divisão do trabalho em seqüências de execução e as operações são executadas apenas com base no fluxo de dados da seqüência. O *microtasking,* por outro lado, explora o paralelismo ILP, no qual as diversas operações podem ser executadas simultaneamente pelos EPs.

Esta arquitetura aproveita o conceito de Fluxo de Dados, na qual as operações - aqui chamadas de *templates* - podem ser executadas pelos EPs, a partir do momento em que tiverem todos os dados para serem processados. Deste modo, esta arquitetura não utiliza o contador de programa, como no caso das arquiteturas *von Neumann*, pois a ordem de execução é determinada pela disponibilidade dos dados nos *templates*. Não há um controle sobre a seqüência de execução, que não seja aquela definida pela própria execução dos *templates*, ou seja, os *templates* estruturam a seqüência das operações. Isto é importante, pois cada *template* pode ser processado independentemente dos demais, na medida em que houver EPs livres para processamento.

A arquitetura paralela é desenvolvida em lógica programável, utilizando dispositivos programáveis FPGA. Isto possibilita a utilização do conceito de reconfigurabilidade, pois os blocos lógicos dos FPGAs são reconfigurados para implementar as funções lógicas especificadas, inclusive na reconfiguração da interconexão entre estes blocos lógicos. Esta reconfiguração da arquitetura ocorre imediatamente antes da execução da aplicação, de forma semi-estática, especificamente para o problema a ser executado.

Esta arquitetura pode ser utilizada em aplicações com eventos simultâneos, principalmente em cálculos, que podem ser executados paralelamente, como na resolução de equações diferenciais. Os problemas desta natureza são freqüentemente encontrados, tanto nas áreas de Engenharia quanto nas Ciências Exatas e Tecnológicas. Entretanto, com uma correta reconfiguração, pode-se expandir a sua utilização para outras áreas.

3.1 CARACTERÍSTICAS

No desenvolvimento desta arquitetura utilizaram-se as características apresentadas na Tabela 1. Contudo, a arquitetura proposta tem dois pontos principais: o modelo computacional fluxo de dados e a reconfigurabilidade.

Tabela 1: Características da arquitetura

Característica	Descrição
Complexidade do EP	ULA (Inteiro ou Ponto Flutuante)
Quantidade de EPs	Variável (módulo com 4 EPs) máx 16
Modelo Computacional	Fluxo de Dados
Paralelismo	Nível de Rotina (TLP) e de Instruções (ILP)
Precisão dos Dados	Palavra de 8, 16 ou 32 *bits*
Programabilidade	Completa
Reconfigurabilidade	Semi-estática

O modelo de programação desta arquitetura possui os seguintes pontos básicos:

- Não há variáveis.
- Baseada em grafo de fluxo de dados.
- O fluxo de execução é determinado pelos dados (*data-driven*).
- Os nós representam as computações.
- Vários EPs processam em paralelo.
- Um controle centralizado dos EPs.

3.2 VISÃO GERAL DA ARQUITETURA

A idéia básica deste projeto é o desenvolvimento de uma arquitetura paralela, composta por um único Controle e vários EPs, implementada em lógica programável (FPGA), como apresentado na Figura 11.

O Controle faz o gerenciamento de toda a arquitetura e a comunicação com o exterior, recebendo os dados e operações e enviando o(s) resultado(s). Esta unidade também gerencia o envio dos *templates* (operações) para os EPs e o armazenamento do processamento.

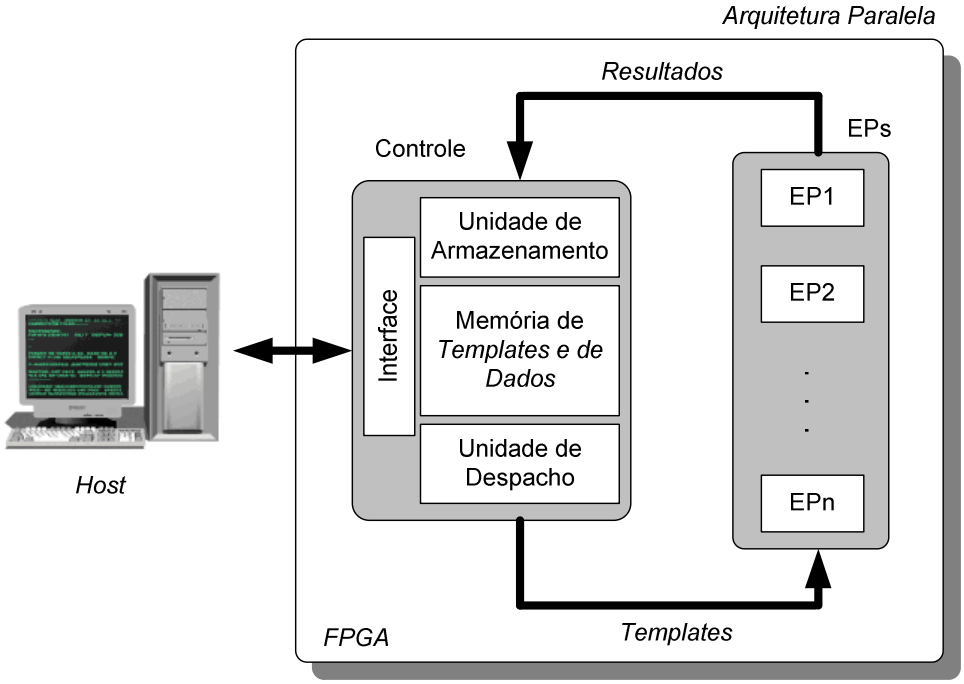


Figura 11: Diagrama em blocos da arquitetura paralela reconfigurável

O Controle realiza o ciclo de execução seguindo o grafo de fluxo de dados da aplicação. O Controle é composto por cinco componentes básicos, mostrados na Figura 12:

- **Interface** que faz a comunicação com o *host.*
- **Memória de *Templates* (MT)** na qual ficam armazenados os *templates* para serem processados.
- **Memória de Dados (MD)** na qual ficam armazenados os dados durante o processamento.
- **Unidade de Despacho (UD)** que verifica se há *templates* para serem processados e os envia para os EPs disponíveis.
- **Unidade de Armazenamento (UA)** que recebe os resultados dos EPs e os atualiza nos *templates* apropriados ou na memória de dados.

Os EPs, por sua vez, são os responsáveis pelo processamento dos *templates*. É importante destacar que os EPs trabalham de forma independente. A comunicação entre o Controle e os EPs ocorre por meio de barramentos paralelos. O *template* é equivalente a uma instrução em um microprocessador e possui todas as informações necessárias para o processamento da operação: código da operação, operandos e destinos. Neste modelo não há leituras subseqüentes da memória para os operandos, como acontece no modelo de *von Neumann*.

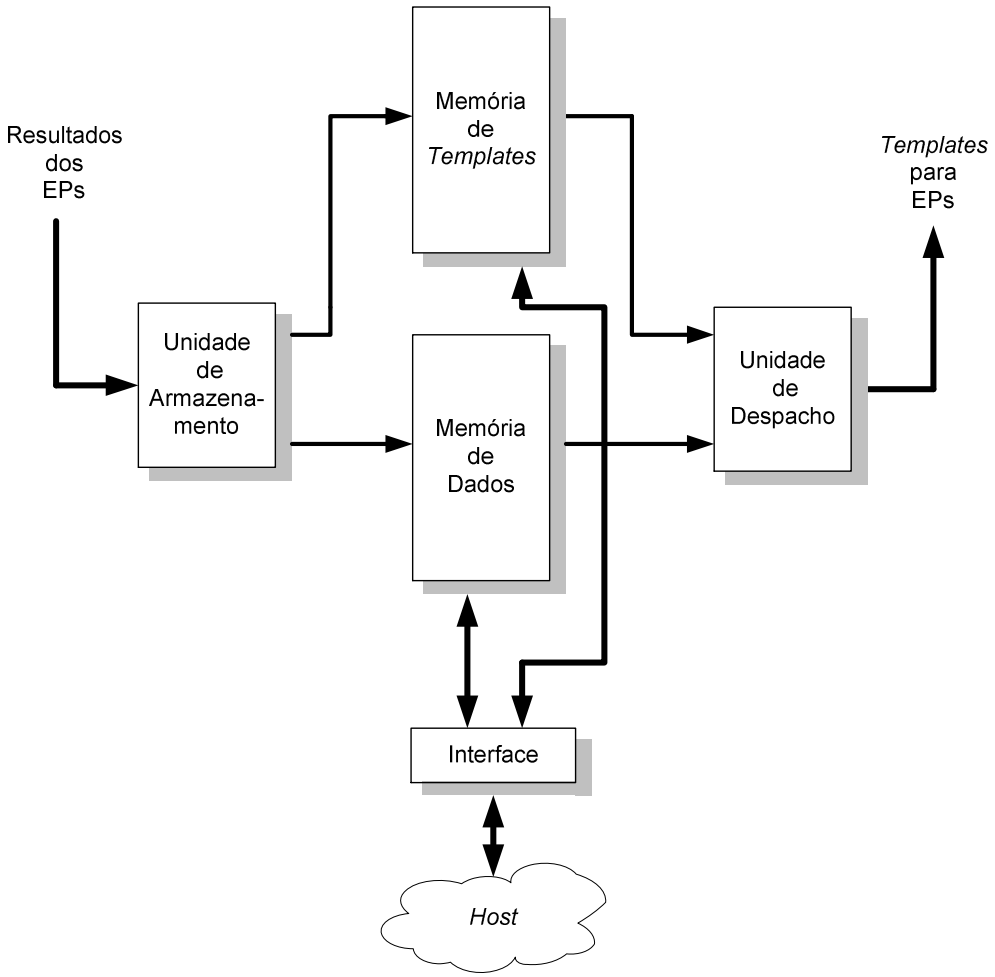


Figura 12: Detalhamento do controle

Esta arquitetura é acoplada a um *host* que tem a responsabilidade de servir como porta de acesso do usuário à arquitetura. O *host* tem as funções de enviar os *templates* e dados para a arquitetura paralela, receber o(s) resultado(s) do processamento e apresentá-lo(s) ao usuário. Com isso, o processamento fica a cargo da arquitetura paralela enquanto o *host* apenas fornece suporte para o processamento.

No *host* também está o ambiente de desenvolvimento/operação, pois contém o *software* montador para a geração dos *templates*, e o *software* de interfaceamento para o envio dos *templates* para processamento e recebimento do(s) resultado(s) da arquitetura. Neste caso em particular, o *host* é um microcomputador e utiliza uma interface serial USB (*Universal Serial Bus)* para a conexão com a arquitetura. Contudo, para facilitar a interligação do *host* com a arquitetura, colocou-se entre eles um processador NIOS II da Altera que é embarcado junto com a arquitetura paralela em FPGA. Isto facilita em termos de conexão entre a arquitetura e o *host*, neste caso um microcomputador PC, pois utiliza-se uma interface padrão de comunicação como é a USB.

3.3 DESCRIÇÃO DA ARQUITETURA

A arquitetura, mostrada em detalhe na Figura 13, é composta por seis componentes básicos: Memória de *Templates* (MT) – utilizada para armazenamento dos *templates*, Memória de Dados (MD) – utilizada para armazenamento dos dados, Unidade de Armazenamento (UA) – responsável pela atualização dos valores nas memórias de *templates* e de dados, Unidade de Despacho (UD) – responsável pelo envio dos *templates* prontos para processamento para os EPs, Elementos Processadores (EPs) – efetivamente executam o processamento dos *templates* e Interface – serve para controlar a comunicação entre a arquitetura e o *host*. O Diagrama em blocos da implementação da arquitetura completa está mostrado no Anexo 1.

As unidades básicas da arquitetura (despacho, armazenamento, EPs e as memórias MT e MD) operam simultaneamente. Enquanto a unidade de despacho envia *templates* para um ou mais EPs, os demais podem continuar com o processamento. Isto ocorre ao mesmo tempo em que a unidade de armazenamento atualiza os *templates* processados na MT.

A UA é composta por três subunidades: UA_Armazenamento – que controla a atualização nos *templates* dos dados gerados pelo processamento dos EP; UA_Atualiza – recebe as requisições de atualização da MD; e UA_Snoop – responsável pela alteração do estado dos *templates* em função da atualização da MD.

A UD também é composta por três subunidades: UA_Consulta – responsável por buscar os *templates* da MT, os dados da MD e montar os *templates* para serem processados; UD_Buffer – armazena os *templates* prontos para o despacho; e UD_Despacho – que efetua o envio dos *templates* para os EPs.

Se estas subunidades fossem decompostas em processos poderia ter 36 processos simultâneos operando na arquitetura:

- 4 UD_Consulta;
- 4 UD_Despacho;
- 16 EPs;
- 8 UA_Armazenamento;
- 4 UA_Snoop.

Desta forma, garante-se, o paralelismo não somente com as unidades operativas (16 EPs), mas também com as unidades funcionais (UD_Consulta, UD_Despacho, UA_Armazenamento e UA_Snoop).

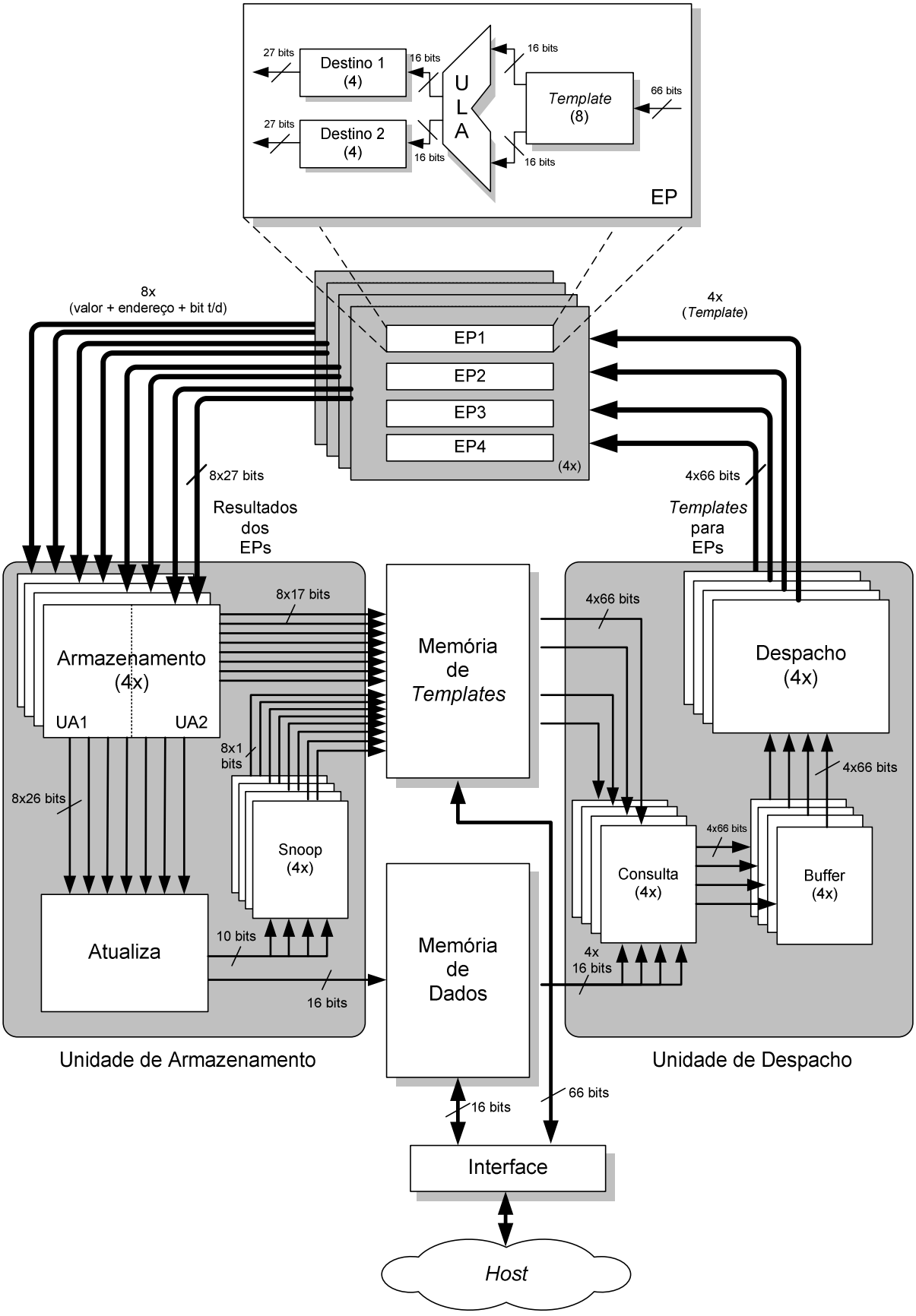


Figura 13: Visão detalhada da arquitetura paralela reconfigurável

3.3.1 *Templates*

O *template* corresponde à instrução que o EP executa e possui uma estrutura composta, basicamente, por cinco campos: Operação, Operando 1, Operando 2, Destino 1 e Destino 2, como mostrado na Figura 14. Cada *template* agrega todas as informações necessárias para sua execução. A operação indica a função a ser realizada. Os operandos armazenam os dados que serão processados e os destinos contêm as informações para onde o resultado é encaminhado.

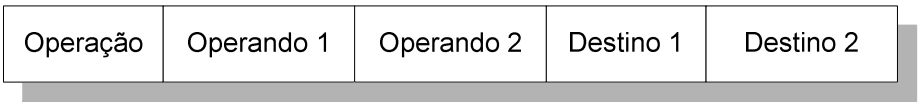


Figura 14: Estrutura básica do *template*

Os *templates* possuem a seguinte estrutura física: Operação (16 possíveis) a ser realizada (4 bits), dois Operandos (16 bits) e dois Destinos (endereços de 10 bits), como mostrado na Figura 15, além dos *bits* de controle (M, Válido, *Stick*, Porta, T/D). Com base nestas informações, percebe-se que o *template* contém uma operação a ser executada com dois operandos, no máximo, podendo executar operações monádicas e diádicas. Outra característica é que cada *template* pode armazenar o endereço de no máximo outros dois destinos (*templates* ou dados). Esta última característica garante a economia de uma operação de transferência de dados, pois o resultado pode ser enviado para dois destinos distintos em uma única operação, e isto contempla a maioria das situações.

Operação	Operando 1	M 1	Válido 1	*Stick* 1	Operando 2	M 2	Válido 2	*Stick* 2	Destino 1	Porta 1	T/D 1	Destino 2	Porta 2	T/D 2
4 bits	16 bits	1 bit	1 bit	1 bit	16 bits	1 bit	1 bit	1 bit	10 bits	1 bit	1 bit	10 bits	1 bit	1 bit

Figura 15: Formato do *template*

O *template* tem o formato apresentado na Figura 15 e possui um tamanho de 66 *bits*, em que:

- Operação – indica a operação a ser realizada (4 *bits*);
- Operando 1 e 2 – armazena o valor de ambos os operandos (16 *bits*);
- M 1 e 2 – indica se o operando é um dado da MD – "0" Não e "1" Sim (1 *bit*);
- Válido 1 e 2 – indica se o operando está atualizado (1 *bit*);
- *Stick* 1 e 2 – indica o que deve ser feito com os operandos após o *template* ter sido despachado – "0" descartar e "1" preservar (1 *bit*);

- Destino 1 e 2 – contêm o endereço dos *templates* destinos (10 *bits*);
- Porta 1 e 2 – indica qual é a porta do *template* destino (operando) que deve ser atualizado – "0" operando 1 e "1" operando 2 (1 *bit*);
- *T/D* 1 e 2 – indica se o destino é um *template* ou uma posição da MD – "0" *template* e "1" posição da MD (1 *bit*).

O Bit_Porta só é utilizado caso o Destino seja um *template*, pois se o Bit_T/D for "1" informando que é uma posição da MD, este indicativo do operando (porta) não tem sentido.

Algumas informações dos *templates* são fixas durante todo o processamento já que são decorrentes da lógica da aplicação. Na verdade, as únicas informações que podem ser modificadas na execução do programa são Operandos (1 e 2), Bits_Válido (1 e 2) e, eventualmente, Bits_Stick (1 e 2), mas isto depende da programação.

3.3.2 Memória de *Templates*

A aplicação a ser processada é decomposta em um conjunto de *templates* e que são armazenados na Memória de *Templates* (MT) durante o processamento. Salienta-se que nos *templates*, além da operação, também são armazenados os dados que serão processados.

A MT, do ponto de vista lógico, é uma memória única, que armazena uma quantidade máxima de *templates*. Fisicamente esta memória é dividida em vários módulos separados, cada qual com um controlador de memória. Os controladores de memória operam simultaneamente e gerenciam a busca dos *templates* em paralelo. Esta abordagem possibilita que o processamento tenha um despacho mais uniforme, desde que os *templates* a despachar não estejam no mesmo módulo, independentemente da quantidade de *templates* do programa. Além disto, esta organização da memória tende a diminuir a ociosidade dos EPs durante o processamento.

O esquema de segmentação da memória utiliza o Princípio da Localidade, no qual observa-se que as referências à memória efetuadas durante um intervalo de tempo, tendem a acessar uma região específica da memória (TANENBAUM, 2007). Deste modo, a probabilidade de que os *templates* consecutivos possam ser executados simultaneamente é grande ou pelo menos bem maior do que em outros casos. Com isto, segmentou-se a memória de forma que cada módulo físico tenha posições lógicas intercaladas, minimizando os possíveis atrasos no acesso à MT.

A adoção nesta proposta do modelo *Data Driven*, como descrito por VEEN (1986), é particularmente interessante, pois não há problema com o compartilhamento da memória, já que é a própria execução que disponibiliza os dados para os próximos *templates*. Desta forma, a coerência dos dados da memória é preservada e não há necessidade de endereçar a memória, em virtude de não haver variável compartilhada.

Esta memória possui capacidade para armazenar 1K *templates* de 66 *bits* e é segmentada em quatro pedaços iguais de memória, nas quais as posições são intercaladas, como mostrado na Figura 16, no qual o parâmetro *n* tem 10 *bits* para possibilitar que tenha 1K posições. Em cada posição da MT fica armazenado um *template*. O diagrama em blocos da implementação da MT está apresentado no Anexo 4.

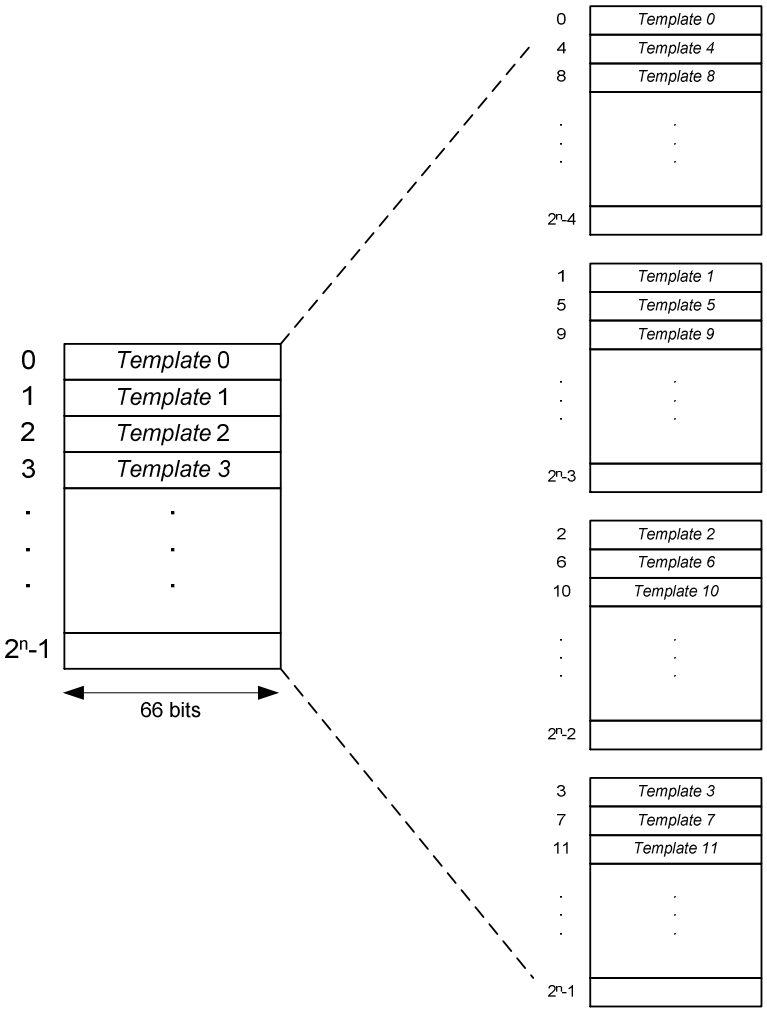


Figura 16: Memória de *templates*

Esta memória em termos lógicos é apresentada como um módulo único com largura de 66 *bits*, mas em termos físicos é particionada em três fatias: um para cada operando (1 e 2)

juntamente com seu Bit_M, Bit_Válido e Bit_Stick; e outro para a operação e as informações de controle (destinos, portas e Bit_T/D), como apresentado na Figura 17.

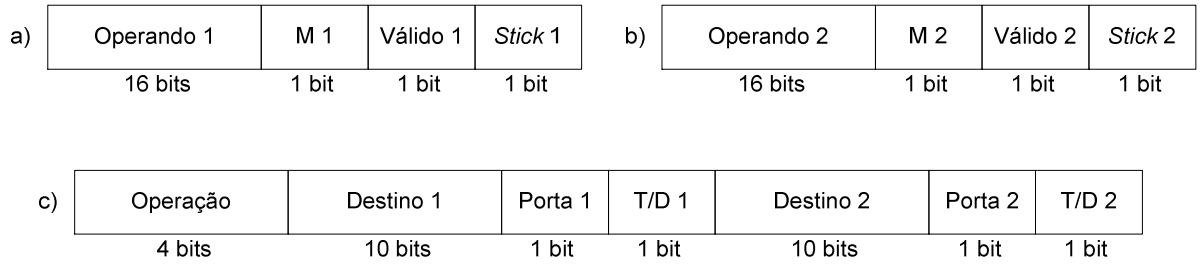


Figura 17: Particionamentos da memória de *templates*

A MT é particionada em módulos físicos, nos quais as posições lógicas são intercaladas para possibilitar acessos simultâneos, reduzindo o tempo de acesso à memória pelas UA e UD.

Um aspecto importante para se destacar é que a MT tem duplo acesso (*dual-port*), uma para a UA e a outra para UD. Isto facilita as buscas e atualizações. Diminui os tempos de despacho e de armazenamento no envio e atualização dos *templates*. Possibilita também, que tanto a UA quanto a UD executem acessos simultâneos, melhorando o desempenho do sistema.

A MT é modificada pela UA para atualizar os dados (operandos) nos *templates* e alterar o Bit_Válido, utilizando para isso uma das *dual-port* correspondente. Nesta operação os pedaços da memória podem ser acessados simultaneamente. Ressalta-se que as fatias de memória correspondendo aos Operandos 1 e 2 podem ser acessadas independentemente, possibilitando dois acessos simultâneos a cada pedaço da MT, totalizando oito acessos simultâneos em toda a MT. Com o intuito de minimizar ainda mais o impacto dos atrasos decorrentes desta atualização da MT, a UA utiliza as informações dos destinos armazenados nos EPs.

Os *templates* da MT são buscados pela UD, utilizando a segunda porta *dual-port*. Entretanto, ressalta-se que nesta operação todas as fatias são acessadas em uma única operação. Com isto, pode-se buscar quatro *templates* simultaneamente, um em cada pedaço da memória.

Inicialmente, esta memória é carregada com o programa, na forma de *templates*, que é proveniente do *host*. Após este estágio, a memória somente é lida pela UD para buscar os *templates* para serem processados e pela UA para atualizar os operandos dos *templates*.

3.3.3 Memória de Dados

A Memória de Dados (MD) é composta por 1K palavras de 16 *bits* e serve para armazenar as estruturas de dados como no caso de vetores. Esta memória é única para toda a arquitetura, mas os acessos de leitura e escrita somente são realizados pela UA e UD. Os EPs não têm acesso a esta memória, pois recebem os dados incorporados nos *templates* e já prontos para processamento.

O ideal é que a aplicação não utilize esta memória, pois isto acarreta um gasto adicional de ciclos de *clock*, resultando em um tempo de atraso, tanto na escrita quanto, principalmente, na leitura dos dados.

Esta memória também é de duplo acesso (*dual-port*) para reduzir o tempo de acesso e as posições são numeradas de 0 a (2^m-1), como mostrado na Figura 18. Neste caso, o parâmetro *m* é de 10 *bits* para possibilitar um endereçamento de 1K posições. Da mesma forma que a MT, uma das portas *dual-port* é utilizada pela UA e a outra pela UD. O diagrama em blocos da implementação da MD está mostrado no Anexo 5.

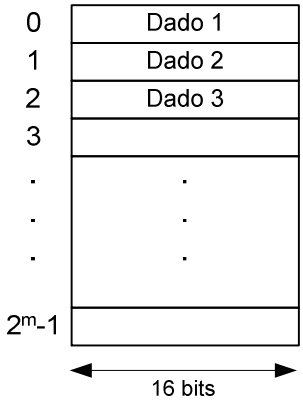


Figura 18: Memória de Dados

Esta memória é carregada com os dados enviados pelo *host* imediatamente antes do processamento e após o carregamento dos *templates* na MT.

3.3.4 Organização

A arquitetura, do ponto de vista lógico, pode ser entendida como um sistema composto por quatro módulos Fluxo de Dados (FDs) cada qual com quatro EPs, uma MT e uma MD

interconectados por meio de barramentos paralelos: um compartilhado para a MD e outros paralelos com a MT, como mostrado na Figura 19.

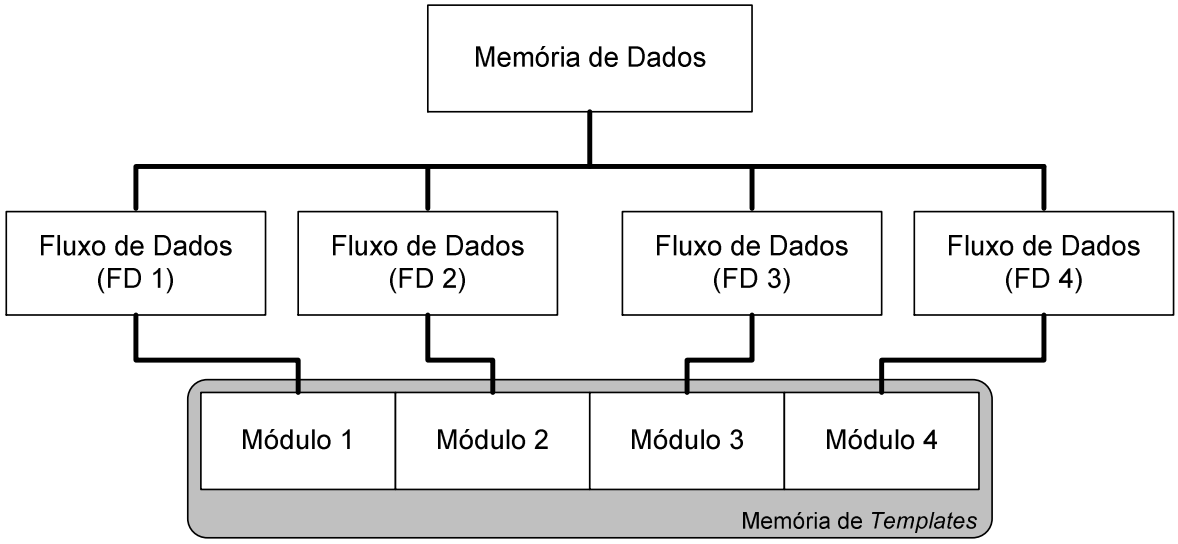


Figura 19: Diagrama lógico da arquitetura

Nesta arquitetura, a MD é centralizada e compartilhada pelos módulos FDs e é usada para armazenar estruturas lógicas e dados constantes durante o processamento. Em virtude da estrutura de conexão ser centralizada deve-se usar esta memória com parcimônia, minimizando a sobrecarga no acesso e os atrasos decorrentes que podem prejudicar o desempenho da arquitetura. Nesse caso, tem-se um modelo UMA (*Uniform Memory Access*) em que os módulo FDs possuem acesso à toda a memória.

Por sua vez a MT é distribuída e cada um dos módulos FDs acessa um dos pedaços desta memória. Apesar desta memória ter um único espaço de endereçamento linear, os módulos são logicamente intercalados e fisicamente distribuídos, possibilitando acessos simultâneos pelos FDs. Essa configuração caracteriza um modelo NUMA (*Non - Uniform Memory Access*) no qual cada módulo FD tem acesso somente ao seu pedaço de memória.

A distribuição dos *templates* para os EPs é realizada de forma automática, pois os FDs podem acessar somente os *templates* que estão nos seus pedaços da MT e estes pedaços são intercalados. Desta maneira, o próprio carregamento dos *templates* na MT faz esta distribuição. A idéia para esta distribuição foi baseada no Princípio da Localidade, como descrito no item 3.3.2, no qual os *templates*, subseqüentes ao atual, têm uma maior probabilidade de serem utilizados, de modo que esta distribuição intercalada proporciona uma estratégia viável para uma arquitetura fluxo de dados.

3.3.5 Estrutura Lógica

A estrutura lógica desta arquitetura paralela se assemelha a uma configuração de um *cluster* de multiprocessador com quatro nós, cada qual com quatro EPs. Há também uma MD compartilhada, mas somente é acessada pelos nós e não pelos EPs. Cada nó do *cluster* é composto por quatro EPs, duas UAs, uma UD e uma MT, formando uma máquina Fluxo de Dados (FD) com quatro unidades funcionais. Na Figura 20 está apresentada a estrutura lógica da arquitetura paralela, com detalhamento do nó do *cluster*.

Nesta configuração pode-se entender que há um ambiente paralelo homogêneo com dois modelos de acesso à memória: UMA para a MD e NUMA para a MT, aproveitando o melhor de ambos os casos, ou seja, consistência de dados na UMA e paralelismo no acesso na NUMA.

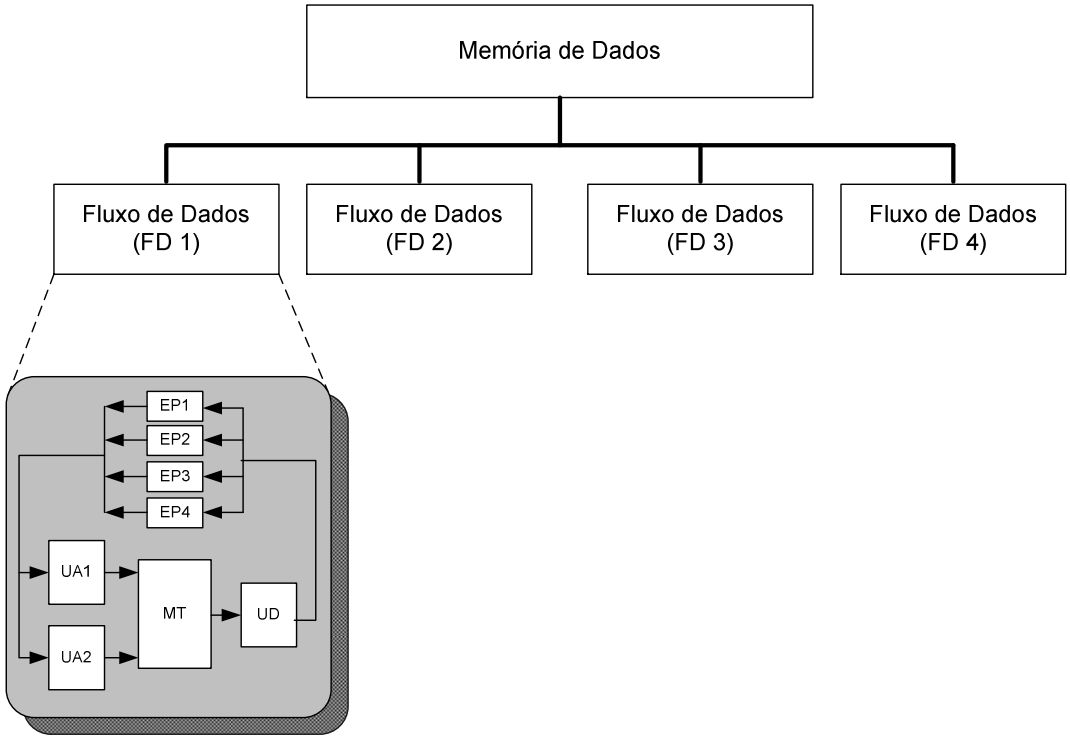


Figura 20: Estrutura lógica da arquitetura

Em uma estrutura ideal, cada FD teria apenas um EP, pois evitaria a concorrência entre os EPs no acesso às outras unidades. Contudo, isto causa um grande impacto na quantidade de portas lógicas consumidas para a replicação das unidades e para a conexão destes componentes, limitando o número de EPs na implementação em FPGA. O diagrama em blocos da implementação da arquitetura com 4 EPs (módulo FD) está mostrado no Anexo 2.

Outro ponto a ser salientado é que o número de FDs tem relação direta com o número de pedaços da MT, pois cada FD tem sua própria MT. Conseqüentemente, uma maior quantidade de FDs implica em um maior número de pedaços da MT e isto influencia o consumo de portas lógicas do FPGA para implementar as conexões e lógica de controle necessárias.

Os testes preliminares indicaram que a divisão da MT em quatro pedaços teria a melhor relação custo-benefício, pois sendo este número par, facilita a segmentação da MT em pedaços com tamanhos iguais. Outro ponto é que o mapeamento do endereçamento lógico para o físico é menos complexo, reduzindo o gasto de portas lógicas com a lógica de controle. Como se estabeleceu que a arquitetura deveria ter 16 EPs, isto fez com que cada pedaço da MT tivesse um conjunto de quatro EPs.

A decisão de ter duas UAs por pedaço da MT é decorrente do *template* ter dois operandos. O objetivo neste caso é o de minimizar os atrasos decorrentes de possíveis concorrências no acesso aos pedaços da MT. Desta forma, podem ser realizados dois acessos simultâneos, um para cada operando, inclusive se estes operandos forem pertencentes ao mesmo pedaço da MT.

3.3.6 Barramentos

Um problema usual nas arquiteturas paralelas é a maneira como os componentes são interconectados, principalmente os EPs. Esta interconexão pode ocorrer em vários níveis, dependendo do tipo de acoplamento que se pretende adotar. Quanto mais forte for este acoplamento, maior é a matriz de interconexão e o oposto é verdadeiro. Como conseqüência disto, há um consumo maior ou menor de portas lógicas do dispositivo programável e isto afetará toda a arquitetura, pois os recursos do dispositivo são limitados.

Nesta proposta, utiliza-se um acoplamento, denominado fortemente acoplado, pois é importante que haja um caminho rápido entre os EPs e as unidades de despacho e de armazenamento, reduzindo o tempo de comunicação. O tempo de comunicação é um dos fatores que causa um grande efeito no tempo total de processamento e que, por decorrência, afeta também o desempenho geral do sistema. Para isto, utilizam-se barramentos paralelos entre as unidades, possibilitando um alto *throughput*.

A interconexão entre a UD e os EPs ocorre por meio de 4 barramentos paralelos com 66 *bits* cada. Esta largura corresponde ao tamanho do *template* da arquitetura. Cada um dos barramentos interconecta uma UD a um conjunto de quatro EPs. Desta forma, um grupo de

quatro EPs é atendido única e exclusivamente por uma UD, de modo que a UD1 atende os EP1 a EP4, a UD2 os EP5 a EP8 e, assim, sucessivamente.

Os EPs estão conectados à UA por meio de 8 barramentos paralelos de 27 *bits*, correspondendo a 16 *bits* de dados, 10 *bits* de endereço e o Bit_T/D para indicar se é *template* ou dado. Cada grupo de dois barramentos possibilita a conexão entre os quatro EPs de um FD e uma UA. As UAs não estão atreladas aos EPs ou a um FD, ou seja, qualquer EP pode ser atendido por uma das UAs, de modo que o EP1 pode enviar dados para qualquer UA, não somente para a UA1 ou UA2, às quais pertencem ao mesmo FD.

3.3.7 Ciclo de Processamento

Inicialmente, o *host* envia para a arquitetura o programa na forma de *templates*, que são instruções da arquitetura e que são armazenados na MT. Após esta etapa, a UD consulta a MT e encaminha para os EPs os *templates* para serem processados, na medida em que estiverem disponíveis. Os EPs, tão logo recebam os *templates*, podem efetuar as operações indicadas no campo operação e produzir os resultados, que serão armazenados na MT pela UA. Este processo se repete até que não haja mais *templates* para serem processados.

O ciclo de processamento desta arquitetura está mostrado no Algoritmo 1.

Algoritmo 1 – *Ciclo de Processamento na Arquitetura*

Descrição:

Este algoritmo descreve todo o processo de carregamento da aplicação e o seu processamento pela arquitetura paralela.

Funcionamento:

1: Carregamento da MT com os *templates* provenientes do *host*;
2: Carregamento da MD com os dados/estruturas provenientes também do *host*;
3: Aguarda sinal (*start*) para iniciar o processamento;
4: Quando o processamento for acionado, as cinco unidades funcionais são iniciadas e operarão simultaneamente até que a aplicação tenha sido finalizada:
   a) UD_Consulta – para saber se há *templates* para serem executados;
   b) UD_Despacho – envia para os EPs os *templates* prontos;
   c) EPs – controla a execução dos *templates*;
   d) UA_Armazenamento – armazena os operandos/dados nas memórias (MT e MD);
   e) UA_Snoop – verifica se há *templates* a serem liberados na MT em função da atualização dos dados na MD;
5: Envia para o *host* o conteúdo da MD;
6: Envia para o *host* o conteúdo da MT.

O funcionamento da arquitetura pode ser visualizado na máquina de estados geral, como mostrado na Figura 21. As etapas de carregamento da MT com os *templates* e da MD com os dados do *host* são executadas seqüencialmente e antecedem o processamento. Em

decorrência do sinal (*start*), as subunidades (UD_Consulta, UD_Despacho, EPs, UA_Armazenamento e UA_Snoop) são acionadas simultaneamente.

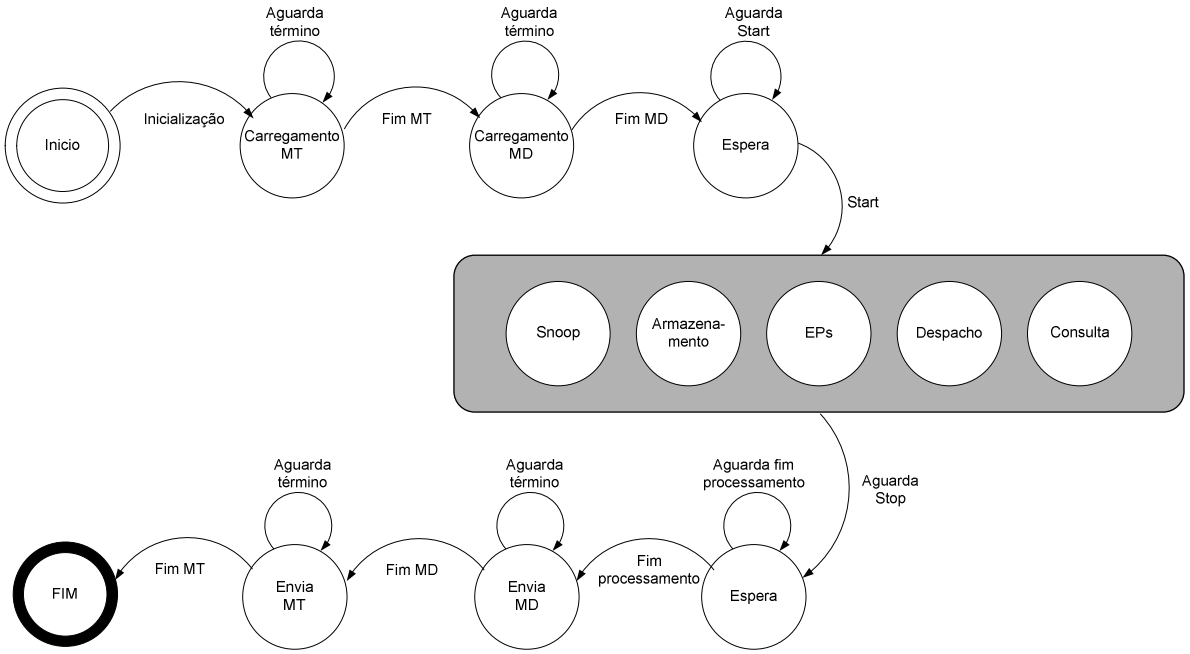


Figura 21: Máquina de estados geral da arquitetura

O escalonamento das operações ocorre de forma automática pela arquitetura, pois a lógica envolvida segue uma única diretriz: "a instrução pode ser executada desde que o(s) dado(s) esteja(m) disponível(eis), ou seja com o Bit_Válido setado, e, também, que haja um EP que esteja liberado".

3.3.8 Unidade de Armazenamento

A Unidade de Armazenamento (UA) é responsável por receber os resultados dos *templates* que foram processados pelos EPs e atualizar os valores nos *templates* que estão na MT para execuções futuras ou então atualizar os dados na MD. O diagrama em blocos da UA está apresentado na Figura 22, no qual o parâmetro (*n*) é a quantidade de *bits* de endereçamento da MT e o parâmetro (*m*) é a quantidade de *bits* de endereçamento da MD. Neste caso, *n* e *m* são de 10 *bits* possibilitando 1K posições de memória em ambas as memórias (MT e MD). No Anexo 7 está apresentada a listagem do código VHDL da UA.

A UA é composta por três subunidades básicas: a) UA_Armazenamento que recebe os resultados dos EPs e atualiza os *templates* na MT; b) UA_Atualiza que atualiza os dados na

MD; e c) UA_Snoop que modifica o estado dos *templates* nos quais o(s) operandos(s) são dados da MD.

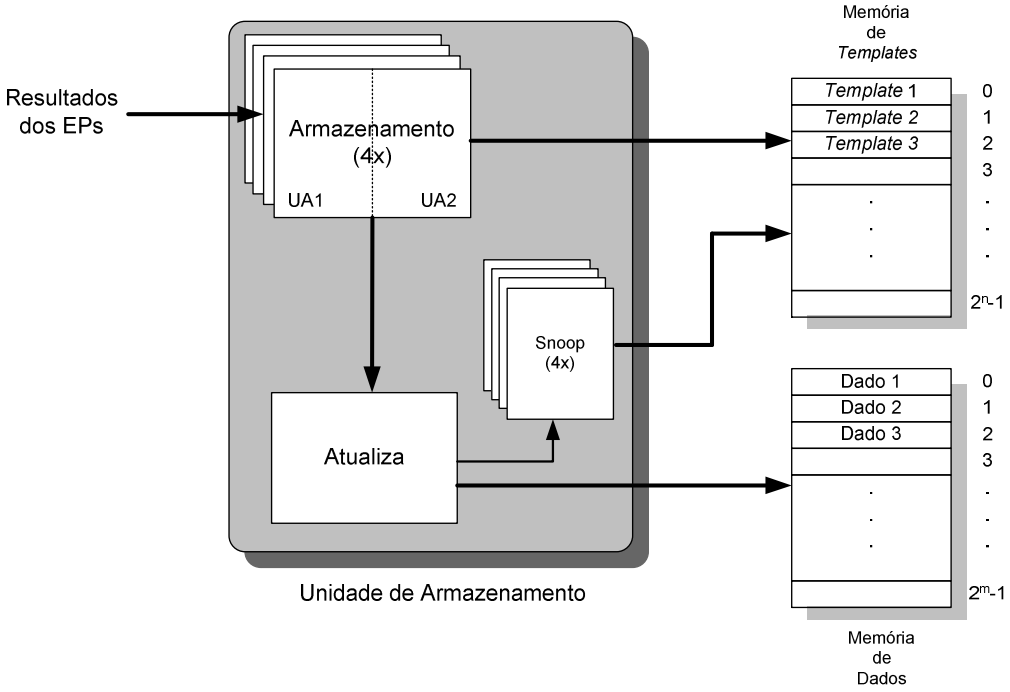


Figura 22: Estágio de armazenamento

A UA utiliza os dados provenientes dos EPs para atualizar a MT e a MD. A comunicação entre os EPs e a UA ocorre por meio de 8 barramentos paralelos (27 *bits*) contendo o valor (16 *bits*), destino (10 *bits*) e Bit_T/D (1 *bit*). A comunicação entre a UA e a MT também é por meio de 8 barramentos paralelos de 17 *bits* (valor - 16 *bits* e Bit_Válido - 1 *bit*) e com a MD é por meio de um barramento de 16 *bits* (valor - 16 *bits*).

Salienta-se que somente a UA tem permissão de escrita na MT e na MD utilizando uma das entradas das memórias *dual-port*.

- **UA_Armazenamento**

A subunidade UA_Armazenamento é composta por quatro componentes idênticos, um para cada pedaço da MT. Cada componente opera somente com o seu pedaço da memória. O componente é constituído de duas partes, uma para acesso à fatia do Operando 1 e outra para acesso à fatia do Operando 2. Com este arranjo pode-se atualizar oito (4x2) *templates* simultaneamente. As informações oriundas dos EPs são atualizadas nos *templates* da MT, com exceção dos dados. Estes são enviados à MD.

Os quatro componentes UA_Armazenamento efetuam o controle do armazenamento dos operandos (1 e 2) e dos dados, seguindo a seqüência de passos descrita no Algoritmo 2.

Algoritmo 2 – *Ciclo de Operação da UA_Armazenamento*

Descrição:

Este algoritmo descreve todo o processo de gerenciamento da operação da UA_Armazenamento.

Funcionamento:

1: Consulta os *buffers* de saída dos EPs para saber se há operandos/dados para serem atualizados;
2: Se houver valores, busca os operandos/dados;
3: Inicia o processo de atualização dos operandos/dados;
4: Neste passo podem ocorrer 3 sub-passos simultaneamente:
   a) Se houver Operando 1, atualiza-o na MT;
   b) Se houver Operando 2, atualiza-o na MT;
   c) Se houver dado, encaminha p/ UA_Atualiza;
5: Volta ao **Passo 1** para armazenar outro resultado gerado pelos EPs.

Um detalhamento maior pode ser visto na máquina de estados UA_Armazenamento, mostrado na Figura 23. Esta máquina de estados é iniciada pelo sinal (*start*) que é gerado pela máquina de estados geral. Esta máquina de estados nunca é finalizada, a não ser que ocorra um *Reset*.

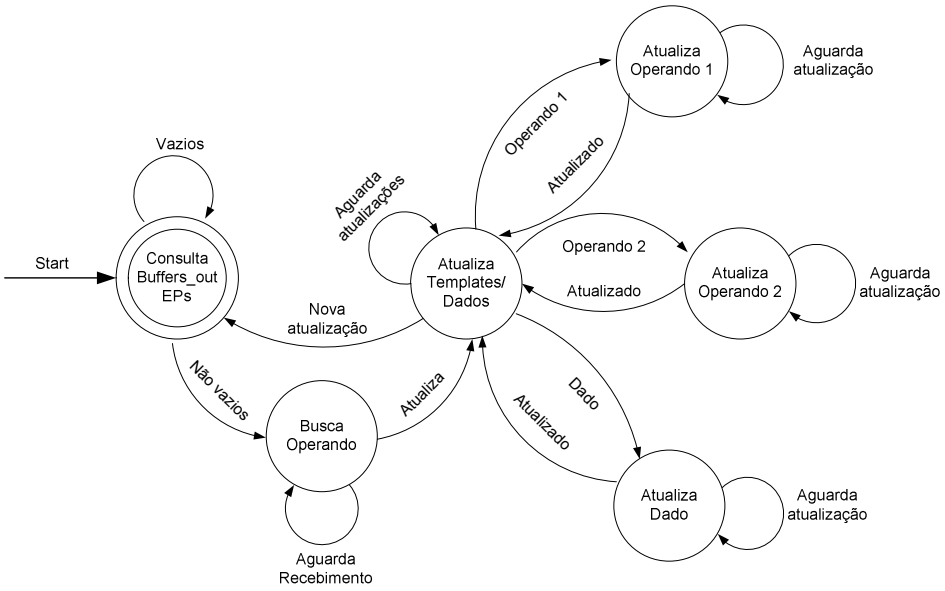


Figura 23: Máquina de estados de controle da UA_Armazenamento

- **UA_Atualiza**

Uma outra subunidade é a UA_Atualiza que armazena temporariamente as solicitações de atualização dos dados na MD. Esta subunidade é constituída por uma fila (*buffers*) com oito posições contendo endereço (10 *bits*) e valor (16 *bits*). As requisições de escrita são

processadas exclusivamente de forma seqüencial, evitando incoerências de dados na memória. Esta subunidade recebe as informações referentes às atualizações da MD de cada um dos componentes da UA_Armazenamento. Estes dados são enfileirados e atualizados seqüencialmente na MD, reduzindo os problemas decorrentes da coerência de dados nesta memória.

- **UA_Snoop**

A terceira subunidade é a de UA_Snoop que fica "escutando" o barramento da MD e toda vez que um dado é atualizado na memória, esta subunidade realiza a alteração do estado dos operandos que possuem o mesmo endereço de memória. Esta subunidade opera simultaneamente as demais e de forma autônoma, pois toda a vez que um dado for atualizado na MD esta subunidade efetuará a habilitação (Bit_Válido) dos respectivos operandos nos *templates* na MT.

Outra característica desta subunidade é a de ser constituída por quatro componentes iguais, um para cada pedaço da MT, possibilitando uma redução no tempo de atraso decorrente do acesso a esta memória. Desta forma, as subunidades UA_Armazenamento e UA_Snoop concorrem no acesso à MT, possibilitando até oito acessos simultâneos.

O ciclo de operação da UA_Snoop é apresentado no Algoritmo 3.

Algoritmo 3 – *Ciclo de Operação da UA_Snoop*

Descrição:

Este algoritmo descreve o funcionamento da UA_Snoop.

Funcionamento:

1: Verifica se o *buffer* da UA_Atualiza está vazio;
2: Se houver valores, busca os operandos/dados;
3: Inicia o processo de atualização dos operandos/dados;
4: Caso tenha dado, ocorrerá duas sub-sequencias:
   a) Atualização/armazenamento dos dados na MD e vai para o **Passo 3**;
   b) Efetua o UA_Snoop do endereço com os endereços catalogados;
   c) Realiza a varredura da MT para encontrar os endereços iguais, caso chegue ao fim da memória, vai para o **Passo 3**;
   d) Quando encontra um endereço igual, altera o estado do Bit_Válido deixando-o disponível para processamento;
   e) Volta para o **Passo 4c** para continuar a varredura da MT.
5: Efetua o sincronismo entre a atualização da MD e a atualização do *template* na MT;
6: Quando todas as atualizações estiverem efetuadas, volta ao **Passo 1** para nova atualização de dados.

Este ciclo é executado pelos quatro componentes UA_Snoop de forma autônoma e simultânea. Um detalhamento maior pode ser visto na máquina de estados UA_Snoop na

Figura 24. A máquina de estados UA_Snoop entra em operação mediante o sinal (*start*) que é gerado pela máquina de estados geral. Esta máquina de estados nunca é finalizada, a não ser que ocorra um *Reset*.

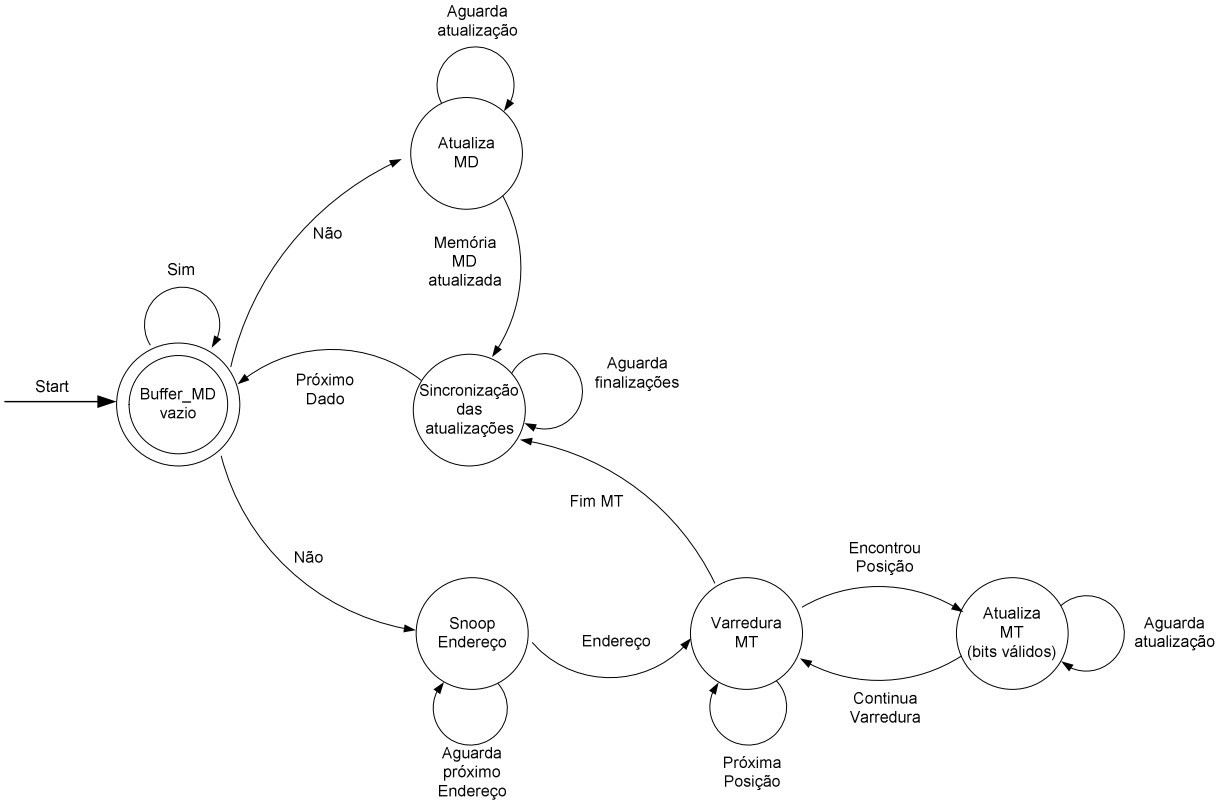


Figura 24: Máquina de estados de controle da UA_Snoop

3.3.9 Unidade de Despacho

A Unidade de Despacho (UD) é outro elemento-chave, pois é responsável pela busca dos *templates* prontos na MT e o envio destes *templates* para os EPs, à medida que estiverem livres para novo processamento. O diagrama em blocos desta unidade é mostrado na Figura 25. Como descrito anteriormente, o parâmetro *m* indica o número de *bits* de endereço da MD e o *n* a quantidade de *bits* no endereço da MT. Neste caso, *n* e *m* são de 10 *bits* e isto possibilita 1K posições em ambas as memórias (MT e MD). No Anexo 6 está apresentada a listagem do código VHDL da UD.

Esta unidade faz a leitura do(s) *template*(s) pronto(s) para a execução, consulta os EPs para saber quais estão disponíveis e envia o(s) *template(s)*. Caso encontre a instrução *Stop*, então pára de buscar novos *templates*, fazendo com que os EPs sejam esvaziados à medida em que não há novos envios de *templates* para processamento. Com isto, os EPs concluem os

templates em andamento e finalizam por completo o processamento da aplicação. O *template* é considerado pronto quando os Bits_Válido dos operandos estão ativos.

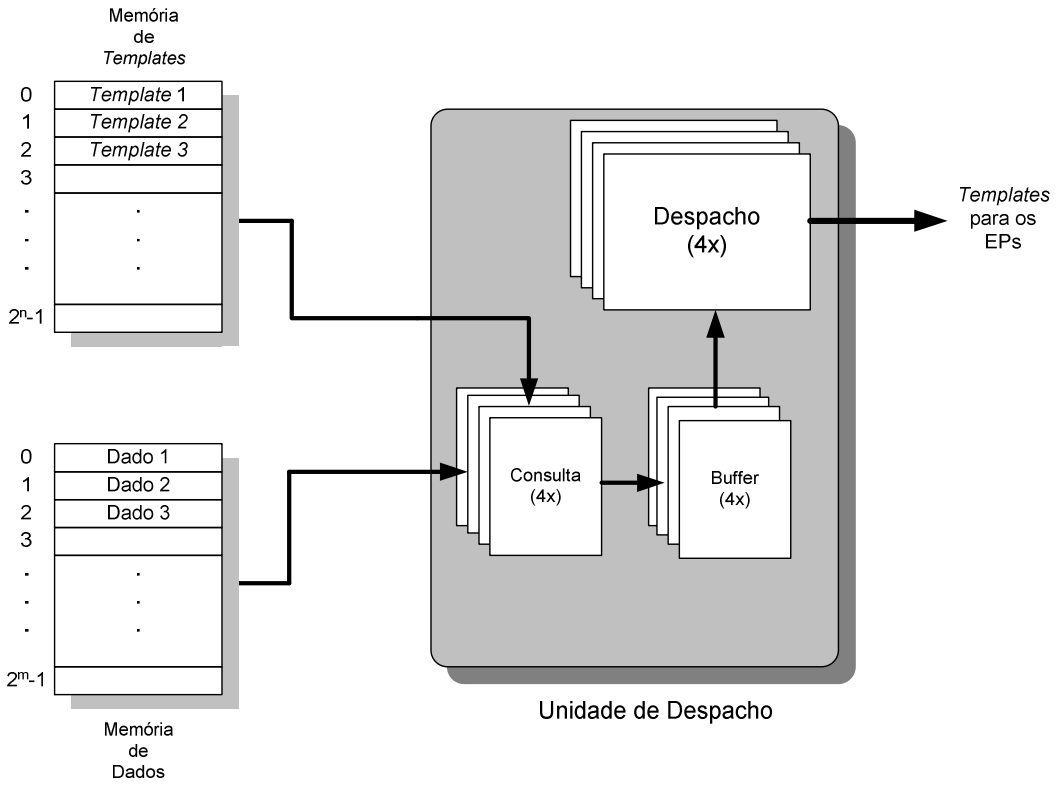


Figura 25: Estágio de despacho

A UD é composta por três subunidades: a) UD_Despacho – responsável pelo envio dos *templates* prontos para os EPs; b) UD_Buffer – que armazena os *templates* prontos para a execução; c) UD_Consulta – que busca os *pseudo-templates* da MT e monta os *templates* que serão armazenados no UD_Buffer. Considera-se *pseudo-template* quando ele utiliza dados da MD e que precisam ser atualizados no *template* antes de serem processados. A UD utiliza os dados provenientes da MT e da MD para montar o(s) *template*(s).

A comunicação entre a UD e a MT e entre a UD e os EPs é por meio de quatro barramentos paralelos de 66 *bits* que é a largura do *template*. A comunicação entre a UD e a MD ocorre por meio de um barramento de 16 *bits* que é a largura do dado da arquitetura.

- **UD_Despacho**

A UD_Despacho é composta por quatro componentes iguais que enviam para os EPs os *templates* prontos e que estão armazenados no UD_Buffer. Nesta configuração, cada

componente envia os *templates* oriundos de um dos quatro pedaços da MT para os EPs, por meio de quatro barramentos paralelos. Estes barramentos possuem a largura do *template*. Atrelado a cada componente UD_Despacho há quatro EPs. Os quatro componentes UD_Despacho trabalham independentes e com isso podem despachar até quatro *templates* simultaneamente.

Os componentes da UD_Despacho obedecem a um ciclo básico mostrado no Algoritmo 4.

Algoritmo 4 – *Ciclo de Operação da UD_Despacho*

Descrição:

Este algoritmo descreve o funcionamento da UD_Despacho.

Funcionamento:

1: Verifica se há *templates* prontos no UD_Buffer;
2: Caso haja *templates* prontos verifica a disponibilidade de EPs;
3: Se as duas condições forem satisfeitas, despacha o *template* para o EP designado e volta ao **Passo 1** para novo despacho. Caso contrário aguarda até que a(s) condição(ões) seja(m) satisfeita(s).

Este ciclo ocorre de forma independente para os quatro componentes e um maior detalhamento pode se visto na máquina de estados UD_Despacho, na Figura 26. A inicialização desta máquina de estados é pelo sinal (*start*) gerado pela máquina de estados geral. Esta máquina de estados nunca é finalizada, a não ser que ocorra um *Reset*.

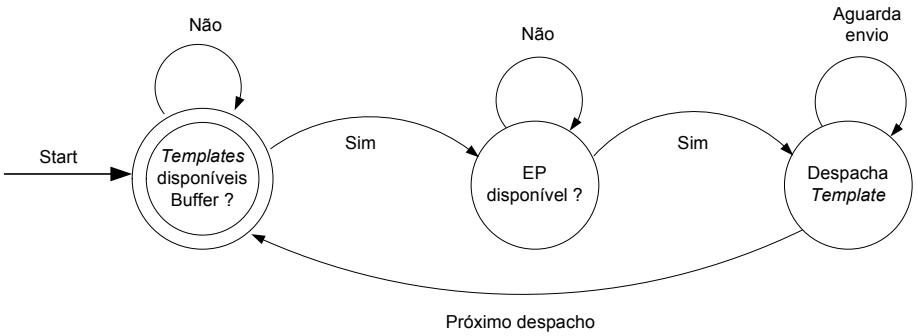


Figura 26: Máquina de estados de controle da UD_Despacho

- **UD_Buffer**

O UD_Buffer é constituído por quatro *buffers* paralelos e servem para armazenar os *templates* prontos, que foram montados pela UD_Consulta, enquanto a UD_Despacho está enviando os *templates* armazenados. Os *buffers* são estruturas FIFO (*First In, First Out*) e armazenam no máximo quatro *templates*, que correspondem ao número de EPs atrelados a

cada UD_Despacho. A comunicação entre a UD_Buffer e a UD_Consulta também ocorre por meio de quatro barramentos paralelos com a mesma largura dos *templates*.

- **UD_Consulta**

A subunidade UD_Consulta verifica os Bits_Válidos para determinar quais os *templates* que estão prontos e que podem ser executados. Contudo, caso algum *template* tenha operando que seja um endereço da MD (Bit_M igual a "1"), esta subunidade efetua a substituição do endereço pelo valor correspondente a esta posição na memória. Não havendo a necessidade desta substituição, o *template* é despachado diretamente para o UD_Buffer. Este processo depende exclusivamente da disponibilidade de entradas livres nos *buffers*.

Da mesma forma que outras subunidades, a UD_Consulta é composta também por quatro componentes iguais, um para cada pedaço da memória, possibilitando a busca de quatro *templates* simultaneamente. Um destaque é que os quatro componentes concorrem para o acesso à MD que, neste caso, é compartilhada, pois os acessos de leitura são seqüenciais, reduzindo os problemas decorrentes da coerência de dados nesta memória.

Os componentes UD_Consulta seguem o ciclo básico, como apresentado no Algoritmo 5.

Algoritmo 5 – *Ciclo de Operação da UD_Consulta*

Descrição:

Este algoritmo descreve o ciclo de operação da UD_Consulta.

Funcionamento:

1: Verifica se há *templates* prontos;
2: Busca o *template* pronto na MT;
3: Se houver dados, busca na MD;
4: Se for a instrução STOP:
   a) Interrompe a busca de *templates*;
   b) Aguarda a finalização dos *templates* enviados;
   c) Fim do processamento.
5: Se o *template* estiver pronto, armazena-o no UD_Buffer;
6: Volta ao **Passo 1** para buscar novo *template*.

Este ciclo ocorre simultaneamente nos quatro componentes UD_Consulta e pode ser visualizado na máquina de estados UD_Consulta, na Figura 27. A máquina de estados que controla a UD_Consulta é inicializada pelo sinal (*start*) proveniente da máquina de estados geral. Quando a instrução *Stop* for lida da MT, a máquina finaliza a busca dos *templates* e aguarda a atualização das memórias para gerar o sinal de término de processamento.

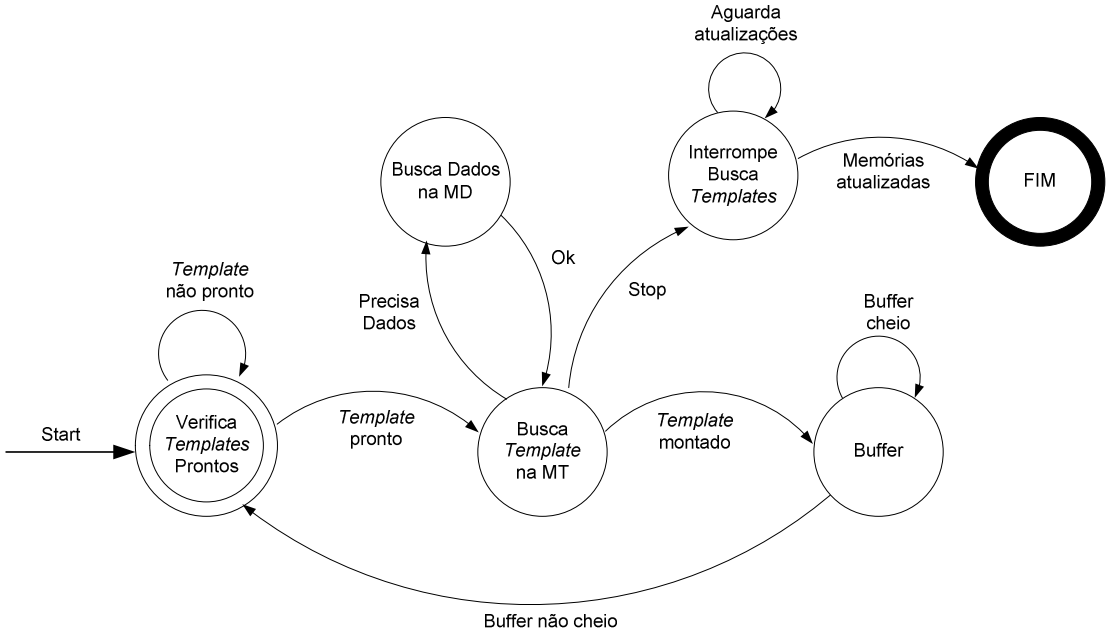


Figura 27: Máquina de estados de controle da UD_Consulta

3.3.10 Elementos Processadores

Os Elementos Processadores (EPs) são os responsáveis pelo processamento dos *templates*. Os EPs são idênticos e compostos basicamente por ULAs que realizam as operações como adição, subtração, multiplicação, divisão, lógicas, relacionais e condicionais. Além disso, cada EP possui também um conjunto de *buffers* de entrada e de saída para facilitar o sincronismo entre as unidades que se interligam aos EPs. Na Figura 28 é mostrado o diagrama em bloco do EP.

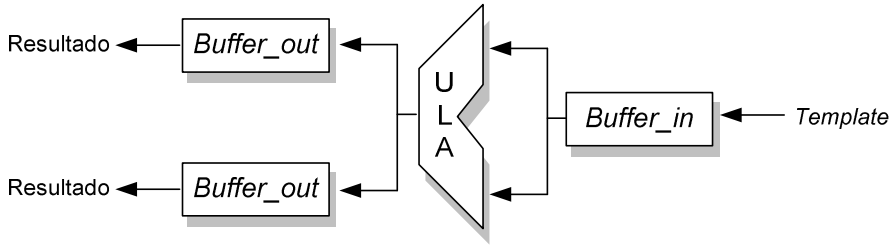


Figura 28: Diagrama em bloco do elemento processador (EP)

A estrutura do EP permite executar as operações mencionadas com até dois operandos (monádicas ou diádicas) e produz como saída um único resultado que pode ser encaminhado para dois destinos.

Um ponto básico adotado é que os EPs sejam simples, exigindo poucos recursos do FPGA, consumindo poucas portas lógicas para a sua implementação. Isto faz com que se consiga incorporar um maior número de EPs no dispositivo programável, pois quanto maior for este número, maior é a quantidade de operações executadas simultaneamente, resultando em aumento do desempenho.

O número de EPs é responsável por grande parte dos limites impostos ao desempenho da arquitetura, porque tem relação direta com a quantidade de operações que podem ser executadas em paralelo. Isto afeta diretamente a quantidade de barramentos paralelos que interligam os EPs às unidades de despacho e de armazenamento.

Obviamente, deve-se observar também as limitações da aplicação para se estabelecer o número de EPs presentes no computador paralelo. É neste ponto que entra em cena a escalabilidade, pois pode-se aumentar a quantidade de EPs em função da aplicação. Entretanto, os aspectos físicos da arquitetura são fatores impeditivos, justamente por se levar em consideração as capacidades dos dispositivos programáveis, tanto em termos de elementos lógicos quanto de interconexões.

Quando a unidade funcional opera com números inteiros diz-se que é uma ALU (*Arithmetic and Logic Unit*). Por outro lado, quando opera com números com ponto flutuante diz-se que é uma FPU (*Float-Point Unit*). Entretanto, neste trabalho adotar-se-á o termo ULA como forma genérica para indicar ambos os tipos de unidades.

A complexidade do EP está intimamente atrelada à aplicação e ao tipo da ULA utilizada, pois a ULA pode ser de inteiro ou de ponto flutuante, operando com palavras de 8, 16, 32 *bits*. Uma ULA de ponto flutuante de 32 *bits* consome mais recursos do que uma ULA de inteiro de 32 *bits*. Logo, a escolha correta do tipo de operação implica em um aproveitamento melhor da arquitetura.

Caso o EP seja composto por uma ULA de inteiro, consegue-se executar operações aritméticas, lógicas e relacionais, que pode ser utilizado em uma grande variedade de aplicações como criptografia, compactação, descompactação de dados e outras. A substituição desta ULA de inteiro por uma outra ULA de ponto flutuante deixa o EP mais adequado para aplicações que envolvem cálculos numéricos. Entretanto, isto acarreta um gasto adicional de tempo e de recursos do dispositivo programável.

Esta arquitetura contém 16 EPs, compostos por ULA de inteiro, todos iguais e agrupados em quatro conjunto com quatro EPs, formando um *cluster* 4x4, com operandos de 16 *bits*, que podem executar simultaneamente.

O EP possui uma estrutura simples, consistindo de uma ULA de inteiro com 16 *bits* e de *buffers* para armazenar tanto os *templates* (um *buffer_in* com 8 posições) quanto o resultado (dois *buffers_out* com 4 posições), como mostrado na Figura 29. A ULA realiza as operações aritméticas, lógicas, relacionais, condicionais e de duplicação. O diagrama em blocos da implementação do EP está mostrado no Anexo 3, e a listagem do código VHDL da ULA está no Anexo 8.

A incorporação dos *buffers* de entrada nos EPs possibilita minorar o problema da lentidão do acesso à memória, pois enquanto a MT estiver livre a UD pode efetuar uma busca antecipada dos *templates* disponíveis e encaminhá-los para os EPs.

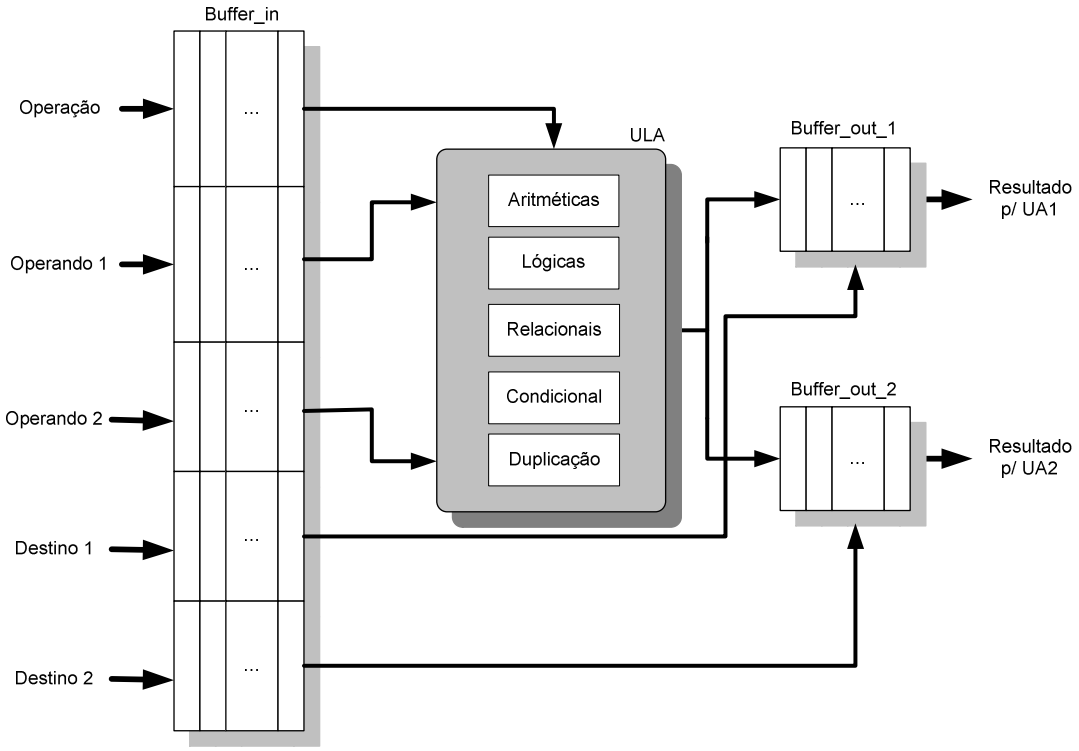


Figura 29: Estrutura do elemento processador (EP)

O EP recebe como entrada um *template* (com 66 *bits*) e produz como saída duas palavras iguais de 27 *bits* (dado, destino e Bit_T/D). Cada EP possui um conjunto de *buffers*/filas de entrada e de saída para minimizar os efeitos decorrentes dos atrasos de armazenamento das UAs. Salienta-se que UA1 atualiza o operando 1 e a UA2 o operando 2 nos *templates*.

Os EPs possuem uma autonomia no nível de função, em virtude de cada EP processar a sua própria operação, independentemente dos demais. Este é um ponto importante para se

obter taxas de paralelismo crescentes, pois não há dependências entre eles durante o processamento, além das impostas pela aplicação.

Os EPs são agrupados em um arranjo com quatro, reduzindo o número de barramentos de interligação com a UD (quatro barramentos de 66 *bits*), totalizando 264 *bits*, e com a UA (oito barramentos de 27 *bits*), totalizando 216 *bits*.

O funcionamento dos EPs pode ser descrito por uma seqüência de passos, como mostrado no Algoritmo 6.

Algoritmo 6 – *Ciclo de Operação dos EPs*

Descrição:

Este algoritmo descreve a seqüência de operações dos EPs.

Funcionamento:

1: Consulta Buffer_in para saber se tem *templates* para serem processados;
2: O EP executa a operação indicada no *template*;
3: Aguarda até que haja uma posição no Buffer_out (1 ou 2) disponível (vago);
4: Armazena resultado no Buffer_out (1 ou 2);
5: Volta ao **Passo 1** para processar novo *template*.

Cada EP segue o ciclo acima e um detalhamento maior pode ser visto na máquina de estados EP na Figura 30. Esta máquina de estados nunca é finalizada, a não ser que ocorra um *Reset*.

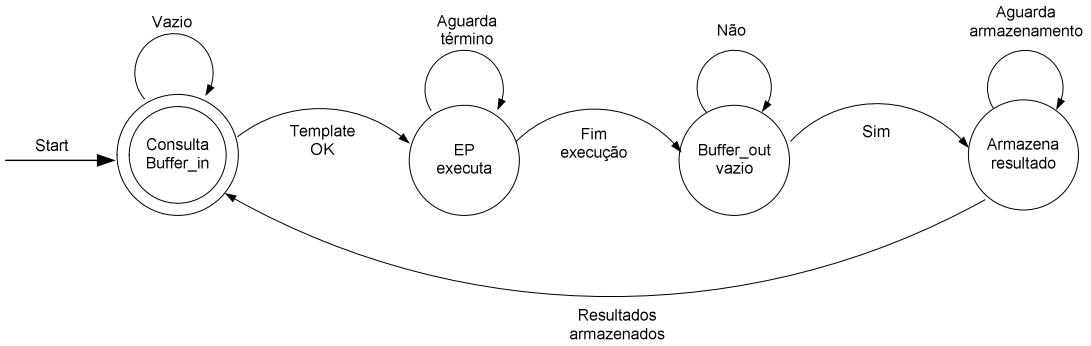


Figura 30: Máquina de estados de controle dos EPs

3.3.11 *Pipeline*

O *pipeline* é uma técnica de implementação em que várias instruções são sobrepostas na execução (STALLINGS, 2006). Este mecanismo tira proveito do paralelismo que existe entre as ações necessárias para executar uma instrução ou neste caso *template*.

A arquitetura apresenta um *pipeline* básico, mostrado na Figura 31, composto por três estágios: Despacho (UD), Processamento (EP1 a EP4) e Armazenamento (UA1 e UA2). Os

estágios apresentam ciclos de *clock* diferentes: UA (1 ciclo), EP (máx 8 ciclos) e UA (3 ciclos). Desta forma, um ciclo completo no *pipeline* é de, no máximo, 12 ciclos de *clock* para instruções com 16 *bits*.

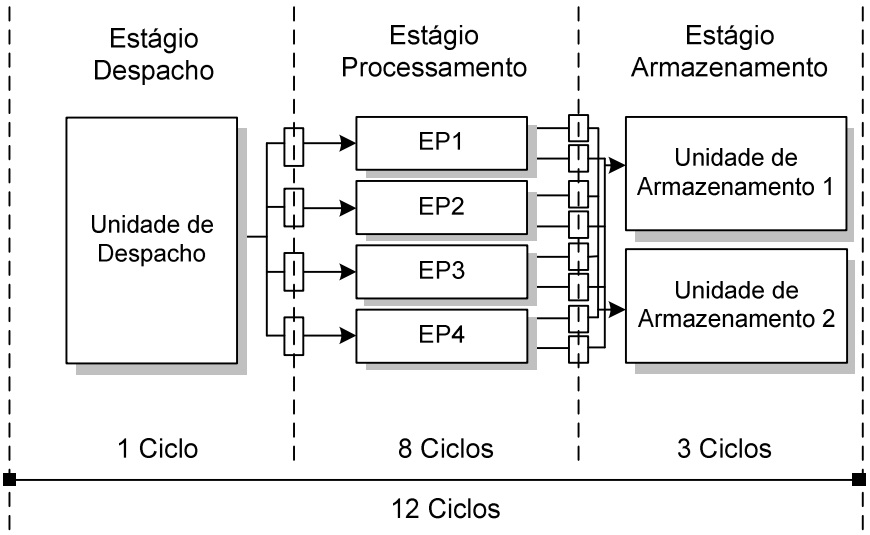


Figura 31: *Pipeline* da arquitetura

O custo de execução paralela pode ser obtido utilizando-se a Equação 9. Destaca-se que as unidades EP_1 a EP_4 operam em paralelo quando não há qualquer dependência entre elas.

$$Execução\ Paralela = UD + \left(EP_1 \middle| EP_2 \middle| EP_3 \middle| EP_4\right) + \left(UA_1 \middle| UA_2\right) \tag{9}$$

3.4 DETALHES DA IMPLEMENTAÇÃO

A arquitetura proposta genérica tem determinadas características que estão listadas na Tabela 2 e que podem sofrer modificações para atender melhor os requisitos da aplicação por meio da reconfigurabilidade. Estes parâmetros podem ser alterados no projeto da arquitetura antes do processamento mediante o envio da nova configuração para o dispositivo.

O número máximo de EPs e a freqüência de operação são decorrentes do *hardware* da arquitetura, mais precisamente da implementação em FPGA. Isto é devido a algumas limitações como a capacidade e estrutura do dispositivo programável. Entretanto, a largura do barramento, o tamanho dos dados, a quantidade de *templates* e a quantidade de instruções são decorrentes das definições da arquitetura.

Tabela 2: Características da arquitetura paralela genérica

Parâmetro	Valor	Unidade
Número máximo de EPs	16	-
Freqüência de Operação	50	MHz
Barramento UD - EP	4x66	*bits*
Bandwidth barramento UD - EP	13,2	*Gbps*
Barramento EP - UA	8x27	*bits*
Bandwidth barramento EP - UA	10,8	*Gbps*
Quantidade de Instruções	16	-
Tamanho dos Dados	16	*bits* (inteiro)
Número (máximo) de *Templates*	1K	-
Memória de Dados (MD)	1K	Palavra (16 *bits*)
Memória de *Templates* (MT)	1K	Palavra (66 *bits*)
Tamanho do *Template*	66	*bits*

3.5 CONJUNTO DE INSTRUÇÕES

O conjunto de instruções desta arquitetura possui um total de 16 instruções básicas, entre aritméticas, lógicas, condicionais, relacionais e miscelâneas. As instruções foram escolhidas de modo a atender uma maior gama de aplicações, como cálculo numérico, criptografia e filtros digitais.

O conjunto de instruções, com o *mnemônico*, descrição, *opcode* e os ciclos de *clock* é apresentado na Tabela 3. O tempo de execução da maioria das instruções é de 11 ciclos de *clock*, as exceções são as instruções de divisão e condicional que consomem 12 ciclos. Outra diferença está presente nas instruções que podem utilizar dois destinos. Neste caso, consomem um ciclo a mais para o destino adicional. Outra diferença ocorre com a instrução *STOP* que consome apenas um ciclo de *clock*, que é o tempo necessário para finalizar a busca de novos *templates*. A instrução *STOP* é executada pela UD, de forma automática, finalizando a busca e o envio de *templates* para os EPs.

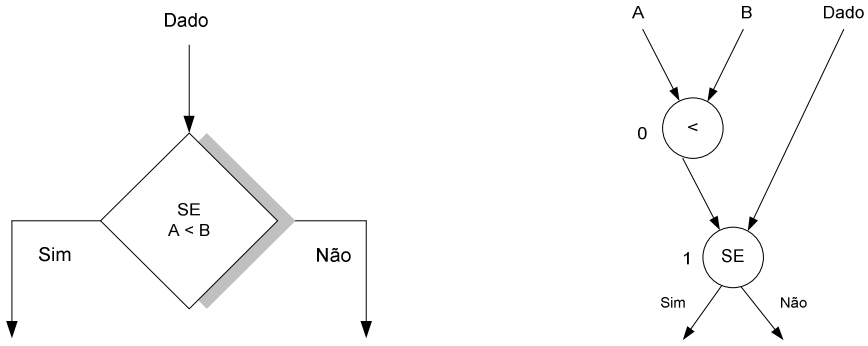
Tabela 3: Conjunto de instruções

Instrução /Mnemônico	Definição	*Opcode*	Ciclos de *Clock*
ADD	Adição	0000	11
SUB	Subtração	0001	11
MUL	Multiplicação	0010	11
DIV	Divisão	0011	12
AND	E Lógico	0100	11
OR	Ou Lógico	0101	11
NOT	Não Lógico	0110	11
XOR	Ou-exclusivo Lógico	0111	11
NOR	Não-Ou Lógico	1000	11
CEQ	Igual a (*compare if equal*)	1001	11
CNE	Diferente de (*compare if not equal*)	1010	11
CGE	Maior ou igual a (*compare if greater or equal to*)	1011	11
CLT	Menor do que (*compare if less than*)	1100	11
SE	Condicional	1101	12
DUP	Duplica	1110	11
STOP	Parada	1111	1

Os testes condicionais são realizados por meio de duas instruções: o relacional (CEQ, CNE, CGE e CLT) e o condicional (SE). Isto é devido à restrição quanto ao número de operandos dos *templates* que, neste caso, é de dois e em função da própria seqüência de execução fluxo de dados, que necessita a passagem de um determinado valor para um dos *templates* específicos, correspondendo às situações "Sim" ou "Não". Em virtude dos testes necessitarem de três valores (dois parâmetros para o teste e o valor a ser passado), esta operação é desmembrada em dois *templates*. Por exemplo, para implementar o Comando SE (A<B), como mostrado na Figura 32a, é necessário utilizar as seguintes instruções:

- CLT A, B ; *A menor que B*
- SE Sim, Não ; *Teste da condição*

Na Figura 32b está representado o grafo de fluxo de dados correspondente ao Comando SE (A<B), que está apresentado como um fluxograma na Figura 32a.



a) b)

Figura 32: a) Fluxograma do Comando SE (A<B); b) Representação em grafo de fluxo de dados

A implementação das duas instruções citadas, correspondendo ao Comando SE (A<B), no formato do *template* para esta arquitetura paralela, é mostrada na Figura 33, em que o primeiro *template* expressa a relação (A<B) e o segundo *template* avalia a condição do teste lógico (SE).

	Operação	Operando 1	M 1	Válido 1	*Stick* 1	Operando 2	M 2	Válido 2	*Stick* 2	Destino 1	Porta 1	T/D 1	Destino 2	Porta 2	T/D 2
0	CLT	A	0	1	0	B	0	1	0	0	0	0	X	X	X
1	SE	?	0	?	0	Dado	0	1	0	Sim	?	0	Não	?	0

Figura 33: Implementação do Comando SE (A<B) em *templates*

Com isto, uma operação condicional consome duas instruções (*templates*), uma a mais do que qualquer outra operação. Isto causa o aumento do grafo de fluxo de dados em um nó por teste condicional. O condicional somente será avaliado quando o seu operando 1 estiver válido.

A quantidade de *templates* não está somente relacionado ao código da aplicação, como também aos dados para processamento, haja visto que o *template* compreende a operação e os operandos. Por exemplo, o código abaixo:

for (i=0; i<1000; i++) { c[i] = a[i] * b[i]; }

corresponde a 1000 *templates* operando em paralelo como segue:

c[0] = a[0] * b[0];

c[1] = a[1] * b[1];

c[2] = a[2] * b[2];

⋮

c[999] = a[999] * b[999];

sem ter que utilizar uma unidade SIMD/vetorial especializada com 1000 ULAs.

3.6 EXEMPLO DE PROGRAMA

Como exemplo de programa no formato de *templates* para a arquitetura desenvolvida, mostrado na Figura 34, foi utilizada a solução numérica da equação diferencial ordinária (Equação 1) sujeita às condições de contorno *y(0)=1* e *y'(0)=0* e descrita no grafo de fluxo de dados da Figura 7.

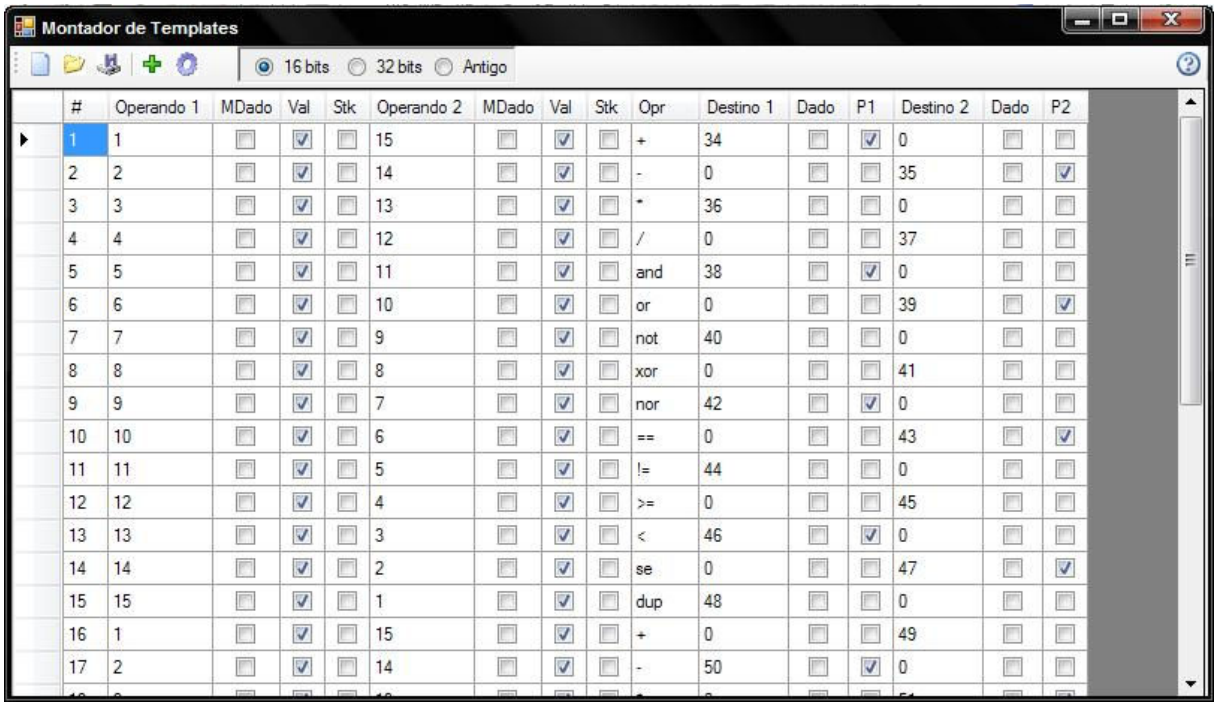
		Operando 1			**Operando 2**			**Destino 1**		**Destino 2**	
Nó	**Operação**	**OP1**	**V1**	**S1**	**OP2**	**V2**	**S2**	**DEST 1**	**P1**	**DEST 2**	**P2**
1	*	**u**	1	0	**dx**	1	1	6	1	0	0
2	*	**2**	1	1	**x**	1	0	6	2	0	0
3	*	**2**	1	1	**y**	1	0	7	2	0	0
4	*	**u**	1	0	**dx**	1	1	8	2	0	0
5	+	**dx**	1	1	**x**	1	0	9	1	10	1
6	*	0	0	0	0	0	0	11	2	0	0
7	*	**dx**	1	1	0	0	0	13	2	0	0
8	+	0	0	0	0	0	0	12	1	0	0
9	<	0	0	0	**a**	1	1	12	2	0	0
10	DUP	0	0	0	0	1	1	2	2	5	2
11	-	0	0	0	0	0	0	13	1	0	0
12	SE	0	0	0	0	0	0	14	1	15	1
13	-	0	0	0	0	0	0	1	1	16	1
14	DUP	0	0	0	0	1	1	8	1	3	2
15	STOP	0	0	0	0	0	0	0	0	0	0
16	DUP	0	0	0	0	1	1	11	1	4	1

Figura 34: Exemplo de programa

O programa é composto por vários *templates*, definidos no conjunto de instruções da arquitetura, e escrito seguindo o fluxo de dados da aplicação.

O tamanho do programa em *bytes* depende do tamanho dos dados, pois o *template* tem que comportar os dados nos seus operandos. No caso do exemplo da Figura 34, o programa consome 132 *bytes*.

A programação da arquitetura ocorre por meio do *software* Montador de *Templates*, mostrado na Figura 35. Este *software* foi desenvolvido especificamente para esta arquitetura, com o objetivo de facilitar a programação da mesma, e possui uma operação intuitiva, pois em cada linha se especifica um *template*. Foram também incorporadas outras facilidades como a modificação do tamanho do dado e uma interface gráfica.



#	Operando 1	MDado	Val	Stk	Operando 2	MDado	Val	Stk	Opr	Destino 1	Dado	P1	Destino 2	Dado	P2
1	1	☐	☑	☐	15	☐	☑	☐	+	34	☐	☑	0	☐	☐
2	2	☐	☑	☐	14	☐	☑	☐	-	0	☐	☐	35	☐	☑
3	3	☐	☑	☐	13	☐	☑	☐	*	36	☐	☐	0	☐	☐
4	4	☐	☑	☐	12	☐	☑	☐	/	0	☐	☐	37	☐	☐
5	5	☐	☑	☐	11	☐	☑	☐	and	38	☐	☑	0	☐	☐
6	6	☐	☑	☐	10	☐	☑	☐	or	0	☐	☐	39	☐	☑
7	7	☐	☑	☐	9	☐	☑	☐	not	40	☐	☐	0	☐	☐
8	8	☐	☑	☐	8	☐	☑	☐	xor	0	☐	☐	41	☐	☐
9	9	☐	☑	☐	7	☐	☑	☐	nor	42	☐	☑	0	☐	☐
10	10	☐	☑	☐	6	☐	☑	☐	==	0	☐	☐	43	☐	☑
11	11	☐	☑	☐	5	☐	☑	☐	!=	44	☐	☐	0	☐	☐
12	12	☐	☑	☐	4	☐	☑	☐	>=	0	☐	☐	45	☐	☐
13	13	☐	☑	☐	3	☐	☑	☐	<	46	☐	☑	0	☐	☐
14	14	☐	☑	☐	2	☐	☑	☐	se	0	☐	☐	47	☐	☑
15	15	☐	☑	☐	1	☐	☑	☐	dup	48	☐	☐	0	☐	☐
16	1	☐	☑	☐	15	☐	☑	☐	+	0	☐	☐	49	☐	☐
17	2	☐	☑	☐	14	☐	☑	☐	-	50	☐	☑	0	☐	☐

Figura 35: *Software* montador de *templates*

O *software* Montador de *Templates* gera um arquivo do tipo **mnt** em que cada linha consiste de um *template* próprio para a arquitetura. Os *templates* são enviados, processados e os resultados lidos da arquitetura após o processamento são exibidos no *software*.

3.7 RECONFIGURAÇÃO DA ARQUITETURA

Esta proposta prevê a readequação da arquitetura apenas em tempo de compilação, em um intervalo de tempo anterior ao recebimento das informações para o processamento, em que a programação é enviada para o dispositivo FPGA para configurá-lo. O modelo de reconfiguração adotado é o semi-estático, no qual a reconfiguração só ocorre após o término da execução da tarefa atual e antes da seguinte. Levando-se em consideração que a arquitetura é voltada, principalmente, ao processamento numérico e que uma reconfiguração só é

necessária após o processamento da aplicação, este modelo atende perfeitamente a estes requerimentos.

A adoção da reconfigurabilidade neste projeto possibilita que características e funcionalidades da arquitetura sejam ajustadas de modo a se conseguir que o tempo de processamento real se aproxime do tempo de processamento ideal, garantindo um melhor desempenho, como discutido na seção 2.3.3.

A arquitetura pode ter um número variável de EPs (4, 8, 12, 16 ou superior), utilizando tamanho de palavras de 8, 16 ou 32 *bits*, ou ainda superiores e os EPs podem ter funcionalidades diferentes. Por exemplo, os EPs podem ser reconfigurados para operar com aritmética de ponto fixo ou com ponto flutuante, dependendo do tipo de problema a ser processado. Para tanto, tais parâmetros devem ser definidos em tempo de compilação para que a configuração seja enviada para o dispositivo programável antes do início do processamento.

Em virtude da arquitetura executar as instruções seguindo o fluxo de dados da aplicação, nenhuma modificação no código é necessária para processar em outra versão da arquitetura, exceto a reformatação do *template*, caso seja necessário readequar os tamanhos.

As modificações da arquitetura, quando necessárias, podem ser com referência ao número de EPs a serem utilizados (neste caso, múltiplos de 4, devido à divisão da MT), tamanho das palavras, funcionalidade/complexidade dos EPs, tamanho das memórias e formato dos *templates*. O projeto teve como ponto central que a arquitetura fosse modular, o que facilita as modificações necessárias para readequação das características funcionais e operacionais. O tamanho das memórias também pode ser modificado bastando proceder os ajustes necessários na UD e na UA.

A arquitetura pode suportar uma série de aplicações, como equações diferenciais, processamento de sinais, criptografia, sendo especificamente voltada para as que possuem ao máximo independência de operações e de dados.

Uma forma de realizar a reconfiguração é o *host* ter várias configurações da arquitetura armazenadas para cada tipo de problema, como se fosse um banco de modelos, e que poderiam ser selecionadas em função da aplicação a ser processada, em que além dos dados e *templates*, também seria enviada a configuração para o dispositivo. Entretanto, isto deveria ser realizado em uma etapa anterior à execução, pois, de forma dinâmica, isto demandaria um gasto complementar de recursos do dispositivo, como elementos de memória e lógicos para o controle adicional.

3.8 VANTAGENS DA ARQUITETURA

Esta arquitetura resolve alguns problemas comuns da computação paralela, como a dificuldade de se escrever programas paralelos. Isto porque é necessário apenas gerar o grafo de fluxo de dados e proceder a conversão para o formato de *templates*, sem ter que se preocupar com a extração do paralelismo. O grafo de fluxo de dados facilita também a portabilidade do programa, permitindo que a aplicação possa ser processada em outras versões da arquitetura.

Outro fato é que não requer conhecimento de vários aspectos da arquitetura para a execução, como o caso do escalonamento das operações, pois isto ocorre automaticamente, na medida em que os *templates* são carregados na memória. Além disto, não é necessário o conhecimento prévio do número de EPs, haja visto que a alocação é automática e praticamente uniforme para todos os EPs da arquitetura, independentemente de uma definição prévia. Isto também favorece, inclusive, a escalabilidade já que a quantidade de EPs pode ser aumentada ajustando-se ao volume de processamento requerido pela aplicação.

Ainda dentro deste pensamento, esta arquitetura rompe com o modelo seqüencial de *von Neumann*, pois não há um contador de programa que aponta para a próxima instrução a ser executada e que é um dos problemas relacionados à programação dos computadores paralelos. Também, a arquitetura utiliza o modelo paralelo denominado de *data driven*, o qual indica as próximas instruções que podem ser executadas com base no fluxo com que os dados estejam disponíveis. Em outras palavras, uma instrução pode habilitar várias outras instruções e não apenas uma como ocorre no modelo *von Neumann*.

A extração do paralelismo da aplicação também ocorre de forma automática durante o processamento, pois explora-se o paralelismo implícito, de modo que não é necessário reescrever o código da aplicação para o formato paralelo para aproveitar o ganho decorrente da execução paralela.

Uma das principais vantagens desta arquitetura está atrelada ao fato de ser genérica e poder ser utilizada para a solução de diversos problemas computacionais, principalmente os que não possuem interdependência de dados.

Outro ponto a ser destacado é que a arquitetura pode ser facilmente adaptada a várias classes de problemas, como criptografia, cálculo numérico, processamento de sinais, bastando para tanto ajustar as características desejadas, por exemplo, o tamanho dos dados.

A arquitetura possui uma característica intrínseca muito importante, que é o fato de ser modular, permitindo que as modificações sejam localizadas sem ser necessário alterar obrigatoriamente toda a estrutura.

Ademais, o fato de ser implementada totalmente em FPGA possibilita que a arquitetura possa ser migrada para outras plataformas, além da Altera (www.altera.com), como a Xilinx (www.xilinx.com) ou mesmo para dispositivos lógicos mais atuais e com maior capacidade.

A possibilidade de formar um *cluster* de arquiteturas paralelas fluxo de dados implica em uma escalabilidade que pode chegar a milhares de EPs utilizando, para isto, vários dispositivos FPGA interconectados e um controle central. Este controle pode ser feito por um processador embarcado, como o NIOS, ou externo, com um microprocessador convencional.

Uma novidade nesta arquitetura é a adoção de um esquema de segmentação da memória de *templates*, com base no princípio da localidade, em que a memória é dividida em módulos que operam com as posições intercaladas. Deste modo, a própria alocação da memória garante um escalonamento das operações para os EPs.

Um programa fluxo de dados se ajusta automaticamente à quantidade de EPs da arquitetura, sem que seja necessário reescrevê-lo. Desta forma, um programa que é executado em uma arquitetura com 8 EPs pode perfeitamente ser executado em outra configuração com 16 EPs ou superior, sem qualquer alteração. Neste caso, o fator limitante para o melhor aproveitamento do potencial de processamento é a quantidade de paralelismo implícito na aplicação.

3.9 LIMITAÇÕES DA ARQUITETURA

Uma das limitações desta implementação, decorrente do dispositivo FPGA, é a quantidade de EPs. A arquitetura proposta precisa de, no mínimo, 4 EPs devido à divisão da MT em 4 módulos e opera com, no máximo, 16 EPs em virtude da capacidade do dispositivo utilizado. Por exemplo, fazendo uma projeção em termos de consumo de elementos lógicos, se fosse utilizado um dispositivo FPGA de alta capacidade como o FPGA EP4SE680, da família Stratix IV da Altera, a arquitetura poderia ter aproximadamente 380 EPs.

Os tamanhos dos dados podem ser de 8, 16 e 32 *bits*. Estes tamanhos foram adotados em função do tipo das aplicações, no caso o cálculo numérico. Contudo, poderiam ser

utilizados valores maiores, por exemplo 64 *bits*, muito embora exigiria mais recursos do que há disponível no dispositivo utilizado (Altera EP2S60F672C3 da família Stratix II).

Foram implementadas operações de ponto fixo para as arquiteturas com dados de 8, 16 e 32 *bits*; e ponto flutuante para a arquitetura configurada para operar com 32 *bits*. A justificativa para se trabalhar com tamanhos variados é que naturalmente haverá um consumo maior de recursos do FPGA em uma arquitetura com 32 *bits* ponto flutuante do que uma arquitetura com 32 *bits* ponto fixo.

Outros limitantes intrínsecos da arquitetura são o tamanho da MT (1K *templates*), que define a quantidade de *templates* que a aplicação pode ter e, também, o tamanho da MD (1Kx16 *bits*), que indica a quantidade de dados a serem processados. Por exemplo, fazendo uma projeção em termos de consumo de elementos de memória, se fosse utilizado um dispositivo FPGA de alta capacidade como o FPGA EP4SE680, da família Stratix IV da Altera, a arquitetura poderia ter aproximadamente 170K posições em ambas as memórias.

Com base no que foi exposto, constata-se que o fator principal dos limites impostos pela arquitetura é o dispositivo FPGA. A estrutura, a quantidade de elementos lógicos e de memória, os recursos internos são cruciais para se definir as características da arquitetura, pois estes recursos são finitos no dispositivo. Uma das maneiras de se contornar este problema é utilizar vários dispositivos interligados, possibilitando uma maior disponibilidade de recursos do que haveria com um único dispositivo.

Algumas modificações na arquitetura afetam diretamente o desempenho, como é o caso da quantidade de EPs, pois quanto maior for este número, maior poderá ser a quantidade de operações realizadas por segundo. A complexidade do EP afeta principalmente a classe de problemas que a arquitetura poderá processar. Como não há registradores, os valores são lidos e armazenados diretamente nos *templates* e, desta forma, não há problema com a alocação dos registradores, como é o caso da renomeação de registradores (STALLINGS, 2006).

Ainda, o acesso à MD é seqüencial e, conseqüentemente, as leituras e escritas são realizadas conforme a ordem de chegada das requisições. Este é um problema que causa lentidão no sistema mas, por outro lado, é uma característica que garante a manutenção da coerência dos dados na MD. Esta memória foi incorporada à arquitetura de modo a possibilitar uma maior versatilidade, como no caso de necessitar de estruturas de dados. Contudo, sabe-se que é um elemento que pode degradar o desempenho geral do sistema, caso o seu uso seja muito intenso. Deste modo, não é recomendado o uso da MD além do estritamente necessário.

CAPÍTULO 4

VALIDAÇÃO E RESULTADOS

Esta arquitetura foi implementada em três versões (denominadas de Versão 1.1, Versão 1.2 e Versão 1.3), como apresentado nas sessões seguintes, e que ajudaram nos melhoramentos da proposta. A Versão 1.1 com 8 EPs compostos por ULA de ponto fixo de 8 *bits*, Versão 1.2 com 16 EPs compostos por ULA de ponto fixo de 16 *bits* e Versão 1.3 com 16 EPs compostos por ULA de ponto fixo de 8/16 *bits*, memória de dados e a adoção do processador NIOS II). A última versão (Versão 1.3) ilustra de forma mais completa a proposta da arquitetura em que todos os recursos estabelecidos foram incorporados.

Em todas as versões, o desenvolvimento ocorreu por meio da linguagem VHDL no ambiente QuartusII^{TM} da Altera (www.altera.com) e a implementação deu-se pela utilização de dispositivos programáveis FPGA, também da Altera. Na Versão 1.1, utilizou-se um dispositivo da família Cyclone, na Versão 1.2 da família Stratix e na Versão 1.3 da família Stratix II.

Neste capítulo são apresentados estudos envolvendo as três versões da arquitetura em que se analisam diversos aspectos, como o gasto de ciclos de *clock*, desempenho, execução de algoritmos e o comparativo com outras arquiteturas.

4.1 TEMPO DE OPERAÇÃO DA ULA

O primeiro estudo consiste em comparar o tempo gasto pela ULA para realizar as operações aritméticas com 32 bits considerando ponto fixo e ponto flutuante, no qual o objetivo é verificar o consumo de ciclos de *clock* em cata tipo de operação.

Em virtude do consumo excessivo de recursos do dispositivo FPGA, limitou-se o número de EPs a 16 e o tamanho dos dados a 16 *bits*. Contudo, realizaram-se alguns testes com implementações de ULA com 32 *bits* para se avaliar o desempenho (em ciclos de *clock*) nas configurações de ponto fixo e ponto flutuante.

Observou-se que há o consumo de um ciclo de *clock* para realizar as operações aritméticas, exceto a divisão, com precisão de 32 *bits* (ponto fixo) e de, no mínimo, três ciclos para 32 *bits* (ponto flutuante). A exceção é a operação de divisão (ponto fixo) que pode

consumir até quatro ciclos de *clock* dependendo dos operandos, como apresentado na Tabela 4.

Tabela 4: Tempo (em ciclos de *clock*) gasto pela ULA nas operações com 32 *bits*

Operação com 32 *bits*	Ponto Fixo (ciclos de *clock*)	Ponto Flutuante (ciclos de *clock*)
Adição	1	3 – 16
Subtração	1	3 – 16
Multiplicação	1	3 – 18
Divisão	1 – 4	3 – 18

Desta forma, implementações com 32 *bits* podem ser viáveis desde que se equacione apropriadamente os recursos do FPGA com as características desejadas na arquitetura.

4.2 TEMPO DE EXECUÇÃO DAS INSTRUÇÕES

Este estudo é decorrente da comparação dos tempos de execução das instruções da arquitetura, considerando operações de ponto fixo com 8, 16 e 32 bits, com o objetivo de averiguar as diferenças em ciclos de *clock* entre as instruções.

Na Tabela 5, têm-se os tempos de execução (em ciclos de *clock*) gastos em cada instrução/operação com ponto fixo, considerando 8, 16 e 32 *bits*.

Tabela 5: Tempo (em ciclos de *clock*) gasto em cada instrução com ponto fixo

Tipo de Instrução	8 *bits*	16 *bits*	32 *bits*
Adição	11	11	11
Subtração	11	11	11
Multiplicação	11	11	11
Divisão	12	12	14
Lógicas (E, Ou, Não, Ou-Exclusivo e Não-Ou)	11	11	11
Relacionais (>=, <, ==, !=)	11	11	11
Condicional (*SE*)	12	12	12
Duplica	11	11	11
Stop	1	1	1

Observando-se os valores, percebe-se que algumas instruções consomem mais ciclos de *clock* do que outras. Isto acontece em decorrência da complexidade da instrução a ser realizada, o que pode ser constatado nas instruções de divisão e condicional.

As mesmas instruções de ponto fixo com 8, 16 e 32 *bits* consomem igual quantidade de ciclos de *clock*, com exceção da instrução de divisão com 32 *bits*, que pode consumir até 14 ciclos dependendo dos operandos.

4.3 DIAGRAMA DE TEMPORIZAÇÃO

Neste experimento é realizado um estudo teórico considerando a execução em um módulo FD (Fluxo de Dados), composto por uma UD (Unidade de Despacho), 4 EPs (Elementos Processadores) e 2 UA (Unidades de Armazenamento), com o objetivo de verificar o desempenho que pode ser alcançado nos casos de teste com a arquitetura nas configurações com e sem *buffer*.

Os diagramas de temporização apresentados abaixo foram obtidos levando-se em consideração as seguintes características:

- As operações são de ponto fixo com 16 *bits*.
- O gasto de tempo de uma execução completa é de, no máximo, 12 ciclos de *clock*.
- A Unidade de Despacho (UD) gasta 1 ciclo de *clock*.
- Os Elementos Processadores (EPs) gastam, no máximo, 8 ciclos de *clock*.
- A Unidade de Armazenamento (UA) gasta 3 ciclos de *clock*.
- Considera-se apenas um pedaço da Memória de *Templates* (MT).
- Os *templates* são independentes.
- Os EPs somente estão disponíveis após a UA ter armazenado os dados na MT.

Nos três casos simulados, visualiza-se o processamento de 12 *templates* independentes, na estrutura composta por uma Unidade de Despacho (UD), quatro Elementos Processadores (EP1 a EP4) e duas Unidades de Armazenamento (UA1 e UA2) em uma organização *pipeline*, como mostrado na Figura 31. O tempo está expresso em ciclos de *clock* e o objetivo é visualizar temporalmente o processamento levando-se em consideração três situações hipotéticas, sendo uma para o Pior Caso, outra para o Caso Intermediário e a terceira para o Melhor Caso.

4.3.1 Situações de Estudo (Casos)

Os testes neste estudo foram realizados utilizando-se três situações: Pior Caso em que há dependência entre os *templates*, Caso Intermediário com 50% de dependência, e Melhor Caso com nenhuma dependência.

- **Pior Caso**

Nesta situação, considerou-se que todos os *templates* atualizarão os Operandos 1 do mesmo pedaço da memória. Com isto, a unidade de armazenamento do segundo operando (UA2) estaria ociosa enquanto a UA1 seria responsável por todo o trabalho de atualização dos *templates*.

- **Caso Intermediário**

Nesta situação intermediária, considerou-se que os *templates* 2, 6 e 10 atualizam somente os Operandos 2, e os demais *templates*, somente os Operandos 1 da MT. Desta forma, tem-se um caso em que há um paralelismo entre as unidades de armazenamento (UA1 e UA2), mas ainda manteve-se a serialização no armazenamento em três seqüências de *templates* (1-3-4, 5-7-8 e 9-11-12).

- **Melhor Caso**

Nesta última situação, Melhor Caso, tem-se uma seqüência de *templates* que utilizam alternadamente as unidades de armazenamento (UA1 e UA2). Desta forma, enquanto uma das UAs está em operação, a outra está liberada para atender a requisição de atualização dos dados de outro EP (*template*).

4.3.2 Arquitetura Sem *Buffer*

Na configuração da arquitetura sem *buffer*, os testes são realizados considerando-se que as unidades da arquitetura não possuem qualquer tipo de *buffer*. Não havendo armazenamentos temporários entre as unidades, isto implica que elas ficam ocupadas até todo o ciclo estar completo, limitando o paralelismo das operações.

- **Teste Sem *Buffer* - Pior Caso**

Na Figura 36, apresenta-se o teste com 12 *templates* para o Pior Caso. Observa-se que o tempo gasto para o processamento do primeiro *template* foi de 12 ciclos de *clock* e do 12º *template* foi de 45 ciclos de *clock*, o que dá uma média de 3,75 ciclos por *template*. Se for considerado o gasto de ciclos de *clock* para processar serialmente os 12 *templates* (144 ciclos), a economia é de 99 ciclos de *clock*, o que representa uma redução de 3,2 vezes.

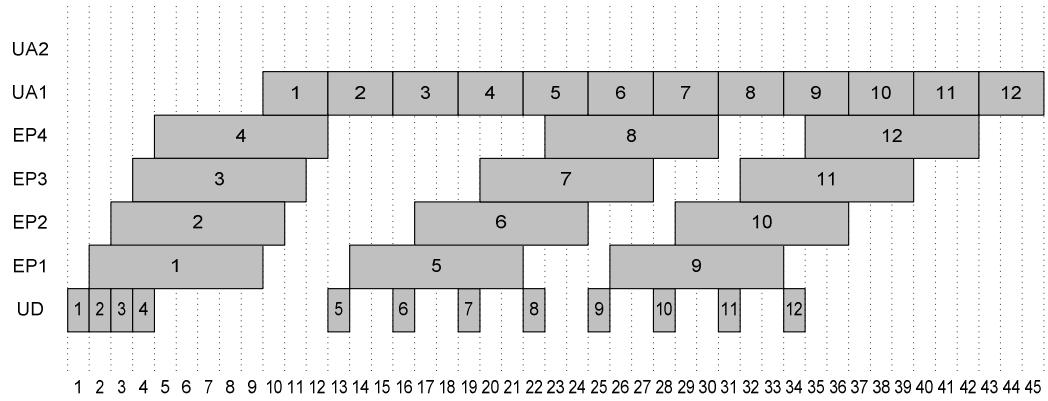


Figura 36: Diagrama de temporização (sem *buffer*) - pior caso

- **Teste Sem *Buffer* - Caso Intermediário**

Na Figura 37, percebe-se que há seis ciclos de *clock* em que as unidades de armazenamento (UAs) estão ociosas (ciclos 19-20-21 e 31-32-33). Isto já sugere que há uma necessidade de *buffers* para o armazenamento do processamento, possibilitando uma liberação dos EPs antes que as UAs finalizem o armazenamento. Este procedimento possibilita que os EPs possam iniciar o processamento de um novo *template* antes de o *template* anterior ter sido completamente finalizado, ou seja, antes de o resultado ter sido armazenado.

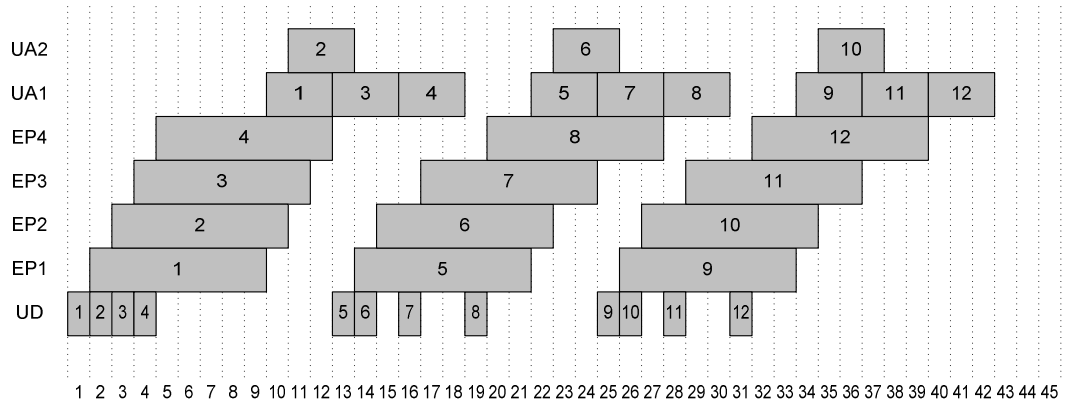


Figura 37: Diagrama de temporização (sem *buffer*) - caso intermediário

Neste teste do Caso Intermediário, obteve-se uma redução no tempo de processamento ainda maior do que no caso anterior (Pior Caso), pois agora o tempo total foi de 42 ciclos de *clock*, três a menos do que o anterior, para os mesmos 12 *templates*. Agora, o tempo médio por *template* é de 3,50 ciclos de *clock*. Contudo, nota-se que o tempo gasto para o primeiro *template* permanece em 12 ciclos de *clock*.

- **Teste Sem *Buffer* - Melhor Caso**

Observa-se na Figura 38 que o tempo total de processamento (dos 12 *templates*) foi ainda menor do que o caso anterior (40 ciclos de *clock*), o que resulta em um tempo médio de 3,33 ciclos de *clock*.

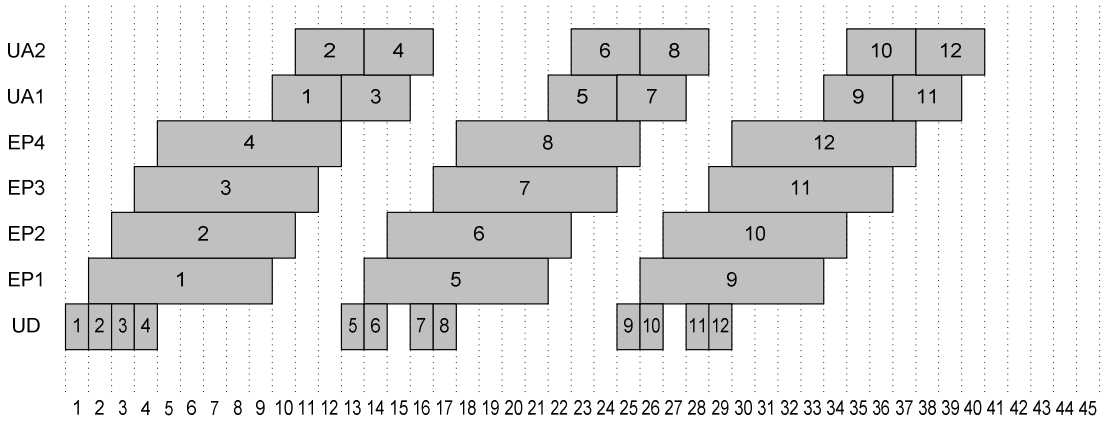


Figura 38: Diagrama de temporização (sem *buffer*) - melhor caso

Contudo, observa-se que neste caso o período de tempo em que as unidades UAs estão ociosas é ainda maior (12 ciclos de *clock*) e o aproveitamento deste tempo resultaria em uma redução ainda mais significativa no gasto de ciclos de *clock*. Entretanto, para isto se faz necessário que haja *buffers* entre os EPs e as UAs, de modo que os EPs sejam liberados para novo processamento enquanto as UAs ainda estão armazenando os resultados processados.

Olhando atentamente para a Figura 38, observa-se que na melhor situação se conseguiria uma redução de quatro ciclos de *clock*, pois não haveria ciclos de ociosidade entre o processamento de dois *templates* pelo EP. Entretanto, ainda resultaria em um período de ociosidade de pelo menos dois ciclos de *clock* nas UAs.

- **Considerações**

Nos três casos apresentados, constata-se a necessidade da utilização de *buffers* entre a UD e os EPs, em decorrência da alta taxa de transferência (*throughput*) da UD e também pelo fato de os EPs serem mais lentos e, com isto, estarem indisponíveis para nova operação.

Outra melhoria é a utilização de *buffer* entre os EPs e as UAs. Nesta situação, a necessidade do *buffer* é decorrente do gasto de ciclos para este processo pelas UAs, pela sobrecarga de requisições a essas unidades (UA1 e UA2) e a conseqüente serialização dos atendimentos dos EPs.

Na Tabela 6 está apresentado um resumo dos dados das simulações dos três casos para a execução dos 12 *templates*. Os valores foram obtidos considerando-se a execução seqüencial de 12 *templates*, consumindo 144 ciclos de *clock*.

Tabela 6: Resultado das simulações para a arquitetura sem *buffer*

Característica	Pior Caso	Caso Intermediário	Melhor Caso
Tempo de Execução (Ciclos de *Clock*)	45	42	40
Tempo médio por *template* (Ciclos de *Clock*)	3,75	3,50	3,33
Economia (Ciclos de *Clock*)	99	102	104
Economia (%)	68,75	70,83	72,22
Speedup	3,2	3,4	3,6

Com base no que foi apresentado, constata-se que para 12 *templates* o ganho em termos de tempo de execução (ciclos de *clock*) é de no mínimo 68,75% (Pior Caso), podendo chegar a 72,22% (Melhor Caso), o que resulta em um *Speedup* de 3,2 e 3,6, respectivamente.

Neste caso, define-se ganho como sendo a porcentagem de redução do tempo de execução de cada um dos casos de estudo em relação ao tempo de execução seqüencial.

4.3.3 Arquitetura Com *Buffer*

Nesta situação, os testes são realizados considerando-se que as unidades da arquitetura possuem um *buffer* FIFO. Considera-se que o armazenamento no *buffer* de saída gasta um ciclo de *clock*. Desta forma, as unidades ficam livres para nova operação um ciclo de *clock* após terem finalizado a operação anterior. Este recurso possibilita um maior paralelismo na

execução, em decorrência de as unidades estarem liberadas antes que o processamento anterior esteja finalizado.

- **Teste Com *Buffer* - Pior Caso**

Na Figura 39, observa-se que a UD envia um *template* por ciclo de *clock* e são armazenadas nos *buffers* dos EPs, liberando a UD para novos *templates*. Na operação de armazenamento pela UA1, já que é a única que tem requisições, ocorre a sobreposição de três solicitações e os valores são armazenados nos *buffers* de saída dos EPs. Contudo, para efeito de estudo, assumiu-se que este processo de armazenamento e envio para a UA1 gasta um ciclo de *clock*, garantindo as saídas dos EPs válidas durante um período de tempo em que os resultados são armazenados nos *buffers*. Observa-se, ainda, que o tempo total de execução dos 12 *templates* foi de 31 ciclos de *clock*, o que resulta em uma média de 2,58 ciclos por *template*. Se for considerado o gasto de ciclos de *clock* para processar serialmente os 12 *templates* (144 ciclos), a economia é de 113 ciclos de *clock*, o que representa uma redução de 4,7 vezes. Analisando o teste de Pior Caso, chega-se à conclusão de que para este caso a arquitetura necessita de um *buffer* de entrada com 3 posições e um *buffer* de saída com 4 posições.

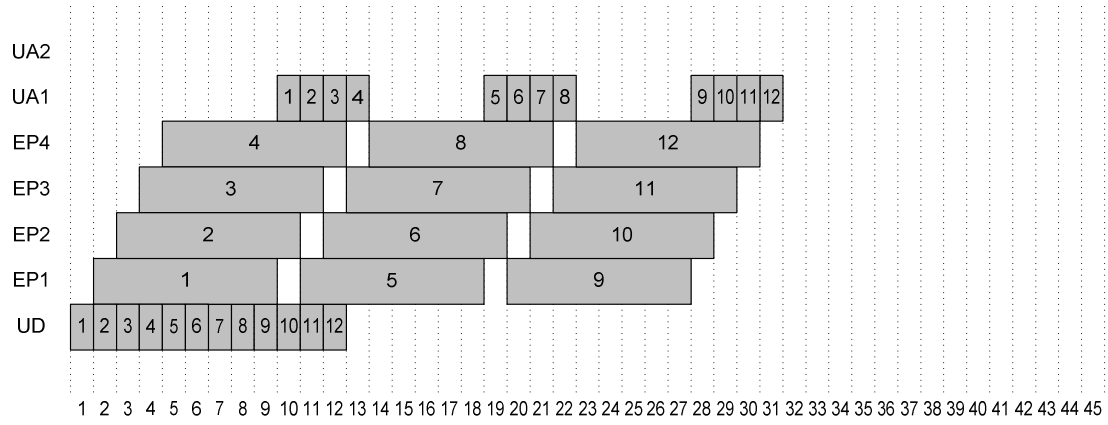


Figura 39: Diagrama de temporização (com *buffer*) - pior caso

- **Teste Com *Buffer* - Caso Intermediário**

O teste de Caso Intermediário apresentado na Figura 40 mostra que o tempo total de execução para os 12 *templates* foi, como na situação anterior, de 31 ciclos de *clock*. Nesta

situação não foi observado qualquer ganho em ciclos de *clock*, apenas constata-se que o tamanho do *buffer* de saída pode ser menor que o anterior. Com este teste, conclui-se que a arquitetura necessita de um *buffer* de entrada com 3 posições e um *buffer* de saída também com 3 posições.

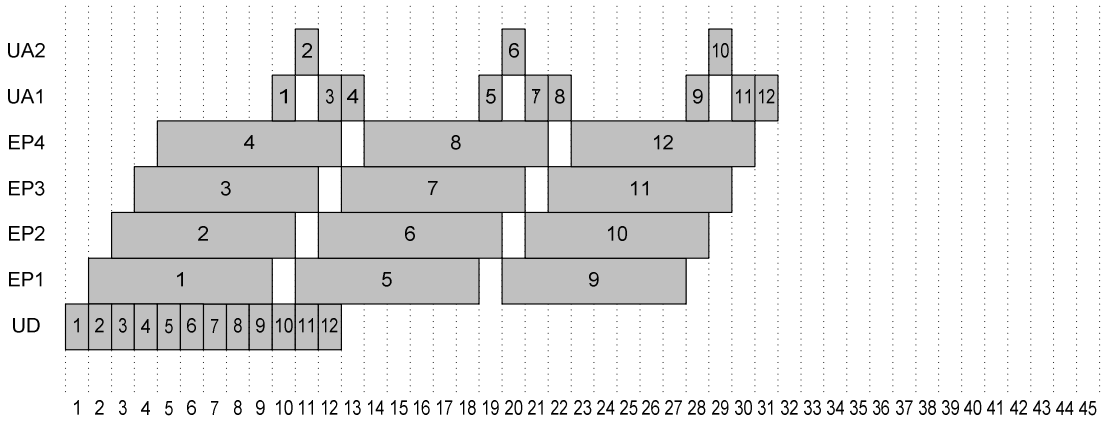


Figura 40: Diagrama de temporização (com *buffer*) - caso intermediário

- **Teste Com *Buffer* - Melhor Caso**

Da mesma forma que nos casos anteriores, no teste do Melhor Caso, mostrado na Figura 41, o tempo de execução é de 31 ciclos de *clock*, também não apresentando ganhos em relação aos outros casos. O que se pode verificar é que, pelas condições do teste, há a necessidade de uma menor quantidade de *buffers* de saída.

Neste último teste, chega-se à conclusão de que a arquitetura necessita de um *buffer* de entrada com 3 posições e um *buffer* de saída com 2 posições.

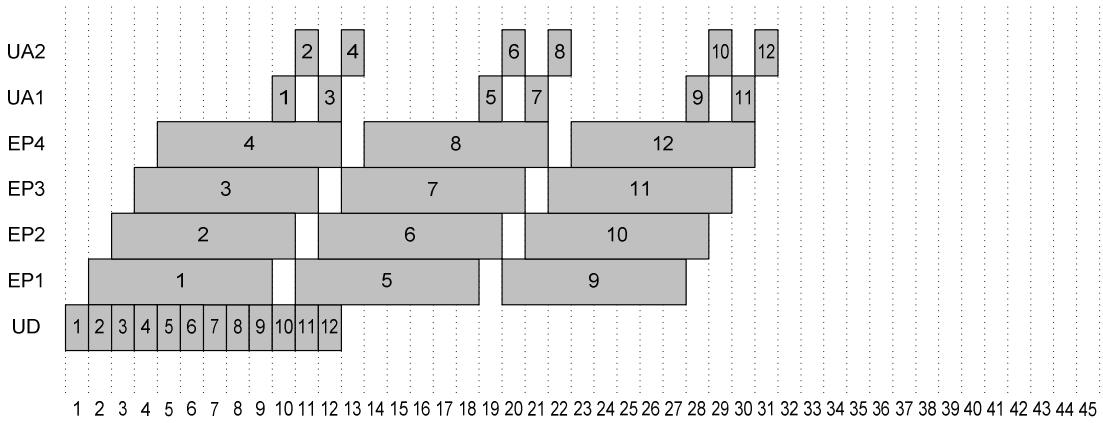


Figura 41: Diagrama de temporização (com *buffer*) - melhor caso

- **Considerações**

Nos três casos apresentados, observa-se que o tempo total de execução dos 12 *templates* foi de 31 ciclos de *clock*, o que resulta em uma média de 2,58 ciclos por *template*. Se for considerado o gasto de ciclos de *clock* para processar serialmente os 12 *templates* (144 ciclos), a economia é de 113 ciclos de *clock*, o que representa um ganho de 78,47% em termos de tempo de execução, indicando um *Speedup* de 4,7. A Tabela 7 contém os resultados das simulações para os três casos na arquitetura com *buffer*.

Com os testes realizados, constata-se que a arquitetura deve ter o *buffer* de entrada com 3 posições, em todas as situações, e o de saída com 4 posições para atender o Pior Caso.

Tabela 7: Resultado das simulações para a arquitetura com *buffer*

Característica	Pior Caso	Caso Intermediário	Melhor Caso
Tempo de Execução (Ciclos de *Clock*)	31	31	31
Tempo médio por *template* (Ciclos de *Clock*)	2,58	2,58	2,58
Economia (Ciclos de *Clock*)	113	113	113
Economia (%)	78,47	78,47	78,47
Speedup	4,7	4,7	4,7

4.3.4 Gasto de Ciclos de *Clock*

O estudo do gasto de ciclos de *clock*, considerando-se o Melhor Caso nas situações da arquitetura sem e com *buffer*, é mostrado na Figura 42, realizando uma projeção para 1.000.000 *templates*. As fórmulas para a estimativa do consumo de ciclos de *clock* em que N representa a quantidade de *templates* são:

- Sem *buffer*: N + 15 * INT((N - 1)/4) + ARRED((N - 1)/4) + 12;
- Com *buffer*: N + 5 * INT((N - 1)/4) + 9.

Obs.: A função INT retorna a parte inteira de um número real e a função ARRED faz o arredondamento de um número real para o inteiro mais próximo.

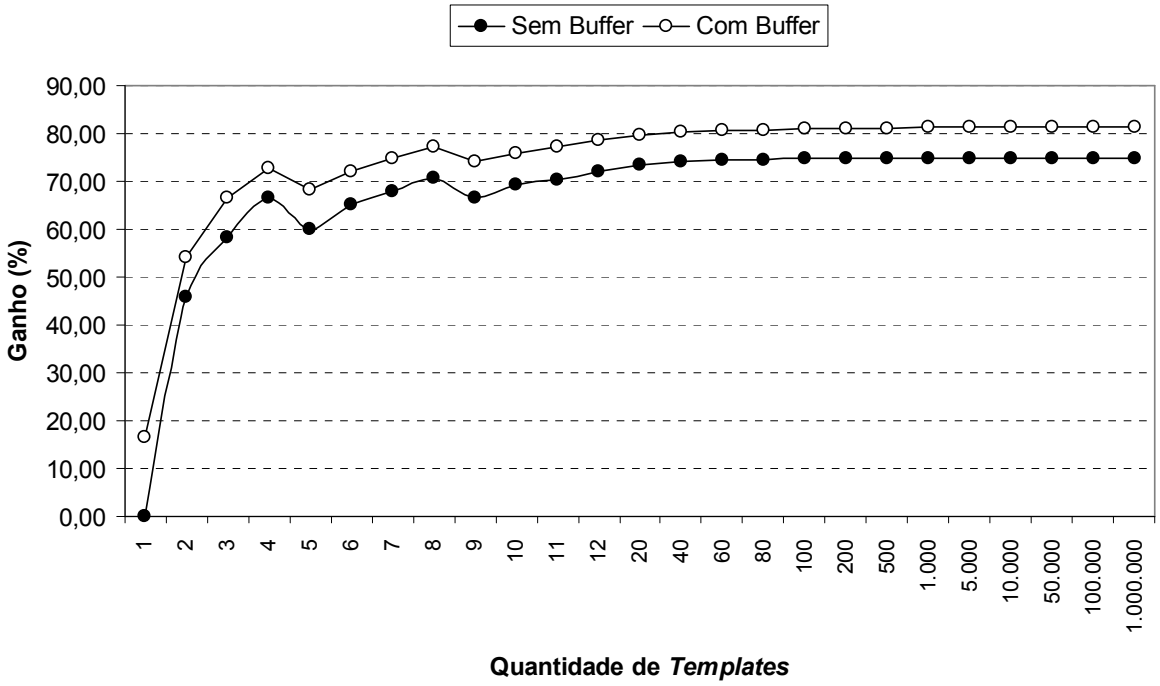


Figura 42: Comparativo das curvas de ganho de ciclos de *clock*

A execução paralela com a arquitetura sem *buffer*, em comparação com a execução seqüencial, obteve um ganho de aproximadamente 75% em ciclos de *clock* e um *Speedup* de 4,0. Este mesmo estudo, considerando-se a situação em que há *buffer* na arquitetura, eleva o ganho em ciclos de *clock* para 81,25%, representando um aumento de 8,33% em relação à arquitetura sem *buffer*. Com este teste, obteve-se o *Speedup* igual a 5,3, e isto demonstra um aumento de 32,50% em relação ao *Speedup* da arquitetura sem *buffer*. Considerando-se o gasto de ciclos de *clock* nas situações com e sem *buffer*, no teste do Melhor Caso, observa-se que houve um ganho de 27,70% na execução com 12 *templates* independentes.

O *Speedup* (S) foi obtido pela razão entre o tempo de execução seqüencial e o produto do tempo de execução seqüencial com a diferença entre unitário e o ganho, como descrito na Equação 10.

$$S = \frac{Tempo\ Sequencial}{Tempo\ Sequencial * (1 - ganho)} \tag{10}$$

Os gráficos da Figura 42 foram obtidos mediante extrapolação da função de ganho para a execução de até 1.000.000 *templates*. Inicialmente, determinou-se o ganho em termos da redução do tempo de execução para os 12 primeiros *templates* e para as outras quantidades de *templates* utilizou-se a extrapolação.

4.3.5 Comparação entre as Arquiteturas Com e Sem *Buffer*

Com base nos resultados dos testes da arquitetura sem *buffer*, mostrados na Tabela 6, obtém-se que a relação entre os casos Pior/Melhor é de 12,50%, e na relação Pior/Intermediário o valor é de 7,14%. Desta forma, as restrições de armazenamento são responsáveis por uma diferença de, no máximo, 12,50% no desempenho em redução dos ciclos de *clock*.

Nos mesmos testes considerando a arquitetura com *buffer*, mostrados na Tabela 7, constata-se que todos os casos apresentaram um consumo de 31 ciclos de *clock*, ou seja, nos testes não foram obtidas diferenças nos valores. Isto indica que as restrições de armazenamento não afetam o desempenho da arquitetura com *buffer*.

Levando-se em conta os resultados dos casos, considerando-se a arquitetura sem e com *buffer*, tem-se que: a) na situação do Pior Caso, o ganho em termos da redução do tempo de execução foi de 45,16%; b) na situação do Caso Intermediário, foi de 35,48%; e c) na situação do Melhor Caso, o ganho foi de 29,03%. Desta forma, o menor ganho (redução do tempo de execução) foi obtido na situação do Melhor Caso, que foi 29,03%, e isto representa um *Speedup* relativo de 1,29. Este *Speedup* relativo é obtido pela razão entre o gasto de ciclos de *clock* do Melhor Caso Sem *Buffer* e o gasto de ciclos de *clock* do Melhor Caso Com *Buffer*.

4.4 ARQUITETURA - VERSÃO 1.1

Os testes nesta versão da arquitetura (Versão 1.1) envolveram a execução de 10 instruções independentes nas configurações com até 8 EPs, em que o objetivo é de verificar a viabilidade da proposta.

Esta Versão 1.1, mostrada na Figura 43 e apresentada em FERLIN *et al* (2005), foi implementada em um dispositivo programável FPGA EP1C20F400C8 da família Cyclone da Altera, operando em uma freqüência de 40MHz e com um ciclo de *clock* de 25*n*s. O tempo de ciclo de execução de um *template* na arquitetura é de aproximadamente 200*n*s. Com isto, esta arquitetura pode executar aproximadamente 5 milhões de operações aritméticas de ponto fixo com palavras de 8 *bits*, como adição ou subtração, em um segundo utilizando apenas um EP.

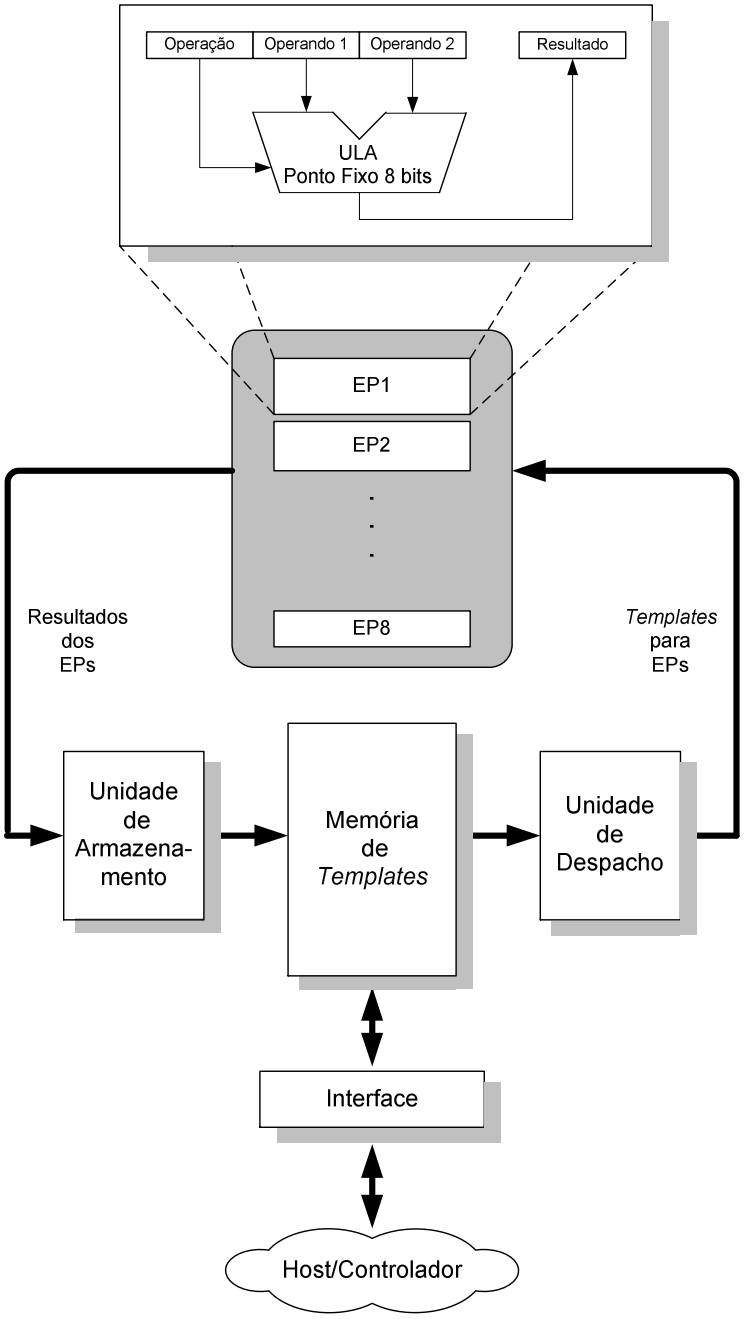


Figura 43: Visão geral da arquitetura – versão 1.1

Esta implementação possui um conjunto de características que estão apresentadas na Tabela 8. Nesta versão, a arquitetura possui 8 EPs, consistindo de ULA de ponto fixo de 8 *bits*. A parte de controle possui quatro unidades: armazenamento, memória de *templates*, despacho e interface. As unidades de despacho e armazenamento operam seqüencialmente. A arquitetura suporta aplicações com até 128 *templates*. Ainda, as operações podem ser de 8 tipos diferentes de instruções, nomeadamente *adição, subtração, se, <, >=, ==,!=, stop*.

Tabela 8: Características da implementação – versão 1.1

Característica	Valor	Unidade
Número máximo de EPs	8	-
Freqüência de Operação	40	MHz
Complexidade do EP	ULA ponto fixo 8 *bits*	-
Despacho	Seqüencial	-
Armazenamento	Seqüencial	-
Quantidade de Instruções	8	-
Tempo de *Clock*	25	*n*s
Memória de *Templates*	128	*Templates*

Os testes foram realizados utilizando-se 10 *templates* independentes, com dados de 8 *bits*, que pudessem ser executados simultaneamente, e gradualmente adicionaram-se EPs até o limite de 8, que é o número máximo de EPs suportados por esta versão. Com estes testes têm por finalidade observar as restrições/limitações de desempenho intrínsecas à arquitetura.

Constatou-se que o tempo total de execução reduziu-se na medida em que se adicionavam mais EPs na arquitetura paralela. Isto é mostrado no gráfico da Figura 44.

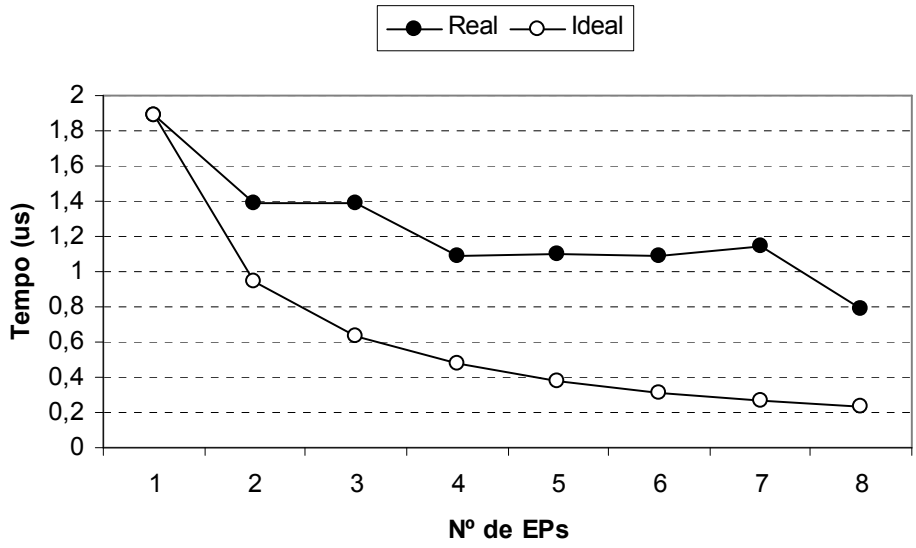


Figura 44: Tempo de execução com 10 *templates* (operações) – versão 1.1

Contudo, ressalta-se que nesta versão, em virtude da estrutura de acesso à memória de *template* não ser *dual-port*, tem-se um período no qual aparentemente não se está conseguindo reduzir o tempo de processamento, apesar do número crescente de EPs, mais especificamente no caso de 4 a 7 EPs. Este é um dos pontos que devem ser remodelados para se reduzir o

tempo de acesso à memória e conseguir um maior ganho em tempo de processamento. Apesar disto, percebe-se que com a inclusão de um 8º EP, a arquitetura ainda obteve uma redução do tempo de processamento.

Se for comparada a curva do tempo obtido com a curva do tempo ideal, como apresentado na Figura 44, percebe-se que as curvas divergem bastante. Isto decorre de limitações da arquitetura, como o fato de o despacho e o armazenamento serem seqüenciais.

Na Figura 45, tem-se um gráfico apresentando a evolução do *Speedup* com o incremento do número de EPs na arquitetura paralela. Observando-se este gráfico, rapidamente constata-se que o *Speedup* não atingiu o valor desejado, pois esperava-se, em função dos testes realizados, que o valor de *Speedup* fosse próximo ao número de EPs. Na configuração com 8 EPs, a eficiência foi de aproximadamente 30%.

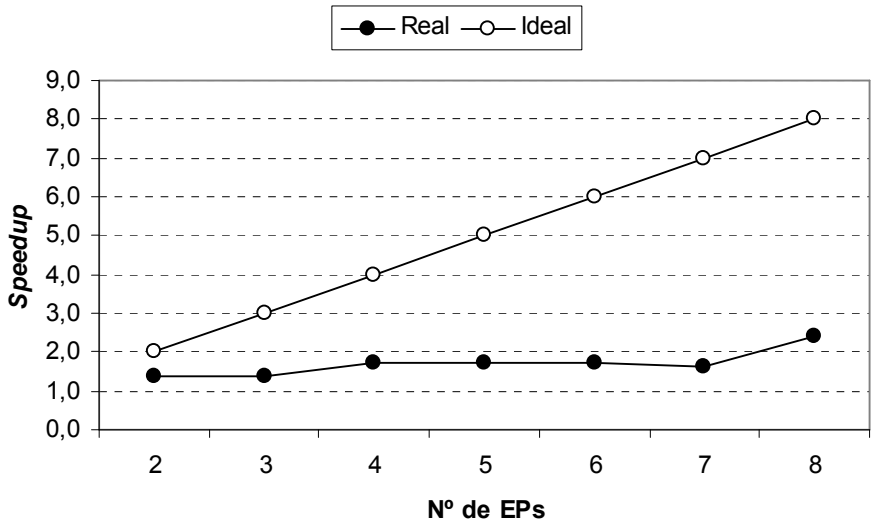


Figura 45: Gráfico do *speedup* - versão 1.1

Esta implementação no dispositivo programável FPGA EP1C20F400C8 da família Cyclone da Altera consumiu 2.128 elementos lógicos e 6.144 bits de memória, correspondendo a 10% e 2% dos recursos disponíveis, respectivamente. Cada EP consome 240 elementos lógicos, representando 1% deste recurso do dispositivo programável.

4.5 ARQUITETURA - VERSÃO 1.2

Nesta versão da arquitetura (Versão 1.2) foram executados o algoritmo com 32 instruções independentes e o algoritmo FIR, com a finalidade de provar o conceito e também comparar o desempenho com outras implementações.

Esta versão é uma evolução da Versão 1.1. A implementação opera em uma freqüência de 50MHz e com um ciclo de *clock* de 20*n*s. O tempo de ciclo de execução de um *template* na arquitetura é de aproximadamente 100*n*s. Com isto, a implementação pode executar aproximadamente 10 milhões de operações por segundo utilizando apenas um EP.

Esta versão apresenta um conjunto de melhoramentos, apresentados na Tabela 9. A implementação possui na parte de controle cinco unidades: armazenamento, memória de *templates*, despacho, tabela de EPs e interface, como apresentado na Figura 46. Além disso, a arquitetura possui um total de 16 EPs, sendo que cada EP é composto por uma ULA de 8 *bits* operando com ponto fixo. A memória suporta aplicações com até 256 *templates.* As operações podem ser de 16 tipos (*adição, subtração, multiplicação, divisão, se, <, >=, ==,!=, E lógico, Ou lógico, Ou-Exclusivo lógico, Negação lógico, Não-Ou lógico, duplicação e stop*).

Tabela 9: Características da implementação – versão 1.2

Característica	Valor	Unidade
Número máximo de EPs	16	-
Freqüência de Operação	50	MHz
Complexidade do EP	ULA ponto fixo de 16 *bits*	-
Despacho	Bufferizado	-
Armazenamento	Bufferizado	-
Quantidade de Instruções	16	-
Tempo de *Clock*	20	*n*s
Memória de *Templates*	256	*Templates*

A memória de *templates* e a tabela de EPs são de duplo acesso, possibilitando acessos simultâneos. As unidades de armazenamento e despacho operam paralelamente, aumentando o desempenho, pois reduziu-se o tempo de acesso à memória de *templates*.

Uma importante modificação incorporada nesta nova arquitetura é a inclusão de *buffers* nas entradas e saída dos EPs. Isto possibilita uma redução no tempo de processamento, pois tanto a unidade de despacho quanto a unidade de armazenamento deixam de acessar diretamente o EP para se comunicar por meio dos *buffers,* cada um com 8 posições.

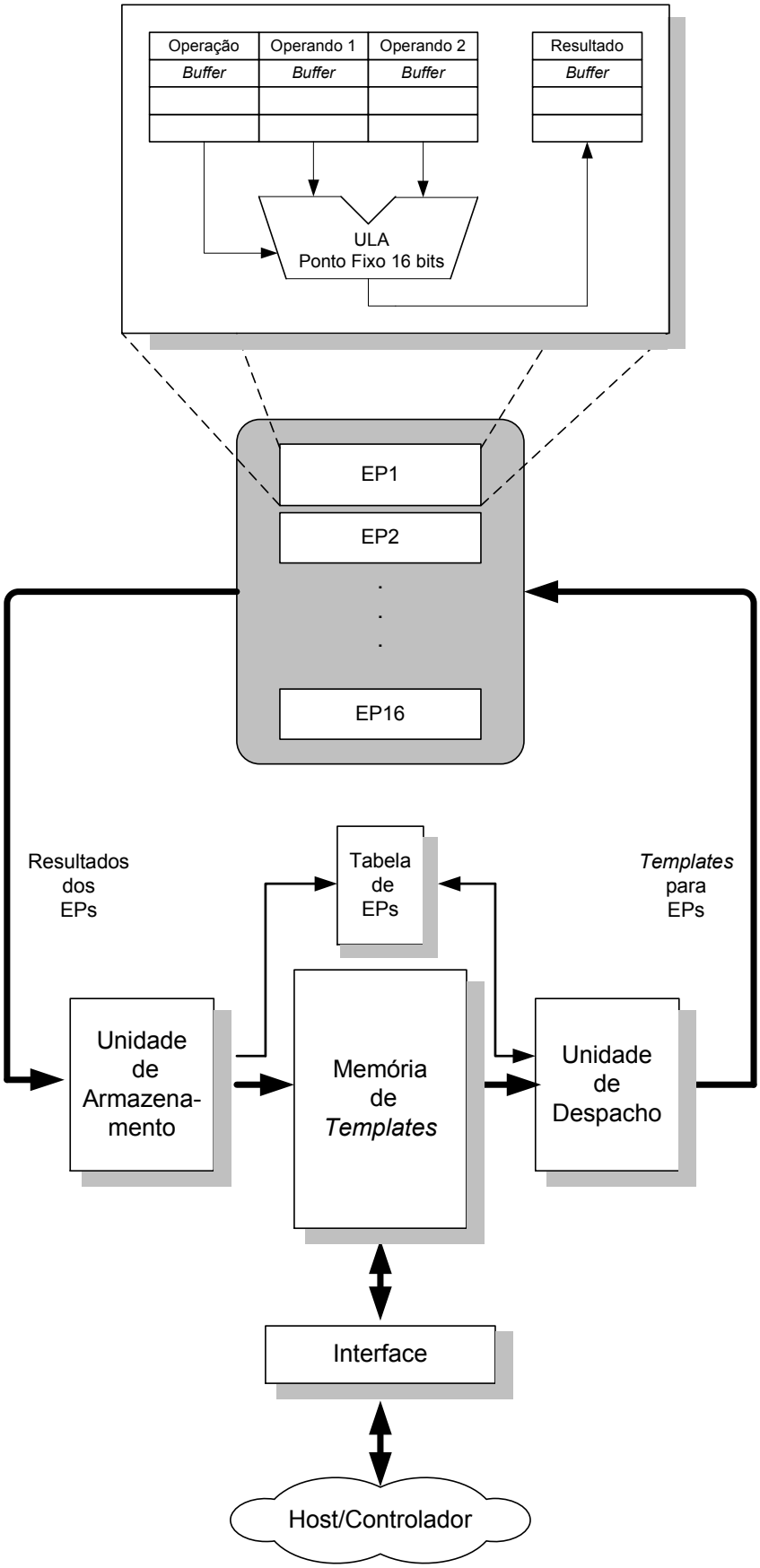


Figura 46: Visão geral da arquitetura – versão 1.2

4.5.1 Teste – Algoritmo Independente (32 Operações)

Os testes com 1, 4, 8, 12 e 16 EPs foram realizados utilizando-se 32 *templates* independentes e uma aritmética de ponto fixo com 16 *bits* de resolução. Gradualmente, adicionaram-se EPs na arquitetura, até o limite máximo de 16. Constatou-se que o tempo total de execução sofreu uma redução na medida em que se adicionavam EPs na arquitetura paralela, como mostrado na Figura 47.

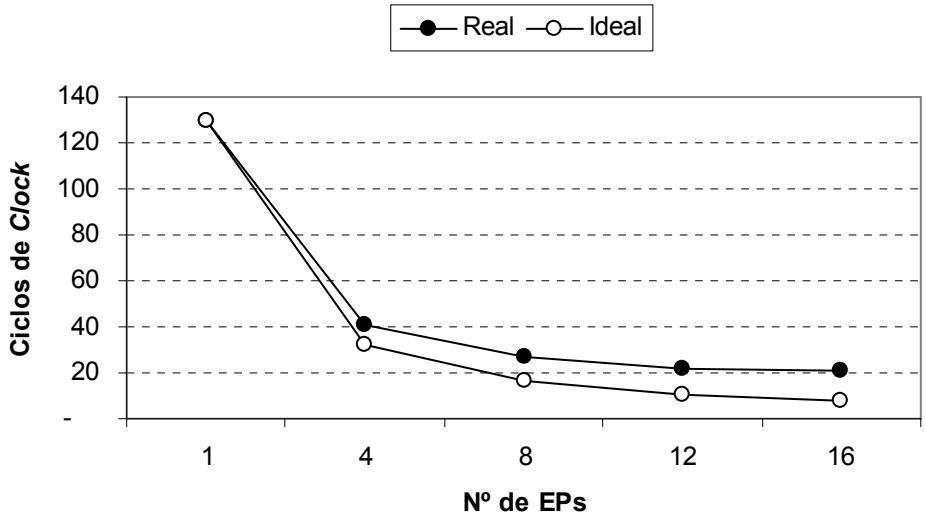


Figura 47: Tempo de execução (ciclos de *clock*) – algoritmo independente – versão 1.2

O tempo de processamento cai a uma taxa variável em função do número de EPs. Os testes revelaram que com 16 EPs o tempo de processamento ficou 39% acima do tempo ideal estimado. Nos testes realizados, o limitante referente ao *software* foi eliminado, já que as instruções são independentes. Com isto, pode-se concluir que a redução do desempenho é decorrente da arquitetura, mais precisamente pelo despacho dos *templates* não ser paralelo.

Na Figura 48 é apresentado um gráfico da evolução do *Speedup* com o incremento do número de EPs na arquitetura paralela. Observando-se este gráfico, constata-se que quando a arquitetura possui dois EPs, o *Speedup* atingiu o valor ideal, que é igual ao número de EPs. Na medida em que foram adicionados mais EPs, o *Speedup* sofreu uma redução, atingindo 60% do valor ideal com 16 EPs. Esta redução do *Speedup* é decorrente da arquitetura e não do *software*, como explicado no parágrafo anterior.

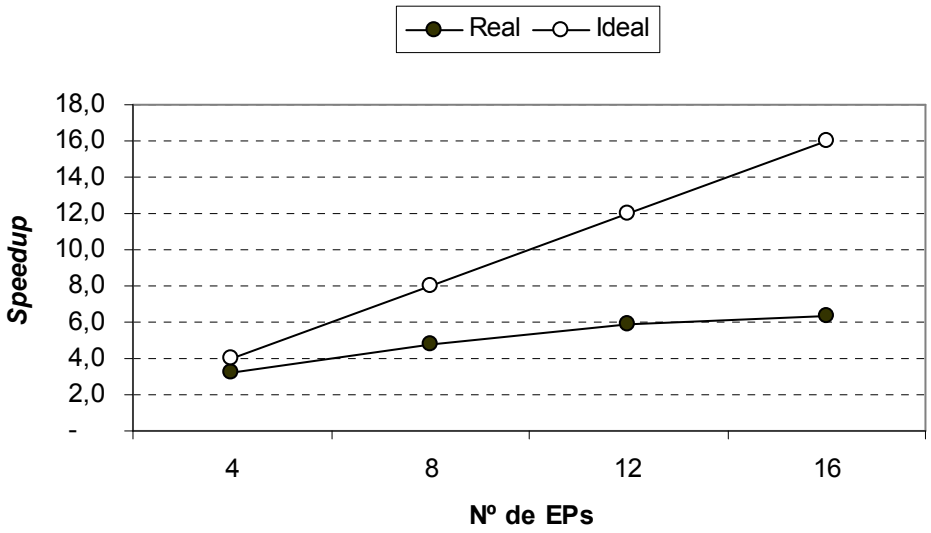


Figura 48: Gráfico do *speedup* – algoritmo independente – versão 1.2

4.5.2 Teste – Algoritmo FIR (*Finite Impulse Response*)

Os testes nas configurações com 1, 4 e 16 EPs foram realizados utilizando-se um algoritmo FIR e uma aritmética de ponto fixo com 16 *bits* de resolução. Este algoritmo foi escolhido por ser um *benchmark* utilizado em outras arquiteturas, possibilitando efetuar comparações de desempenho.

Um filtro FIR ou de resposta ao impulso finito é um tipo de filtro digital caracterizado por uma resposta ao impulso que se torna nula após um tempo finito, em contraste com os filtros IIR (*Infinite Impulse Response*). Um filtro FIR digital genérico terá uma saída dada pela Equação 11.

$$y(n) = h_0 x(n) + h_1 x(n-1) + \ldots + h_p x(n-P) \tag{11}$$

na qual P é a ordem do filtro, $x(n)$ o sinal de entrada, $y(n)$ o sinal de saída e h_i são os coeficientes do filtro. A Equação 11 também pode ser expressa pela Equação 12.

$$y(n) = \sum_{i=0}^{P} h_i x(n-i) \tag{12}$$

Na Figura 49 está apresentado o grafo de fluxo de dados do filtro FIR 5-*tap* e que foi transformado em um conjunto de *templates* próprios para a arquitetura paralela.

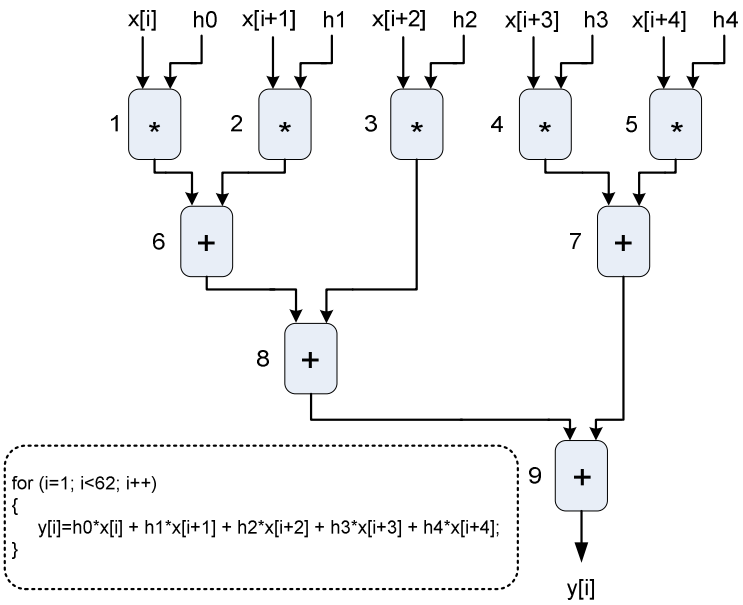


Figura 49: Grafo de fluxo de dados do filtro FIR 5-*tap*

Gradualmente, adicionaram-se EPs na arquitetura até o limite máximo de 16. Constatou-se que o tempo total de execução sofreu uma redução na medida em que se adicionavam EPs na arquitetura paralela, como mostrado na Figura 50.

Os testes revelaram que com 16 EPs o tempo de processamento ficou 60% acima do tempo ideal estimado, enquanto que com 4 EPs este valor foi acima de 80%. Contudo, ressalta-se que a aplicação é real e apresenta dependências entre as operações, o que impede a obtenção do desempenho máximo.

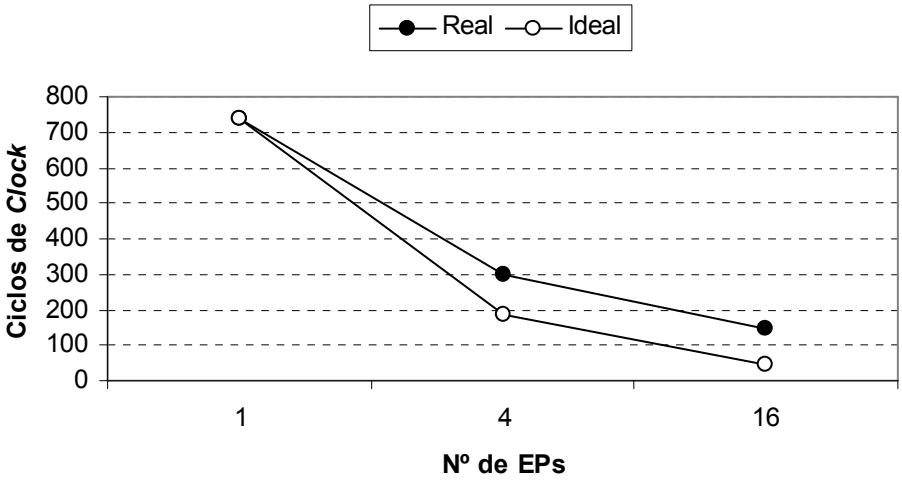


Figura 50: Tempo de execução (ciclos de *clock*) – algoritmo FIR – versão 1.2

Observando-se o gráfico da evolução do *Speedup*, assim como o incremento do número de EPs, mostrado na Figura 51, constata-se que o *Speedup* atingiu 46,88% do valor ideal. Isso é resultado da serialização no acesso à memória de *templates* em decorrência de ela ser composta por um único módulo.

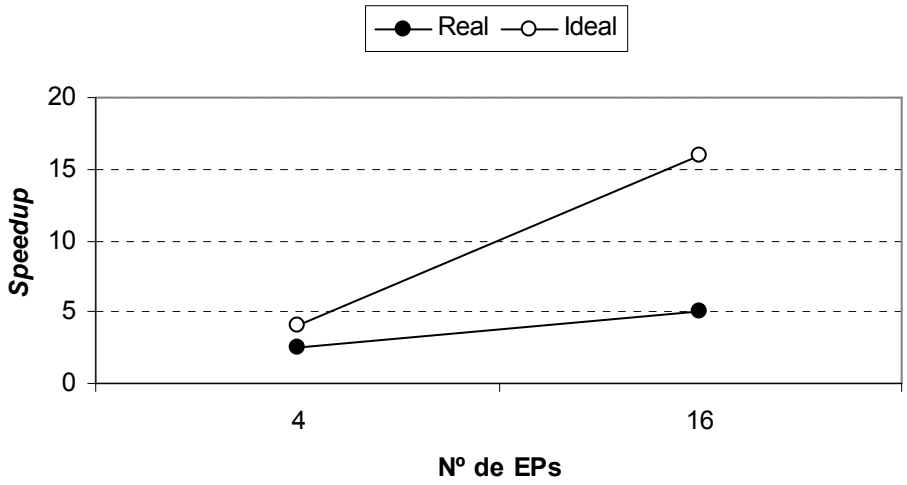


Figura 51: Gráfico do *speedup* – algoritmo FIR – versão 1.2

Somente para efeito de comparação, o mesmo teste realizado nesta arquitetura executando com 1 EP, utilizando o algoritmo FIR, também foi realizado em um computador PC Athlon XP 64 3000+ com 2,17GHz, 512M*Bytes* de memória, e com sistema operacional Windows XP. O tempo total de execução no PC é de 0,358µs, valores bem abaixo do obtido no teste da arquitetura (5,04µs). Contudo, ressalta-se que a freqüência do computador PC Athlon é aproximadamente 43 vezes maior que a utilizada nesta arquitetura paralela. Se for levado em consideração o tempo em ciclos de *clock*, constata-se que o computador PC Athlon gastou 777 ciclos, enquanto a Arquitetura Versão 1.2 com 1 EP gastou somente 740 ciclos.

Comparando-se os valores obtidos nestes testes com outros valores provenientes de testes similares realizados em outras arquiteturas (ROCCC - *Reconfigurable Computing Compiler System* e OGMS - *Optimization Generation Memory Structure for Window Operations*), como descritos em DONG *et al* (2007), que são arquiteturas dedicadas àquele tipo de aplicação, constata-se que a arquitetura obteve um tempo de execução (em ciclos de *clock*) apenas 2,8 vezes maior. O comparativo entre as arquiteturas está apresentado na Tabela 10, a qual contém freqüência de operação, tempo de execução (em ciclos), *throughput*, CPI (*Cycles Per Instruction*), quantidade de instruções executadas e MIPS (*Millions of Instructions Per Second*). O *throughput* é obtido pela divisão do número total de instruções executadas pelo tempo total de execução.

Ainda dentro desta abordagem, foram realizados testes similares em um sistema com o microcontrolador 8051 e em um sistema em FPGA com NIOS II/e embarcado. Neste caso, o tempo de execução com o 8051 ficou muito elevado (8.559 ciclos de *clock*), devido principalmente ao tempo relativo às operações de multiplicação que consomem 48 ciclos de *clock* com palavra de 8 *bits*.

Tabela 10: Comparativo de desempenho – algoritmo FIR – versão 1.2

Parâmetro	Arquitetura Versão 1.2 com 1 EP	ROCCC	OGMS	PC Athlon XP 64	8051 (8 *bits*)	NIOS II/e
Freqüência de Operação (MHz)	50	94	238,664	2.170	12	50
Tempo de Execução (ciclos de *Clock*)	740	262	263	777	8.559	1.986
Throughput	0,34	0,96	0,96	0,32	0,03	0,13
CPI	5	-	-	0,36	12	2,36
Quantidade Instruções Executadas	148	-	-	2.158	713	842
MIPS	10	-	-	5.935	1	45

Na Figura 52 está apresentado o gráfico do tempo de execução (em ciclos de *clock*) das seis arquiteturas citadas, entre elas a Arquitetura Versão 1.2 com apenas 1 EP. Pelo gráfico, constata-se que neste teste (algoritmo FIR) a arquitetura gastou um tempo menor do que o PC Athlon, 8051 e NIOS II/e e apenas 2,8 vezes mais do que nas arquiteturas dedicadas para este problema, como ROCCC e OGMS.

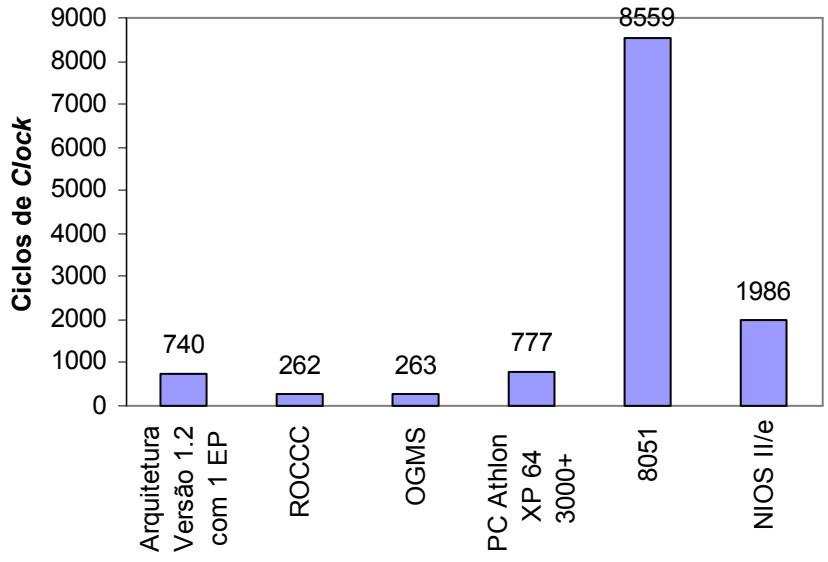


Figura 52: Comparativo de desempenho – algoritmo FIR – versão 1.2

Esta implementação no dispositivo programável FPGA EP1S10F780C7ES da família Stratix da Altera demandou 9.697 elementos lógicos e 10.240 *bits* de memória, o que corresponde a 91% e 1% dos recursos disponíveis, respectivamente. Cada EP usa 931 elementos lógicos e consome 8% dos recursos disponíveis do dispositivo.

4.6 ARQUITETURA - VERSÃO 1.3

O protocolo de testes para esta versão da arquitetura (Versão 1.3) é composto de experimentos que compreendem os algoritmos de melhor caso (com instruções independentes), filtro digital FIR (*Finite Impulse Response*), equação diferencial e criptografia IDEA (*International Data Encryption Algoritm*) em que o objetivo é provar o conceito e a viabilidade da implementação.

A implementação desta versão opera em uma freqüência de 50MHz e com um ciclo de *clock* de 20*n*s. O tempo de ciclo de execução de um *template* na arquitetura é de aproximadamente 240*n*s. Com isto, a implementação pode executar aproximadamente 4,2 milhões de operações por segundo utilizando apenas um EP.

Esta implementação é uma evolução da Versão 1.2, que possuía uma memória para 256 *templates*, 16 instruções e as unidades ainda apresentavam algumas limitações em termos de desempenho, como a unidade de despacho e armazenamento operarem sequencialmente, apesar da inclusão de *buffers* nos EPs.

Esta versão apresenta alguns melhoramentos, mostrados na Tabela 11. A implementação possui, na parte de controle, cinco unidades: armazenamento (atualiza e *snoop*), memória de *templates*, memória de dados, despacho (consulta e *buffer*), tabela de EPs e interface, como apresentado na Figura 53. Além disto, a arquitetura possui um total de 16 EPs, sendo que cada EP é composto por uma ULA de 8/16 *bits* operando com ponto fixo. A memória suporta aplicações com até 1K *templates.* Esta versão pode executar as mesmas 16 instruções da versão anterior.

A memória de *templates* é dividida em 4 módulos, com duplo acesso, possibilitando até 8 acessos simultâneos, sendo dois acessos por módulo. As unidades de armazenamento e despacho operam paralelamente e também são divididas em 4 subsistemas, aumentando o desempenho.

Tabela 11: Características da implementação – versão 1.3

Característica	Valor	Unidade
Número máximo de EPs	16	-
Freqüência de Operação	50	MHz
Complexidade do EP	ULA ponto fixo de 8/16 *bits*	-
Despacho	Paralelo/Bufferizado	-
Armazenamento	Paralelo/Bufferizado	-
Quantidade de Instruções	16	-
Tempo de *Clock*	20	*n*s
Memória de *Templates*	1K	*Templates*
Memória de Dados	1Kx16	*bits*

Uma importante modificação incorporada nesta nova arquitetura é a inclusão do processador NIOS II da Altera, também no FPGA, o qual faz a comunicação entre a máquina paralela e o computador *host*. Isto facilita o processo de envio dos *templates* e do recebimento do resultado do processamento por meio da porta USB, pois antes era feito por meio da porta serial.

Além disso, esse novo componente poderá ter um papel fundamental em trabalhos futuros no processo de gerenciamento de várias máquinas paralelas no modelo de *cluster*, possibilitando uma maior escalabilidade. Isto possibilita a expansão do número de EPs na arquitetura.

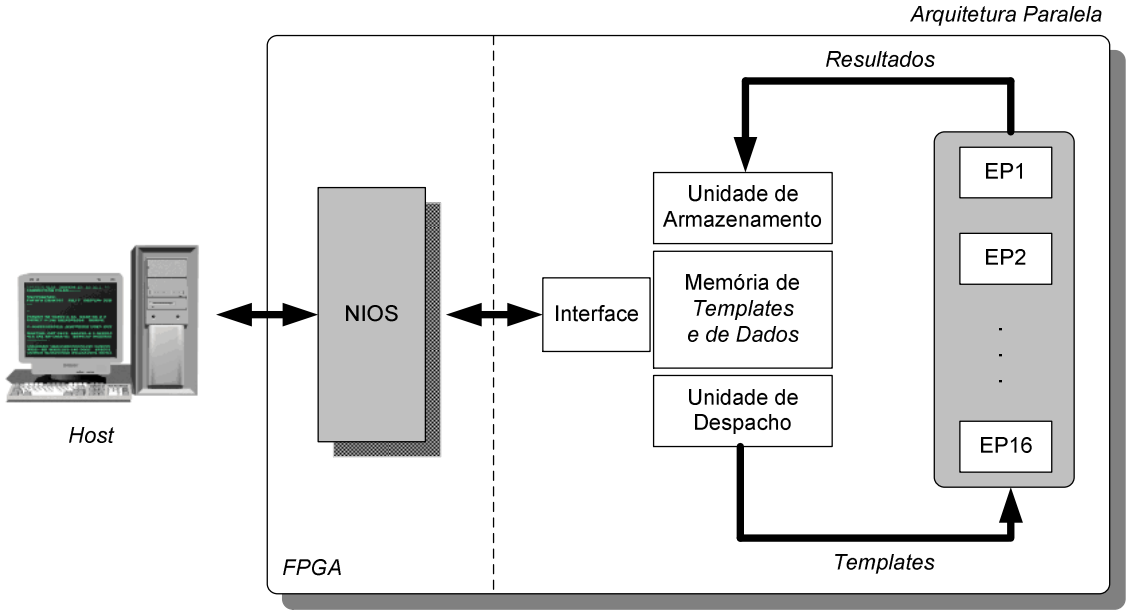


Figura 53: Visão geral da arquitetura – versão 1.3

Um detalhamento maior da arquitetura Versão 1.3 pode ser visto na Figura 54, em que percebe-se as várias unidades que foram replicadas para acessarem simultaneamente os módulos da MT de modo a explorar melhor o paralelismo.

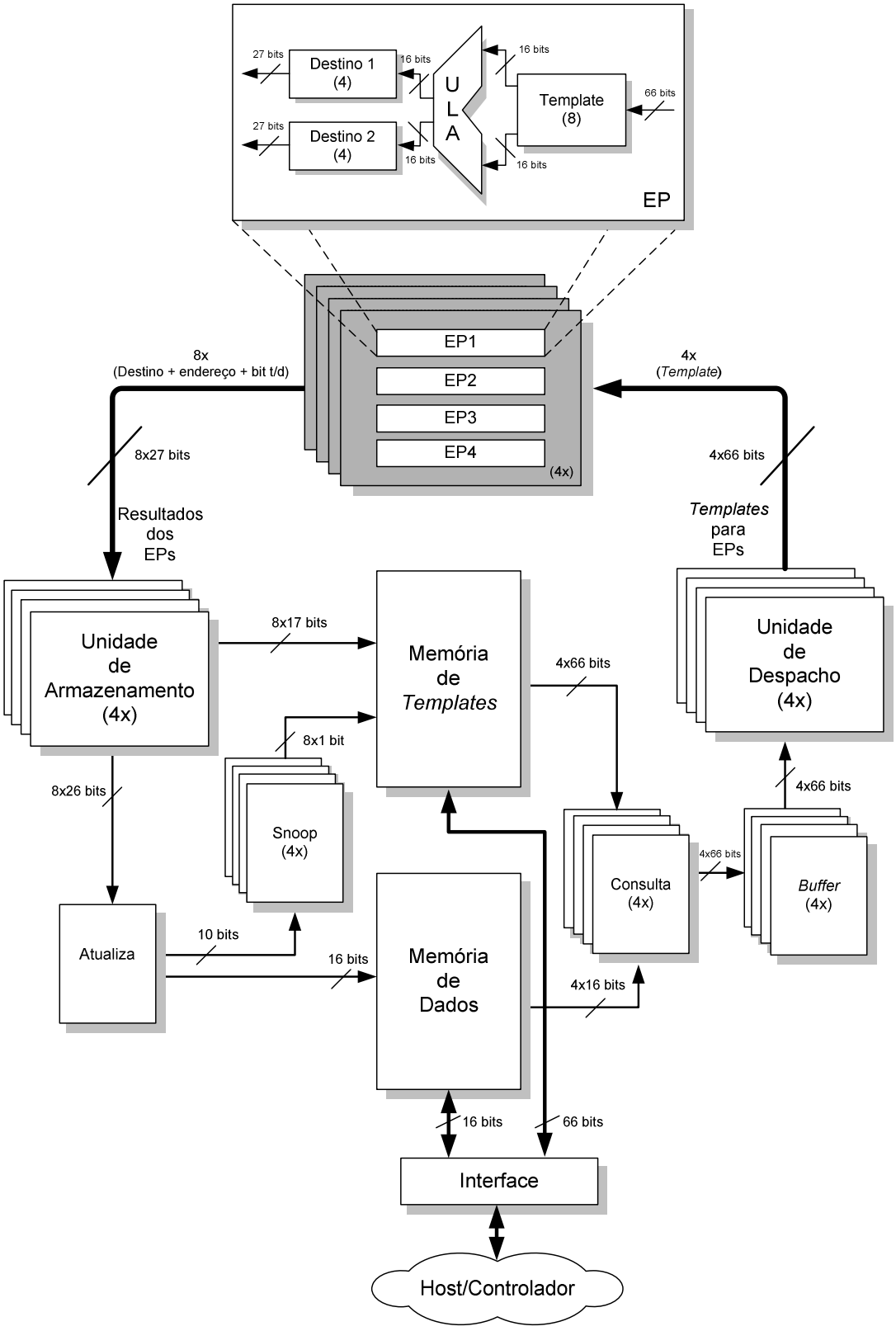


Figura 54: Detalhamento da arquitetura – versão 1.3

4.6.1 Teste – Algoritmo Equação Diferencial

Neste teste foi utilizado o processamento de cálculo numérico para uma solução da equação diferencial ordinária (Equação 1) sujeita às condições de contorno *y(0)=1* e *y'(0)=0* para 255 iterações, como mostrado no grafo da Figura 55

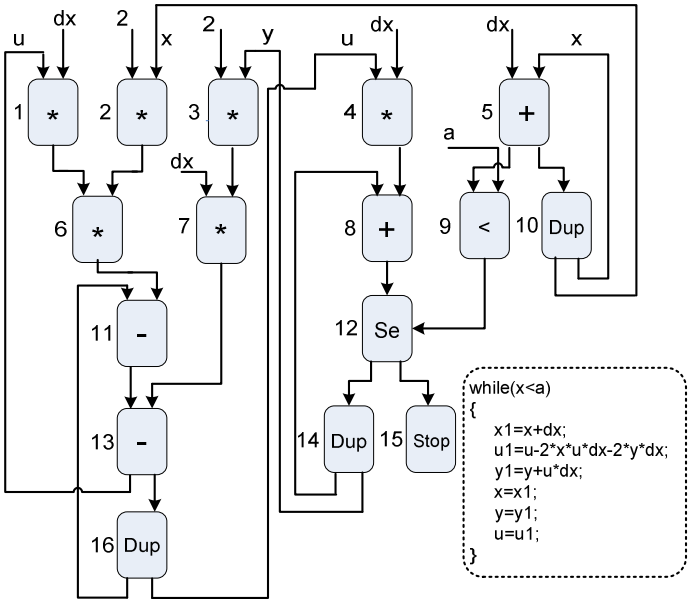


Figura 55: Grafo de fluxo de dados da equação diferencial

Os testes foram realizados em outras três diferentes arquiteturas: PC Celeron, microcontrolador 8051, processador NIOS II/e e nesta versão da arquitetura com 1 EP. Na Tabela 12 estão mostrados os resultados dos testes com palavras de inteiros de 16 *bits* nas quatro arquiteturas com apenas 1 EP, a qual contém a freqüência de operação, tempo de execução (em ciclos), *throughput*, CPI, quantidade de instruções executadas e MIPS.

Tabela 12: Comparativo de desempenho – algoritmo equação diferencial – versão 1.3

Parâmetro	Arquitetura Versão 1.3 com 1 EP	PC Celeron	8051	NIOS II/e
Freqüência de Operação (MHz)	50	2.222	12	50
Tempo de Execução (ciclos de *clock*)	22.925	8.227	593.652	5.866
Throughput	0,18	0,50	0,01	0,70
CPI	12	0,37	12	2,36
Quantidade de Instruções	1.910	22.235	49.471	2.486
MIPS	4	5.935	1	45

Na Figura 56 está apresentado o gráfico do tempo de execução (em ciclos de *clock*) das quatro arquiteturas citadas; entre elas a Arquitetura Versão 1.3 com apenas 1 EP.

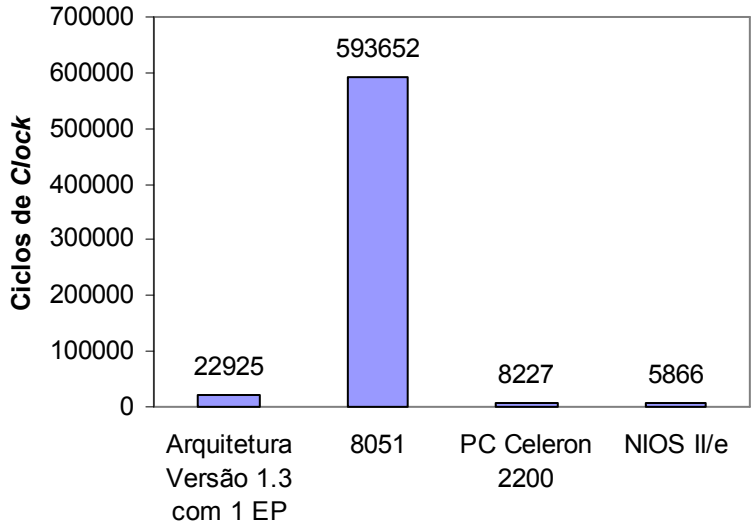


Figura 56: Comparativo de desempenho – algoritmo equação diferencial – versão 1.3

Gradualmente, adicionaram-se EPs na arquitetura até o limite máximo de 16. Constatou-se que o tempo total de execução sofreu uma redução principalmente quando a arquitetura tem 16 EPs, como mostrado na Figura 57. No caso em que há 4 EPs na arquitetura, não foi observado redução do tempo de execução, pelo contrário, houve até mesmo um pequeno acréscimo, mas isto é decorrente do pequeno grau de paralelismo do algoritmo e também por utilizar um bloco único de UA, MT e UD, ocasionando uma execução seqüencial.

Os testes revelaram que, em particular, este algoritmo apresenta poucas operações que podem ser executadas em paralelo, o que resulta no pequeno ganho de desempenho, em que se obteve 7% do ganho real.

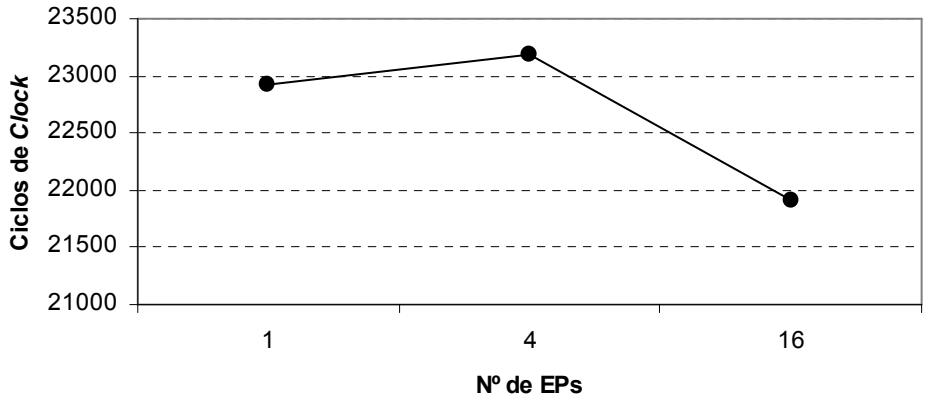


Figura 57: Tempo de execução (ciclos de *clock*) – algoritmo equação diferencial – versão 1.3

4.6.2 Teste – Algoritmo Independente (32 operações) e Algoritmo FIR (5-*tap*)

Os mesmos algoritmos Independente (com 32 instruções) e FIR (5-*tap*) utilizados na Versão 1.2 foram executados nesta versão apenas com os ajustes para adequação à arquitetura e considerando que operam com números inteiros de 16 *bits*.

Os testes utilizando os algoritmos Independente e FIR na arquitetura com memória única (como se fosse um único módulo FD), considerando 1, 2 e 4 EPs, indicam que os tempos de execução tiveram um valor acima do tempo ideal. Os tempos do algoritmo Independente em 36,41% e os tempos do algoritmo FIR em 45,85%, como pode ser observado na Figura 58. A diferença de 9,44% entre os tempos de execução é predominantemente decorrente das dependências inerentes ao algoritmo FIR.

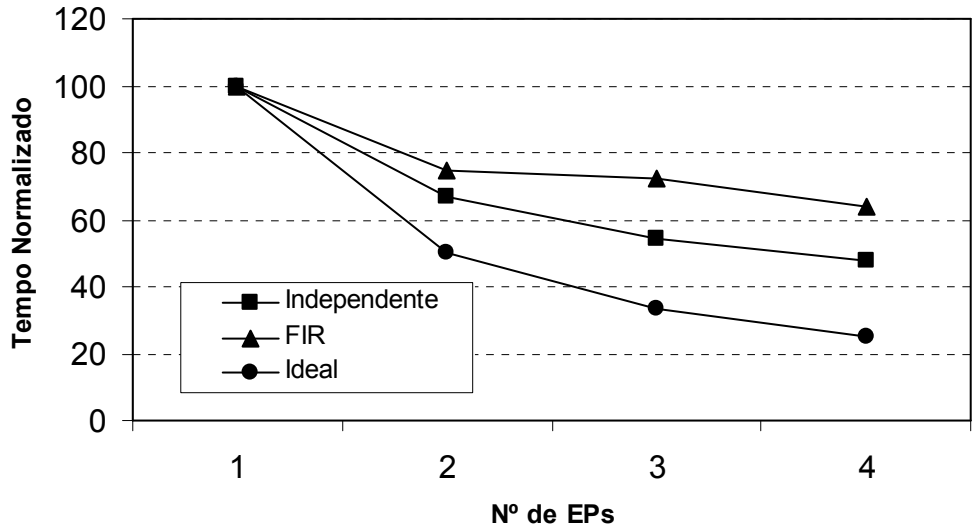


Figura 58: Tempo de execução – algoritmos independente e FIR - versão 1.3 (memória única)

Em termos do *Speedup*, os testes mostraram que o algoritmo Independente atingiu 62,90% do valor ideal nas execuções com 2, 3 e 4 EPs, enquanto que o algoritmo FIR alcançou 50,81%, como pode ser visto na Figura 59. Em ambos os casos, considerou-se a arquitetura com um único módulo de memória.

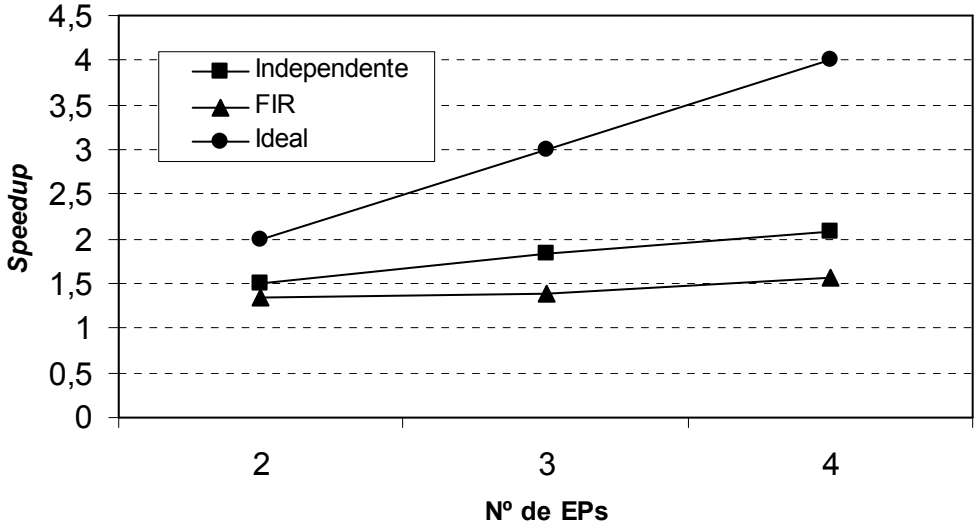


Figura 59: Gráfico do *speedup* – algoritmos independente e FIR - versão 1.3 (memória única)

Na Figura 60 mostra-se a simulação da execução do algoritmo FIR na arquitetura com 4 EPs na configuração de um módulo único de memória. O início do processamento ocorre com a borda de subida do sinal *início* em 2,49µs e o seu término ocorre após a última gravação na memória em 8,29µs. Convém destacar que o sinal *fim* é acionado antes do término do processamento e que neste caso foi em 8,17µs. Desta forma, esta execução consome 290 ciclos de *clock* cada um com período de 20ns. Na figura visualiza-se também o acesso a cada fatia da memória (operando 1 e 2) representado pelos sinais *wr1* e *wr2*.

Repetindo os mesmos testes agora na arquitetura com memória dividida em 4 módulos e considerando 4, 8, 12 e 16 EPs, obteve-se os gráficos da Figura 61. Observa-se que na configuração com 4 EPs o tempo de execução do algoritmo Independente foi igual ao valor ideal, devido às otimizações nas instruções, e o tempo médio de execução foi de 34,17% acima do tempo ideal. Por sua vez, o algoritmo FIR obteve um valor de 41,84% acima do ideal, mas ficou 7,38% acima na situação com 4 EPs. A diferença média entre a execução do algoritmo Independente e a FIR foi de 5,72%, com exceção para a execução com 16 EPs, em que a diferença foi de 11,58%, denotando uma limitação na execução do algoritmo FIR, decorrente dos acessos à memória ocasionados pelas dependências presentes na aplicação e, também, devido a uma restrição no armazenamento dos resultados.

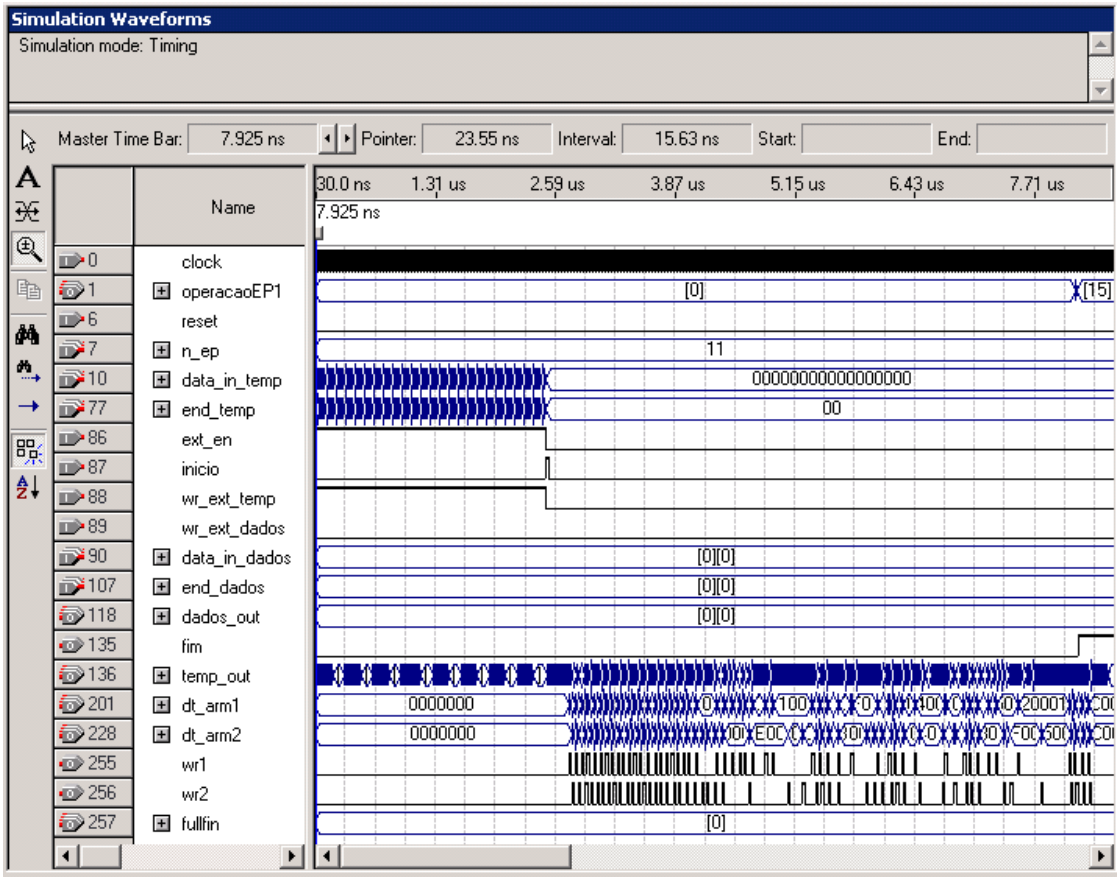


Figura 60: Simulação da execução com 4 EPs – algoritmo FIR - versão 1.3 (memória única)

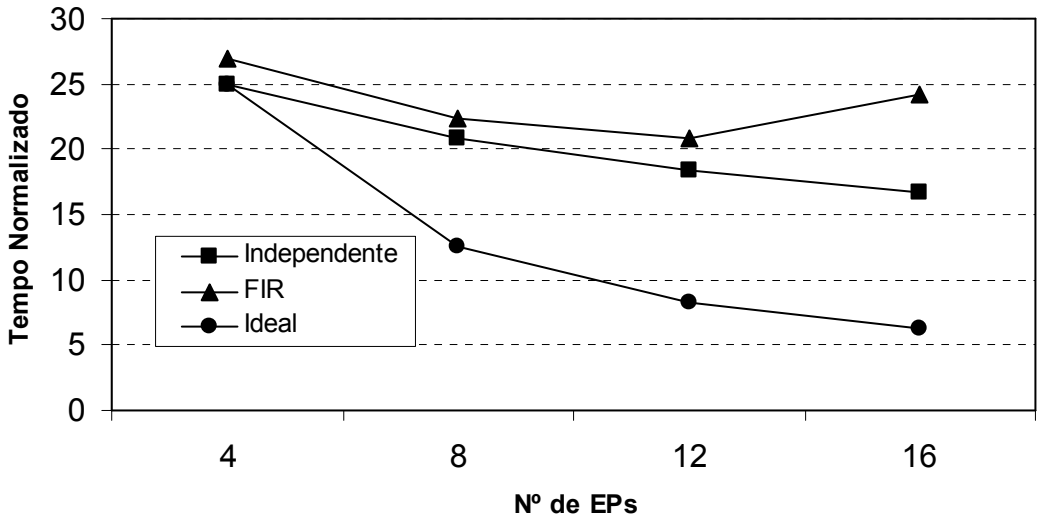


Figura 61: Tempo de execução - algoritmos independente e FIR - versão 1.3 (memória dividida)

Considerando o *Speedup* na configuração da arquitetura em que a memória é dividida em 4 módulos, constata-se que em média atingiu-se 71,36% do valor ideal com o algoritmo

Independente, com a ressalva de que na execução com 4 EPs o valor foi igual ao do melhor caso, *Speedup* igual a 4, como mostrado na Figura 62. Nos testes com o algoritmo FIR atingiu-se 65,73% do valor ideal, sendo que no teste com 4 EPs o valor ficou muito próximo do ideal.

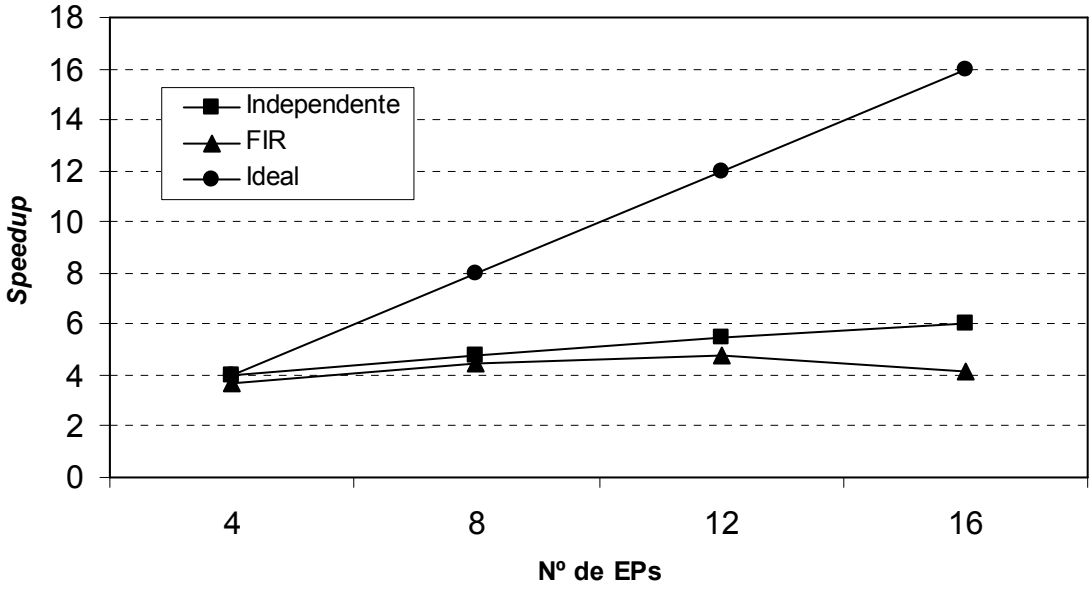


Figura 62: Gráfico do *speedup* - algoritmos independente e FIR - versão 1.3 (memória dividida)

A Figura 63 contém a simulação da execução do algoritmo FIR na arquitetura com 4 EPs na configuração de memória segmentada (4 módulos). Nesta situação o início do processamento acontece em 2,49µs e o término do processamento ocorre em 4,91µs, mas o sinal *Fim* foi acionado em 3,27µs. Isto representa um tempo de processamento de 2,42µs, ou seja, 121 ciclos de *clock* com período de 20ns. Na figura observa-se também a atualização dos módulos de memória, e isto é representado pela ativação dos sinais *pin_name*.

A simulação do algoritmo FIR na execução com 16 EPs, na configuração com a memória dividida em 4 módulos, está mostrada na Figura 64. Também observa-se a atualização dos 4 módulos da memória, indicado pela ativação dos sinais *pin_name*. O tempo de processamento desta simulação consumiu 2,18µs, ou seja, 109 ciclos de *clock* (20ns), resultando em um *Speedup* de 4,15 em relação à execução seqüencial.

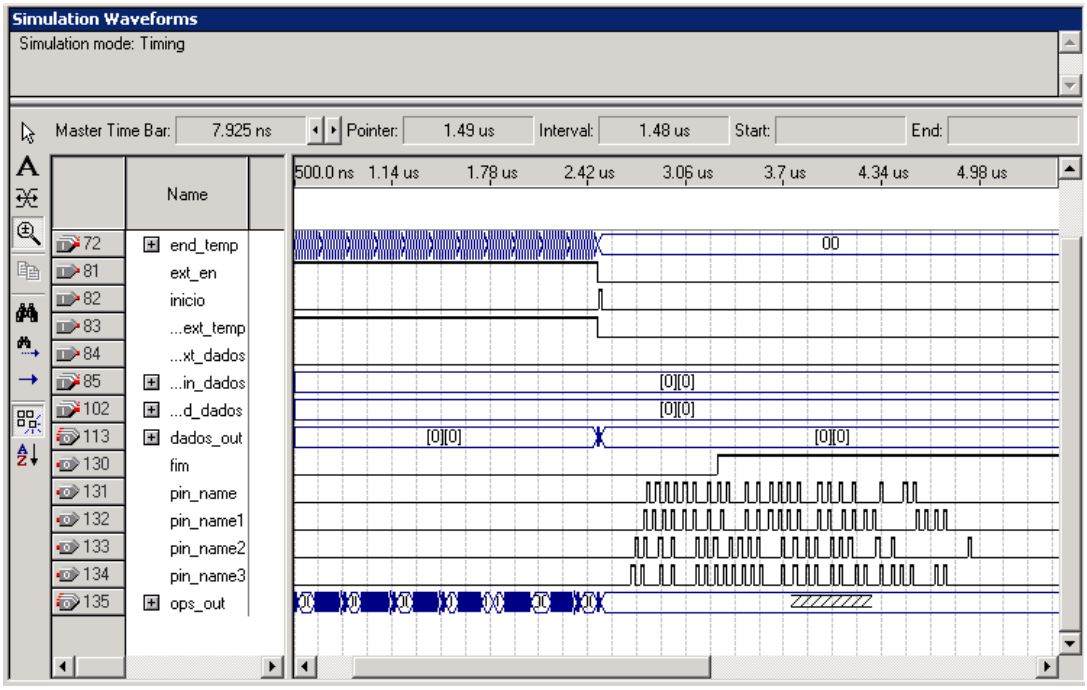


Figura 63: Simulação da execução com 4 EPs – algoritmo FIR - versão 1.3 (memória dividida)

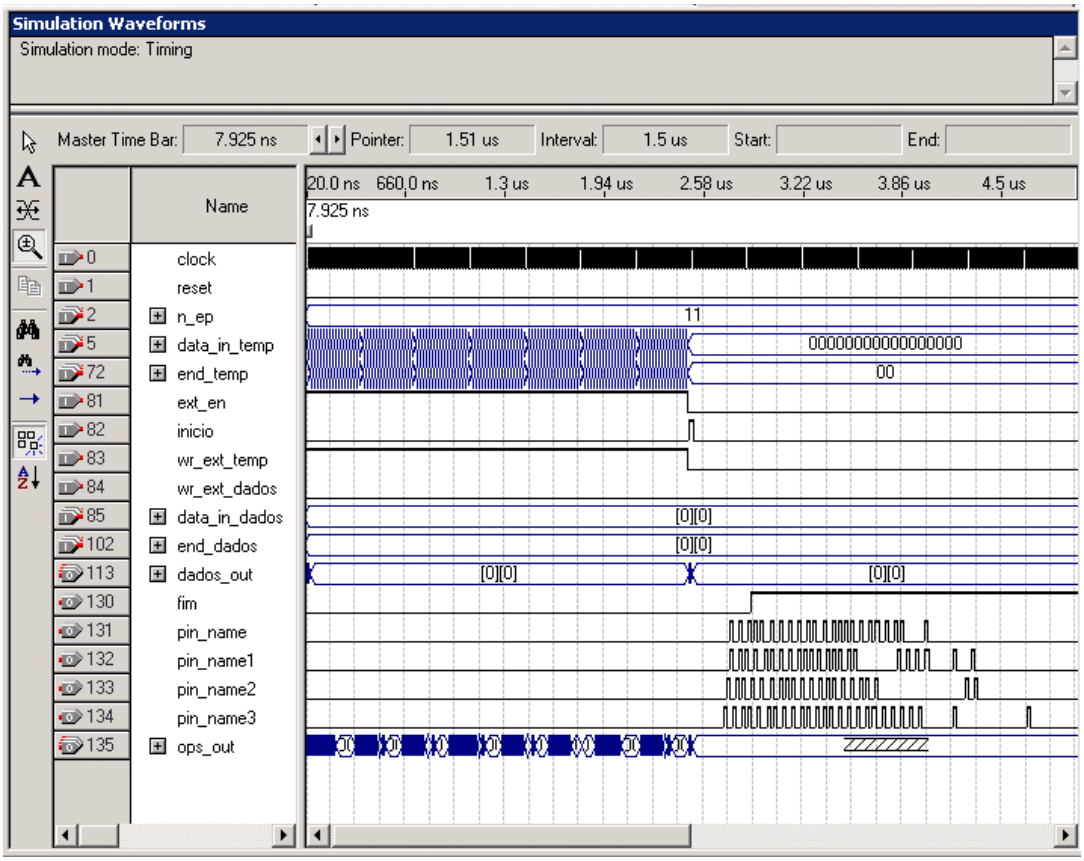


Figura 64: Simulação da execução com 16 EPs – algoritmo FIR - versão 1.3 (memória dividida)

4.6.3 Teste – Algoritmo de Criptografia IDEA (*International Data Encryption Algorithm*)

Outra classe de aplicação sobre a qual foram realizados testes são os algoritmos de criptografia, em especial o IDEA (*International Data Encryption Algorithm*) (SCHNEIER, 1996), pois neste tipo de aplicação tem-se operações que podem ser realizadas em paralelo.

O algoritmo IDEA emprega três operações: OU-exclusivo, adição com módulo 2^{16} e multiplicação de módulo $2^{16}+1$. Algumas operações da função IDEA podem ser realizadas em paralelo, enquanto outras devem ser realizadas sequencialmente. Contudo, o número máximo de operações que são realizadas em paralelo é 4, e isto é decorrente do algoritmo, como pode ser observado na Figura 65.

No algoritmo, as variáveis P1 a P4 representam blocos de informação de 16 *bits*, que juntos formam um bloco de informação de 64 *bits*, denominado de cipher-texto, e S1 a S6 as sub-chaves de criptografia de 16 *bits*. O processo inteiro de criptografia envolve a repetição de 8 vezes o algoritmo apresentado, utilizando-se um total de 52 sub-chaves geradas a partir de uma chave de criptografia de 128 bits.

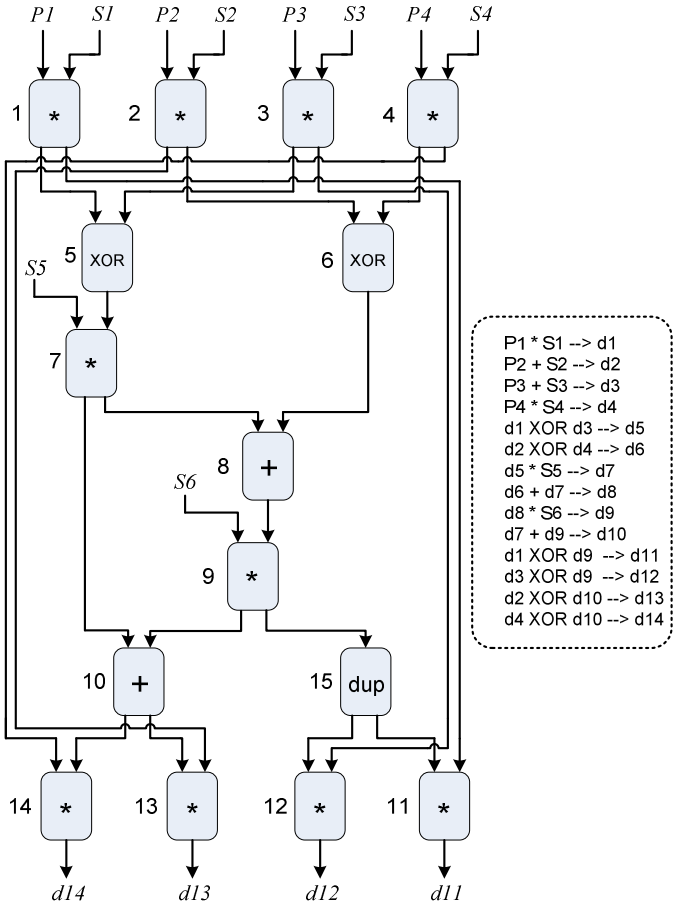


Figura 65: Grafo de fluxo de dados do algoritmo IDEA

A Figura 66 apresenta os resultados obtidos para o algoritmo IDEA. Nesta figura pode-se observar que a arquitetura proposta (Versão 1.3 com 16 EPs) é aproximadamente 15 vezes mais rápida que uma implementação em C++ que executa sobre um processador Pentium II ou um Pentium 4, como resultado dos testes realizados. Quando comparado com o sistema HiPCrypto (SALOMAO *et al*, 1998), que é um *hardware* especializado (ASIC) para o algoritmo IDEA, a arquitetura é 3 vezes mais lenta. Na comparação com a arquitetura reconfigurável Morphosys (SING *et al*, 2000), que é uma arquitetura de co-processador SIMD (*Single Instruction Multiple Data*) em que as operações são mapeadas nos EPs, e que foi configurada especificamente para a aplicação IDEA, a arquitetura proposta é 5 vezes mais lenta.

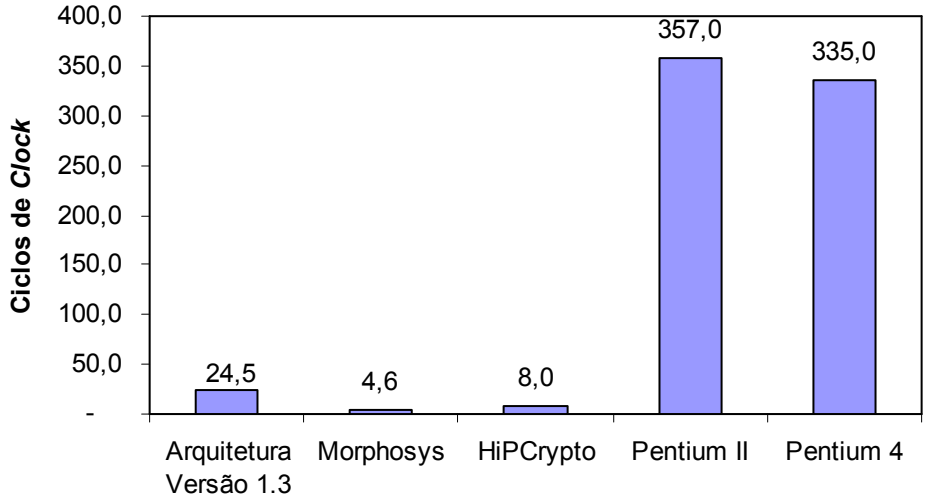


Figura 66: Ciclos para um bloco de cipher-texto – versão 1.3

A arquitetura proposta consegue produzir 10 cipher-texto em 245 ciclos, sendo que este valor de ciclos pode ser reduzido com a execução de uma quantidade maior de cipher-texto em paralelo, como verificado na Figura 67. A limitação para isto é a quantidade de memória de *templates* disponível para comportar os trechos de *templates* do algoritmo. A execução na Morphosys consome 73 ciclos para produzir 16 blocos de cipher-texto. Um *chip* HiPCrypto produz 7 cipher-texto a cada 56 ciclos. A implementação em *software* do IDEA sobre um processador Pentium II requer 357 ciclos e em um processador Pentium 4 o consumo é de 335 ciclos para um cipher-texto.

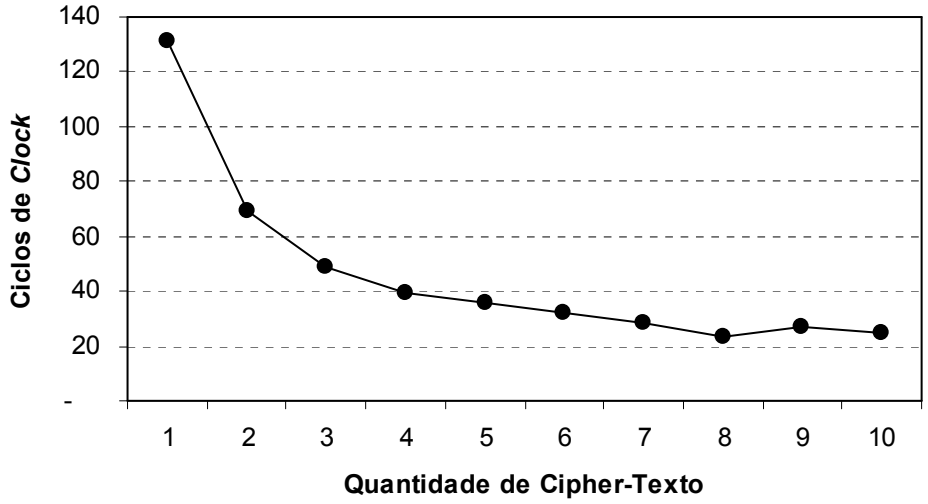


Figura 67: Ciclos de *clock* por bloco de cipher-texto – versão 1.3

Desta forma, observa-se que a arquitetura proposta Versão 1.3 apresentou um ganho expressivo na comparação com as arquiteturas Pentium e, apesar do resultado ser um pouco acima do obtido pelas arquiteturas Morphosys e HiPCrypto, ainda assim apresenta um bom desempenho, se considerarmos que estas duas arquiteturas são específicas (ou no caso da Morphosys configurada especialmente para o problema) para a execução do algoritmo IDEA.

Esta implementação no dispositivo programável FPGA EP2S60F672C3 da família Stratix II da Altera demandou 28.348 elementos lógicos e 132.160 *bits* de memória, o que corresponde a 60% e 5% dos recursos disponíveis, respectivamente.

4.7 COMPARATIVO COM OUTRAS ARQUITETURAS

O estudo comparativo da execução do algoritmo FIR nas duas implementações (Versão 1.2 e 1.3) e nas outras arquiteturas (dedicadas, ASIC, *soft processors*, microprocessadores, DSPs, reconfiguráveis e *dataflow*), tem por objetivo confrontar os resultados decorrentes das execuções nas diversas plataformas.

Com base nos dados da Tabela 13, observa-se que a arquitetura Versão 1.3 com 8 EPs apresentou o melhor desempenho em termos de ciclo de *clock* gastos para a aplicação FIR, inclusive ganhando de arquiteturas dedicadas (ROCCC e OGMS), de processadores DSPs (*Digital Signal Processors*) (BF533, TMS 320C2, TMS 320C5x, TMS 320C55x), dos *soft processors* (NIOS II e SPREE), dos microprocessadores (Athlon e 8051) ou mesmo das

arquiteturas reconfiguráveis (KressArray (24 EPs) e COLT (16 EPs)). Isto também pode ser constatado pelo consumo de ciclos de *clock* por *tap*, em que a arquitetura apresenta o menor valor (aproximadamente 2). Este valor é 2 vezes menor do que o das arquiteturas dedicadas e DSPs, 4,5 vezes menor ao das arquiteturas reconfiguráveis, 6,5 vezes menor do que a do PC Athlon, 16 vezes menor do que o processador embarcado NIOS e 69 vezes menor do que o microcontrolador 8051.

Tabela 13: Comparativo entre arquiteturas

Arquitetura	Nº EPs	Freq (MHz)	Ciclos	Throughput	MIPS	CPI	IPC	IPC/EP	Ciclos/tap
Versão 1.3									
V3 1EP	1	50	452	0,56	13	3,70	0,27		7
V3 4EP	4	50	122	2,07	49	1,00	1,00	0,25	2
V3 8EP	8	50	101	2,50	60	0,83	1,20	0,15	2
V3 12EP	12	50	94	2,68	65	0,77	1,30	0,11	2
V3 16EP	16	50	109	2,31	56	0,89	1,12	0,07	2
Versão 1.2									
V2 1EP	1	50	740	0,34	8	6,17	0,16		12
V2 4EP	4	50	296	0,85	20	2,47	0,40	0,10	5
V2 16EP	16	50	148	1,70	41	1,23	0,81	0,05	2
ASIC									
ROCCC [1]		94	262	0,96					4
OGMS [1]		239	263	0,96					4
Soft Processors									
NIOS II/e	1	50	1986	0,13	45	2,36	0,42		32
SPREE [8]	1	80	1145	0,22	53	1,36	0,74		18
Microprocessadores									
PC Athlon	1	2170	777	0,32	5935				13
8051	1	12	8559	0,03	1	12,00	0,08		138
***Digital Signal Processors* (DSP)**									
ADSP BF533 [4]	1	750	186		500				3
TMS 320C2[4]	1	300	248		12				4
TMS320C5X [3]	1	29	434		50				7
TMS320C55X [6]	1	200	504		400				4
Arquiteturas Reconfiguráveis									
KressArray [2]	24	25	558						9
COLT [5]	16	50	496						8
WaveScalar [10]	512					0,01	71,43	0,14	
TRIPS [7]	256					0,09	11,00	0,04	
Arquitetura *Dataflow*									
Manchester [9]	20	15			6	2,50	0,40	0,02	

[1] (DONG *et al*, 2007), [2] (HARTENSTEIN *et al*, 1994), [3] (PETERSEN, 1995), [4] (MYJAK, M.J., DELGADO-FRIAS, 2008), [5] (CHERBAKA, 1996), [6] (VARNAGIRYTE et *al*, 2002), [7] (SANKARALINGAM *et al*, 2003), [8] (YIANNACOURAS, STEFFAN & ROSE, 2007), [9] (SILC, ROBIC & UNGERER, 1999), [10] (SWANSON et al, 2003)

Comparando a arquitetura proposta com a WaveScalar (SWANSON *et al*, 2003), contata-se que esta atinge um IPC (*Instructions Per Cycle*) por EP de aproximadamente 0,14, enquanto a arquitetura alcança um valor maior de aproximadamente 0,15 para uma aplicação

FIR. Isto representa um ganho de aproximadamente 8,5%, indicando que a arquitetura possui um melhor desempenho em termos de instruções executadas por ciclo de *clock*. A mesma análise é válida na comparação da arquitetura proposta com a arquitetura TRIPS (SANKARALINGAM *et al*, 2003) em que a diferença é de 3,75 vezes, pois a relação IPC por EP da TRIPS é 0,04.

A comparação com a arquitetura *dataflow* original Manchester (SILC, ROBIC & UNGERER, 1999) indica que a arquitetura proposta obtêm um valor de 10,5 vezes maior em termos de MIPS, pois a arquitetura Manchester, com 20 EPs (ULAs), atingiu 6 MIPS, enquanto a proposta, com 16 EPs, obteve um valor de 63 MIPS de pico. Outrossim, considerando-se a mesma freqüência de *clock* a arquitetura proposta têm um ganho em MIPS superior a 3 vezes.

4.8 ANÁLISE DE DESEMPENHO

Este estudo envolve a análise de desempenho das Versões 1.2 e 1.3 da arquitetura considerando a execução dos algoritmos (melhor caso e FIR) em comparação com a execução ideal, tomando por base o tempo de processamento, *speedup* e eficiência.

Comparando as curvas dos tempos Ideal, FIR e Independente normalizados para a execução com 1, 4 e 16 EPs, mostradas na Figura 68, percebe-se que em média a diferença entre os valores Ideais e Independentes é de 9,44% e entre Ideal e FIR é de aproximadamente 17,33%. Teoricamente, esse valor de 9,44% obtido pela diferença média entre o valor Ideal e o Independente é decorrente das restrições impostas pela arquitetura, como quantidade de barramentos, o que não envolveria fatores de *software*, pois as instruções são totalmente independentes, ou seja, execução totalmente paralela.

Se for assumido que a curva Independente representa o paralelismo máximo que pode ser obtido em uma execução paralela, em virtude de não haver dependência entre as instruções, então a diferença média de 7,90%, entre as curvas Independente e FIR, representa a parcela de dependências do algoritmo FIR. Isto implica dizer que, *a priori*, aproximadamente 7,90% do código é exclusivamente seqüencial.

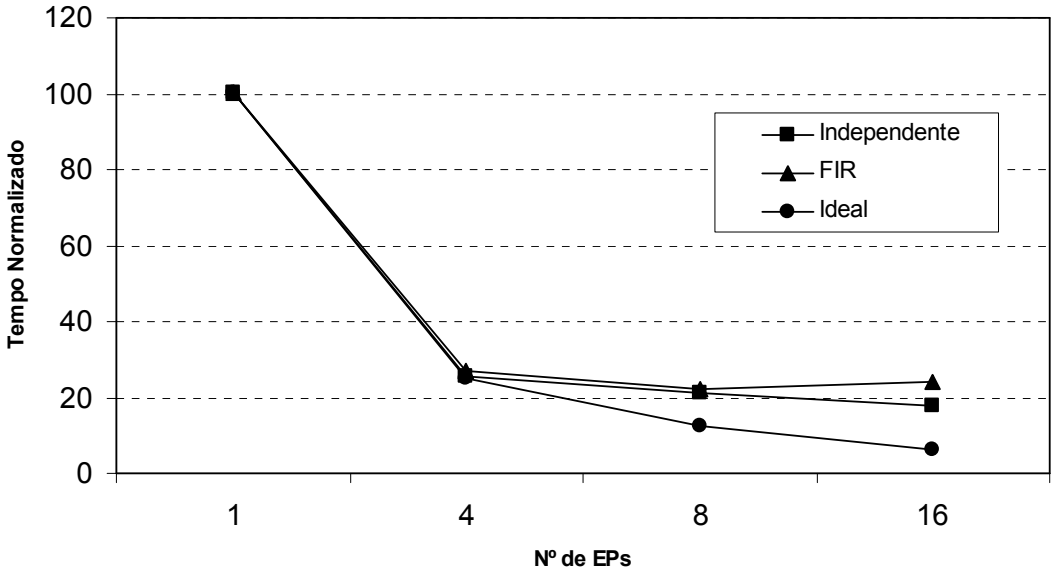


Figura 68: Curvas de tempo normalizado das execuções FIR, independente e ideal

O *Speedup* possui dois limitantes: o primeiro é a parcela do *software* seqüencial e o segundo a arquitetura, em particular, o número de processadores. Caso o *software* seja 100% paralelo, a limitação é decorrente exclusivamente da arquitetura. Utilizando a Equação 6 do *Speedup*, pode-se derivar a Equação 13, a qual pode ser utilizada para se calcular a parcela do *software* (f) que é seqüencial com base no número de processadores (N) e no *Speedup* (S).

$$f = \frac{\left(\frac{N}{S}\right) - 1}{N - 1} \tag{13}$$

Observando-se os gráficos do *Speedup*, mostrados na Figura 69, constata-se que as curvas FIR e Ideal estão próximas e apresentam uma diferença média de 20,61%. O algoritmo FIR alcança *Speedup* de 5,0 na execução com 16 EPs e este valor está próximo do $Speedup_{máximo}$ que neste caso é de 5,7, considerando que 17,33% do código é seqüencial (obtido pela Equação 13). O $Speedup_{máximo}$ é o inverso da porcentagem da parcela do código seqüencial e pode ser obtido pela aplicação dos limites na Equação 6, quando o número de processadores tende a infinito. Isto implica dizer que quanto menor for a parcela do código seqüencial, maior será o *Speedup*.

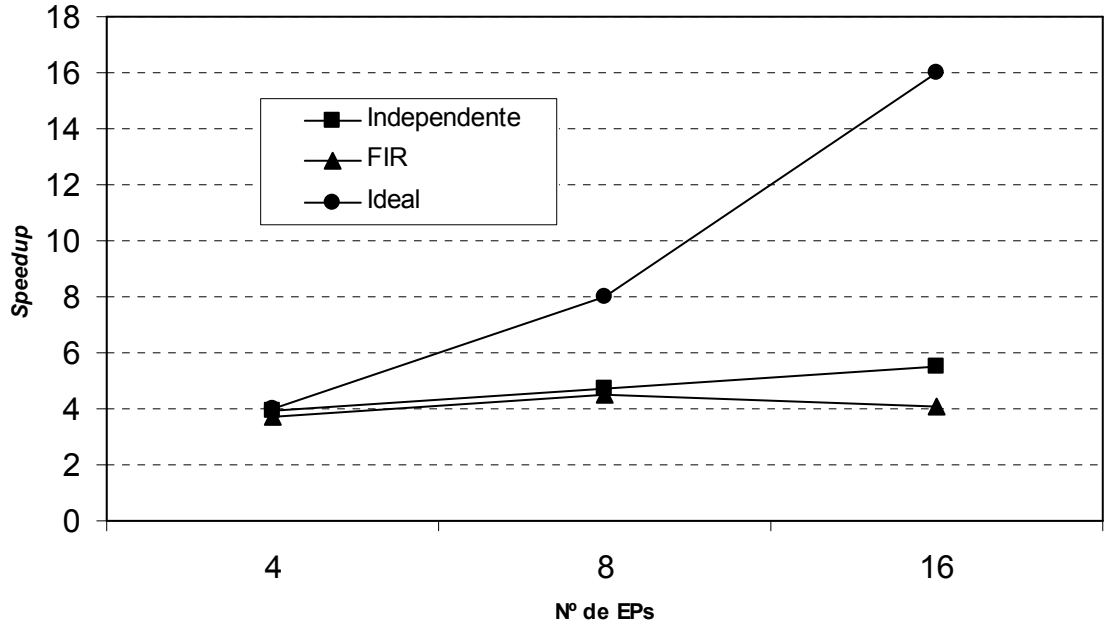


Figura 69: Curvas do *speedup* das execuções FIR, independente e ideal

Contudo, se for utilizado o valor de 7,90% do código como sendo a parcela seqüencial, o $Speedup_{máximo}$ que pode ser alcançado é de 12,6 e, teoricamente, seria alcançado somente na execução com 2.522 EPs, conforme calculado pela Equação 14. O número de processadores (N) em uma arquitetura pode ser determinado pela Equação 14, que é obtida com base na definição de *Speedup* (S) (Equação 6) e da parcela do código (f) que é seqüencial.

$$N = \frac{S(1-f)}{1-Sf} \tag{14}$$

A eficiência também é decorrente do *Speedup*, pois é definida como sendo a razão entre o *Speedup* e o número de processadores. Desta forma, pode-se utilizá-la para delimitar o *Speedup* desejado e em função disto definir o número de processadores na arquitetura com base na parcela do *software* seqüencial. Efetuando as devidas substituições na Equação 6, obtêm-se a Equação 15 que possibilita calcular a eficiência (E) utilizando o número de processadores (N) e a parcela do código (f) seqüencial.

$$E = \frac{1}{f(N-1)+1} \tag{15}$$

Nos testes realizados, a eficiência máxima obtida no algoritmo Independente foi de 80% e no caso do algoritmo FIR o valor foi de 62,50%.

Comparando as duas execuções do algoritmo FIR nas Versões 1.2 e 1.3, mostradas na Figura 70, observa-se que a Versão 1.3 apresenta resultados melhores com uma diferença média de 58,65%. Dois itens implementados na Versão 1.3 foram os grandes responsáveis por esse resultado: a adoção de *buffers* nos EPs e os despachos das operações em paralelo devido à utilização de unidades e barramento replicados.

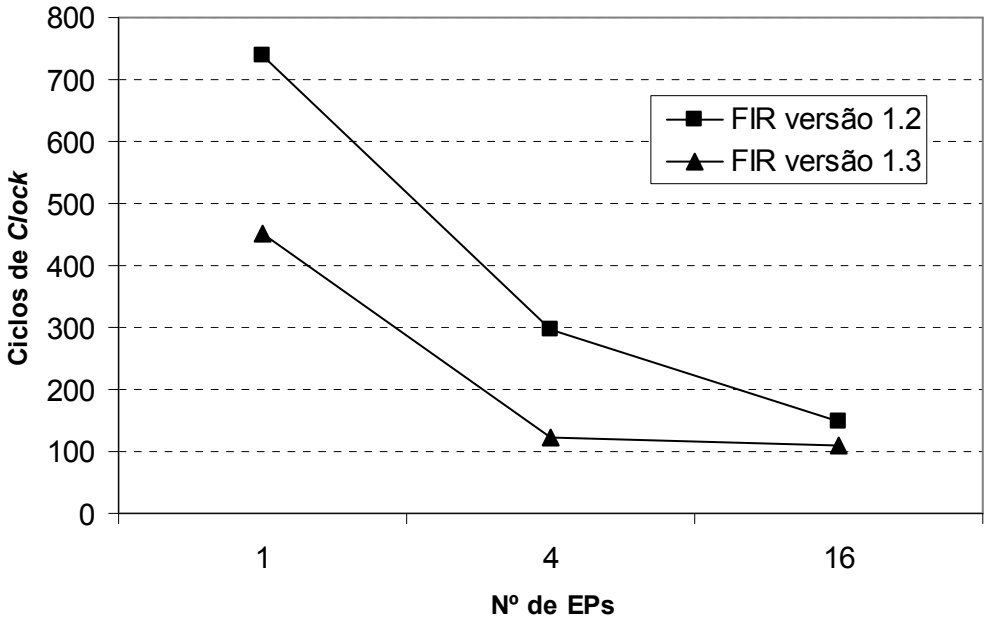


Figura 70: Comparativo tempo de execução (ciclos de *clock*) – algoritmo FIR – versão 1.2 e 1.3

CAPÍTULO 5

DISCUSSÃO E CONCLUSÃO

5.1 ANÁLISE DOS RESULTADOS

O estudo dos diagramas de temporização, considerando a arquitetura com e sem *buffer* de entrada e de saída nos EPs e tomando por base as três situações (pior, intermediário e melhor caso), apontou que a adoção de *buffers* garante um ganho em termos de ciclos de *clock* de 81,25% e um aumento de 32,50% em relação ao *Speedup* no caso da arquitetura sem *buffer*. Desta forma, a incorporação de *buffers* nos EPs possibilita um ganho real de desempenho, no pior caso, de aproximadamente 29%, em virtude de liberar as unidades de despacho e de armazenamento durante o processamento dos *templates* encaminhados para os EPs. Ainda nesta análise, constatou-se que a utilização dos *buffers* eliminou os efeitos das restrições presentes nos testes no desempenho, já que os três casos apresentaram resultados iguais. Isto é diferente do que ocorreu nos testes com a arquitetura sem *buffer*, em que as restrições ficaram evidentes nos resultados.

Os testes realizados nas duas configurações de ULA de 32 *bits*, ponto fixo e ponto flutuante, mostraram que as operações com ponto flutuante gastam uma considerável quantidade de ciclos de *clock* a mais do que as operações com ponto fixo. Normalmente, as operações com ponto fixo gastam 1 ciclo de *clock*, enquanto algumas operações com ponto flutuante podem chegar até 18 ciclos. Estes resultados eram esperados, principalmente no caso das operações em ponto fixo. Além disto, é importante destacar que uma ULA de ponto fixo consome bem menos recurso do dispositivo FPGA do que uma com ponto flutuante, justamente pela maior complexidade das operações destas últimas e, consequentemente, uma correspondente utilização de portas lógicas para a implementação dos circuitos.

Em outro teste foram obtidos os tempos de execução (em ciclos de *clock*) para as operações definidas para arquitetura em que as instruções são de ponto fixo, operando sobre dados com tamanhos de 8, 16 e 32 *bits*. Os resultados das operações para as três configurações foram iguais, apenas com uma divergência para o caso da divisão com 32 *bits*. Isto denota que, no caso com operações em ponto fixo, a utilização de tamanhos maiores dos dados não acarreta prejuízo no desempenho, podendo-se utilizar palavras grandes sem resultar

em um aumento do tempo de processamento. Contudo, isto não é verdade em relação ao consumo de recursos do dispositivo, pois quanto maior for o tamanho dos dados, maior será a utilização de portas lógicas, o que pode ser um problema dependendo da capacidade do dispositivo.

Foram realizadas três implementações básicas para a arquitetura. Em cada uma delas foram incorporados algumas melhorias, como quantidade de EPs, barramentos paralelos, replicação das unidades, *buffers*, interface, entre outros. Estes aperfeiçoamentos se mostraram eficazes no aumento do desempenho da arquitetura, principalmente os que levaram à eliminação/minimização das restrições causadas pela estrutura da arquitetura, como a replicação das unidades e *buffers* nos EPs.

Como pode ser constatado pelos resultados, a Versão 1.1 com 8 EPs possui algumas limitações, principalmente na questão da comunicação das unidades de armazenamento e de despacho com os EPs e no acesso à memória de *templates*. Estes fatores impediram a obtenção de um bom desempenho e foram alvo de estudo nas versões subseqüentes.

A Versão 1.2, com 16 EPs, apresenta um melhor desempenho do que a Versão 1.1. Uma das melhorias é a adoção de barramentos paralelos para a conexão das unidades de despacho e de armazenamento aos EPs. Outra medida adotada foi o particionamento da memória de *templates* em módulos paralelos facilitando, desta forma, o acesso simultâneo, tanto pela unidade de despacho quanto pela unidade de armazenamento.

A Versão 1.3 com 16 EPs é uma continuidade das versões anteriores. Porém foram incorporadas várias melhorias fruto dos testes e das análises realizadas nas outras versões. Algumas destas melhorias merecem destaque, como a utilização de *buffers* nos EPs, replicação das unidades de despacho e de armazenamento, aumento da quantidade de barramentos paralelos para comunicação com os EPs e incorporação da memória de dados. Além disto, a interface foi aprimorada com a utilização do processador NIOS, também incorporado em FPGA, o que facilita a troca de informações entre a arquitetura paralela e o *host* por meio de um canal de comunicação mais eficiente como a USB, pois nas outras versões utilizava-se a porta serial.

O quadro comparativo com as características de cada uma das implementações está apresentado na Tabela 14, em que se pode observar a evolução da arquitetura em termos de incorporação de recursos.

Tabela 14: Comparativo entre as implementações

Característica	Versão 1.1	Versão 1.2	Versão 1.3	Unidade
Número máximo de EPs	8	16	16	-
Freqüência de Operação	40	50	50	MHz
Complexidade do EP	ULA ponto fixo 8 *bits*	ULA ponto fixo 16 *bits*	ULA ponto fixo de 8/16 *bits*	-
Despacho	Seqüencial	Bufferizado	Paralelo/ Bufferizado	-
Armazenamento	Seqüencial	Bufferizado	Paralelo/ Bufferizado	-
Quantidade de Instruções	8	16	16	-
Tempo de *Clock*	25	20	20	*n*s
Memória de *Templates*	125	256	1K	*Templates*
Memória de Dados	-	-	1Kx16	*bits*

Os testes realizados envolveram o processamento de quatro algoritmos: algoritmo com instruções independentes, algoritmo para a solução de equação diferencial de 2ª ordem, algoritmo FIR de 5ª ordem e algoritmo IDEA (*International Data Encryption Algorithm)*. Estes testes aplicados nas versões da arquitetura mostraram que houve uma redução do tempo de processamento (em termos de ciclos de *clock*), em alguns casos menos do que em outros em virtude das dependências nas aplicações. Contudo, para efeito de estudo nas versões implementadas, limitou-se o número de EPs a 16. Outro ponto a ser destacado é que o processamento se ajustou automaticamente ao número de EPs presentes nas execuções, evitando desta forma a necessidade de modificação do código da aplicação.

Nos testes realizados obteve-se uma redução de aproximadamente 6 vezes no tempo de processamento com 16 EPs em comparação com a execução seqüencial. Isto denota que a abordagem adotada nesta arquitetura está em conformidade com o que se espera de uma máquina paralela, ou seja, um ganho crescente de desempenho com o incremento do número de EPs. Contudo, há limites para este ganho, sendo alguns ocasionados pela arquitetura e outros intrínsecos à aplicação, e que foram apresentados ao longo do trabalho.

O conjunto de instruções nas versões implementadas foi restrito a 16 instruções diferentes, entre as aritméticas, lógicas, comparação, relacionais e miscelânea. Estas instruções são genéricas o suficiente para resolver uma grande gama de problemas, em

especial os que envolvem cálculo numérico. Isto pode ser comprovado pelos testes realizados nas diversas versões da arquitetura.

Nas três implementações da arquitetura paralela reconfigurável constata-se que o tempo de processamento decai à medida que se adicionam processadores à arquitetura, o que é um comportamento desejado para uma arquitetura paralela. Isto é mais visível nas Versões 1.2 e 1.3, pela própria característica da arquitetura, pois as curvas do tempo de processamento real e ideal estão próximas, como mostrado nas Figuras 47 e 50 respectivamente.

Uma análise similar pode ser feita com relação ao *Speedup* nas versões implementadas. Percebe-se que o *Speedup* nas Versões 1.2 e 1.3 apresentaram resultados mais condizentes com o que se espera em uma abordagem de processamento paralelo. Os resultados em termos de *Speedup* confirmam o que estabelece a Lei de *Amdahl* e, desta forma, o *Speedup* máximo fica abaixo do *Speedup* ideal, representada por uma curva assintótica característica. Ainda sob este aspecto destaca-se que isto é ocasionado pelas dependências presentes na aplicação e que, na maioria dos casos, não podem ser resolvidos, sendo que algumas são decorrentes da forma como o programador expressa a lógica de programação.

Como última análise, na Tabela 15, é apresentado um quadro comparativo entre a arquitetura proposta Versão 1.3 e outras. As características da arquitetura proposta que mais divergem das demais são:

a) Não efetua o mapeamento das operações nos EPs;

b) Apresenta uma maior flexibilidade de processamento;

c) A interconexão entre os componentes é por meio de barramentos paralelos;

d) A programabilidade é completa garantindo uma maior versatilidade de operações;

e) Uma maior reusabilidade dos EPs durante o mesmo processamento sem a necessidade de reprogramação.

As outras características são importantes para a definição da arquitetura, entretanto não são fatores inequívocos para diferenciação da arquitetura proposta das demais.

Tabela 15: Quadro comparativo entre as arquiteturas

Característica	Arquitetura Proposta	XPP	WaveScalar	KressArray	Computador Funcional	HOSMII WASMII	COLT
Controle	Centralizado	Centralizado	Distribuído	Centralizado	Distribuído	Distribuído	Distribuído
Tipos de Dados	Ponto Fixo 16bits Ponto Flutuante 32bits	Ponto Fixo 16, 24 e 32bits	Ponto Fixo 32bits	Ponto Fixo 32bits	Ponto Fixo 8bits	Não Determi-nado	Ponto Flutuante 16bits
Elemento Processador (EP)	ULA	ULA	ULA	ULA	ULA	Configu-rável	ULA
Mapeamento das operações nos EPs	Não	Sim	Sim	Sim	Sim	Sim	Sim
Modelo *Dataflow*	Dinâmico	Estático	Dinâmico	Estático	Dinâmico	Estático	Dinâmico
Inteiramente em FPGA	Sim	Sim	Sim	Sim	Não	Sim	Sim
Reconfigurabi-lidade	Semi-estática	Semi-estática	Semi-estática	Semi-estática	Semi-estática	Dinâmica	Semi-estática
Número de EPs	16	20	~2K	24	1024	20	16
Tipo de Interconexão	Barramento	Malha	Malha	Malha	Malha	Malha	Malha
Flexibilidade	Sim	Não	Não	Não	Não	Não	Não
Programabili-dade	Completa	Não	Não	Não	Não	Não	Não
Reusabilidade	Sim	Não	Não	Não	Não	Não	Não

5.2 CONCLUSÕES

O objetivo central da computação paralela é conseguir juntar o melhor de dois mundos: um *hardware* de baixo custo à desejável facilidade de programação do computador paralelo (TANEMBAUM, 2001). Infelizmente, no processamento paralelo, a escalada do desempenho da aplicação, frequentemente, pode não coincidir com a velocidade que os recursos dão impressão de possibilitarem, e a programação para os computadores paralelos é pesada (HARTENSTEIN, 2004).

Em muitas aplicações com paralelismo no nível de *threads* (TLP), consegue-se apenas uma fraca melhoria no aumento do desempenho por processador adicionado à arquitetura. A *Lei de Amdahl* explica uma das diversas razões da ineficiente utilização dos recursos computacionais, que é a questão da exploração do paralelismo da aplicação em função de uma

parcela do código ser inerentemente seqüencial. Um dos problemas dominantes, citado em HARTENSTEIN (2001), é o tempo de busca das instruções no modelo *von Neumann*, problema este inexistente em arquitetura *dataflow*, em virtude da busca não depender de um contador de programa

Os sistemas de computação reconfiguráveis têm mostrado a possibilidade de acelerar enormemente a execução do programa, proporcionando uma alternativa de alto desempenho para soluções que antes eram exclusivamente em *software*. Contudo, isto pode ser amplificado com a adoção dos conceitos de processamento paralelo, como a exploração do paralelismo pela execução simultânea nos processadores da arquitetura paralela.

Uma arquitetura paralela implementada em FPGA é uma opção a ser considerada e possibilita uma série de vantagens, como uma rápida readequação do projeto para suportar outros tipos de aplicações, reaproveitamento de recursos. As características funcionais da arquitetura podem ser ampliadas, como aumentar o volume de processamento, com o acréscimo no número de EPs. O aumento da capacidade de processamento depende fortemente do dispositivo programável, como quantidade de elementos lógicos, capacidade de armazenamento e interconexão interna. Um outro aspecto é a exploração do paralelismo em vários níveis, tanto nas operações/instruções quanto nas unidades funcionais da arquitetura.

Apesar dos avanços dos dispositivos programáveis, ainda se tem uma limitação em termos de elementos lógicos e de memória que eles possuem. Como alternativa de implementação, poderiam ser utilizados vários dispositivos acoplados em conjunto para se conseguir uma capacidade maior do que individualmente, mas isto não encerra por completo esta questão.

Independentemente destes limites, os dispositivos lógicos programáveis são uma realidade no desenvolvimento de sistemas digitais. A adoção deles no desenvolvimento de arquiteturas de computadores, quer sejam paralelos ou não, trazem uma série de vantagens, como desenhar uma arquitetura customizável, menor espaço físico, minimização do custo e também um aumento da velocidade de operação. Contudo, uma característica dominante é a questão da reconfigurabilidade, pois pode-se modificar toda a arquitetura sem a alteração física dos componentes necessários.

A incorporação na arquitetura paralela reconfigurável do conceito de fluxo de dados, proporciona uma lógica de controle mais enxuta e com um menor consumo de recursos do dispositivo programável. O conceito de fluxo de dados é uma idéia que, apesar de não ser nova, apresenta algumas vantagens, como na questão do escalonamento das operações, que é

simples, reduzindo a lógica de controle. Neste caso, o controle da execução é realizado pela própria arquitetura seguindo o fluxo natural das operações.

Um programa fluxo de dados é composto por um conjunto de operações, denominados de *templates*. O *template* está atrelado a uma arquitetura, mas o formato é de fácil adequação a outras arquiteturas fluxo de dados, garantindo a portabilidade do programa, isto porque cada *template* corresponde a um nó do grafo de fluxo de dados. Logo, é necessário que haja somente uma reestruturação do formato do *template* para outra arquitetura destino.

Outro ponto a ser salientado é que, neste trabalho, não é necessário utilizar uma linguagem de programação específica voltada ao paralelismo, como exemplo a HPF (*High Performance Fortran*). Entretanto, é necessário que haja uma forma de expressar o paralelismo, como é o caso do grafo de fluxo de dados. Desta forma, o uso de compiladores paralelizadores para a extração do paralelismo é dispensável, pois apenas necessita-se de um *software* montador para a geração do programa, em virtude de se explorar o paralelismo implícito à aplicação. Contudo, a utilização de um dos métodos descritos em FERLIN (1997) pode resultar em uma exploração mais eficiente do paralelismo e, nesse caso, os métodos voltados para a granularidade fina, por exemplo Hiperplano, são mais apropriados.

Um ponto central disso tudo é aumentar o desempenho de um sistema computacional, de modo a reduzir significativamente o tempo de processamento das aplicações. Neste sentido, a avaliação do desempenho é um fator necessário em qualquer análise. Estão disponíveis diversas métricas de desempenho que podem fornecer informações a respeito de parâmetros para avaliação de computadores paralelos, levando-se em conta o tempo de processamento, unidades funcionais, instruções e outros. Contudo, não há uma métrica única que consiga fornecer valores absolutos para a comparação destes computadores, pois cada métrica aborda um determinado aspecto da arquitetura. Neste trabalho, adotaram-se algumas métricas para ajudar na análise das implementações das versões, como o CPI, *Speedup*, Tempo de Execução, MIPS e *Throughput*. Cada uma destas métricas tem sua devida importância, mas é a análise do conjunto delas que fornece informações mais claras sobre o desempenho de um computador.

Com o objetivo de para facilitar o interfaceamento entre a arquitetura e o *host* foi utilizado o processador NIOS também embarcado em FPGA. Todavia, este processador poderia ter outras atribuições que resultariam em um melhor aproveitamento da sua potencialidade, além de proporcionar uma expansão na capacidade de processamento. Isto seria alcançado com o NIOS sendo o controlador do *cluster* de várias arquiteturas,

possibilitando uma escalabilidade de recursos em função da necessidade demandada pela aplicação.

No desenvolvimento da arquitetura, utilizou-se a ferramenta QuartusII da Altera, e os componentes foram descritos em VHDL, que é uma linguagem de descrição *hardware* padrão. Isto facilita a migração desse projeto para outros ambientes de desenvolvimento e para outras famílias de dispositivos programáveis. O desenvolvimento foi modular, pois os diferentes blocos foram projetados separadamente para posterior integração. O ambiente de simulação permite a visualização de possíveis falhas de projeto, antes mesmo da implementação física da arquitetura. Como a arquitetura foi implementada integralmente em lógica programável, consegue-se reconfigurá-la inteiramente ou alguns módulos para atender aos novos requisitos de processamento.

Apesar desta arquitetura ainda apresentar aspectos que podem ser melhorados, os resultados obtidos são significativos.

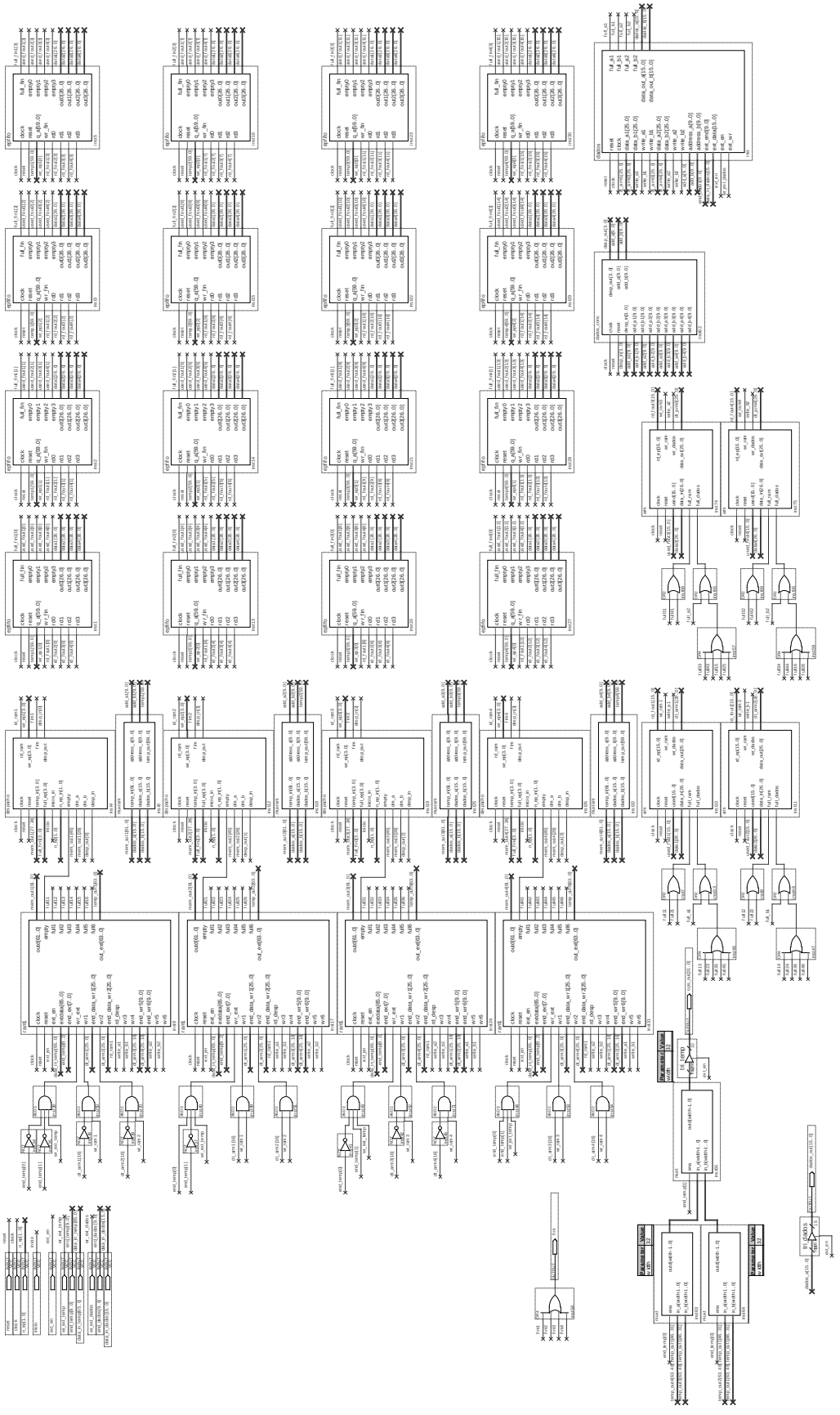
5.3 TRABALHOS FUTUROS

Em virtude desta arquitetura ainda poder ser aprimorada, alguns aspectos merecem atenção especial para as futuras melhorias, como:

- aprimoramento das conexões para aumentar o *Speedup* e reduzir os tempos consumidos pelas unidades da arquitetura, principalmente o acesso aos EPs;
- aumentar a complexidade do EP para realizar operações mais complexas do que as realizadas atualmente;
- ampliar o tamanho das memórias para permitir o processamento de aplicações com maior quantidade de operações;
- aumentar a quantidade de operações/instruções da arquitetura para abranger um número maior de aplicações;
- aumentar o número de EPs possibilitando um maior grau de paralelismo do que é conseguido com a configuração atual;
- adaptar a arquitetura para possibilitar a reconfiguração dinâmica, em tempo de execução, pois atualmente só é permitida a reconfiguração semi-estática, realizada em tempo de compilação.

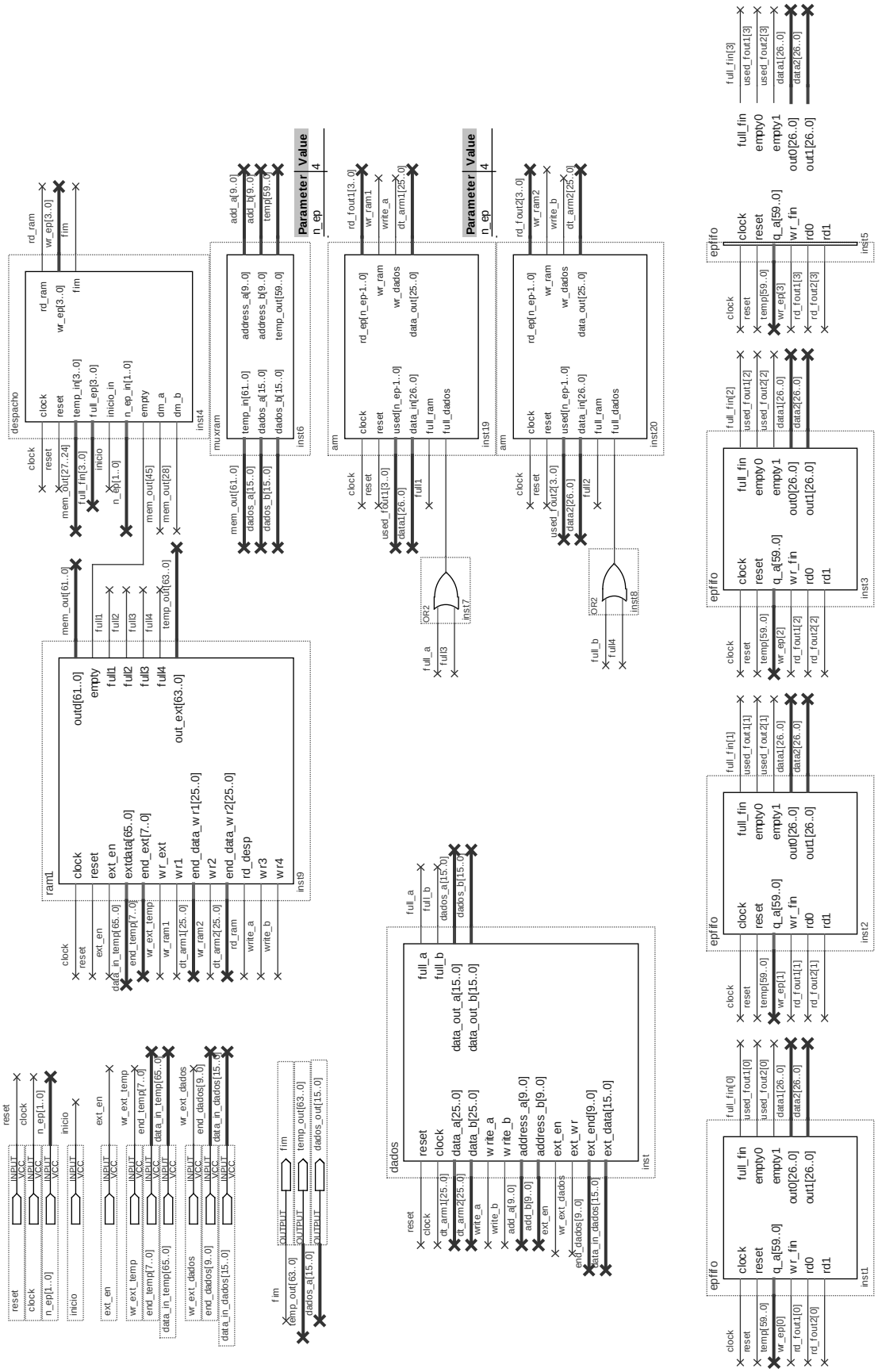
ANEXO 1

DIAGRAMA EM BLOCOS DA ARQUITETURA COM 16 EPs (MEMÓRIA DIVIDIDA)



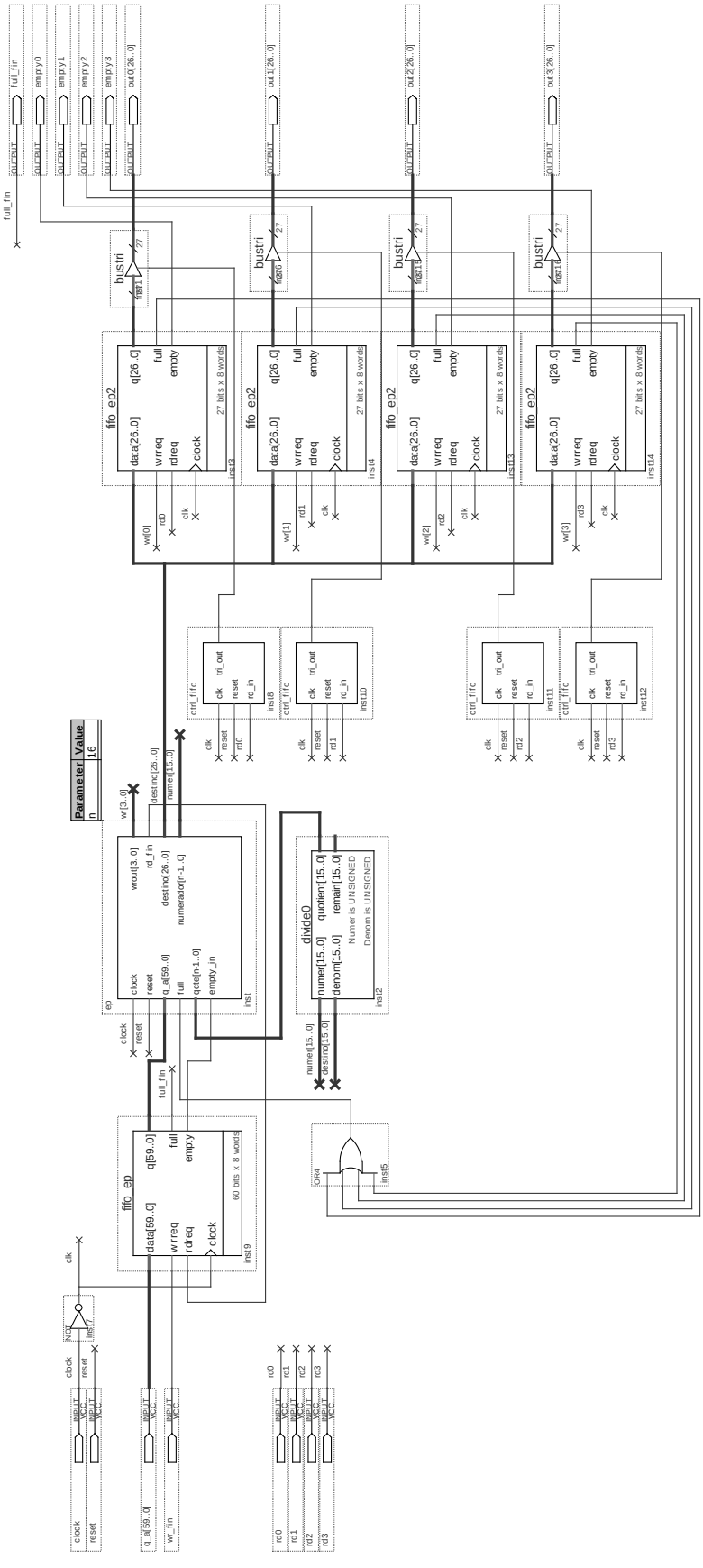
ANEXO 2

DIAGRAMA EM BLOCOS DA ARQUITETURA COM 4 EPs (MEMÓRIA ÚNICA)



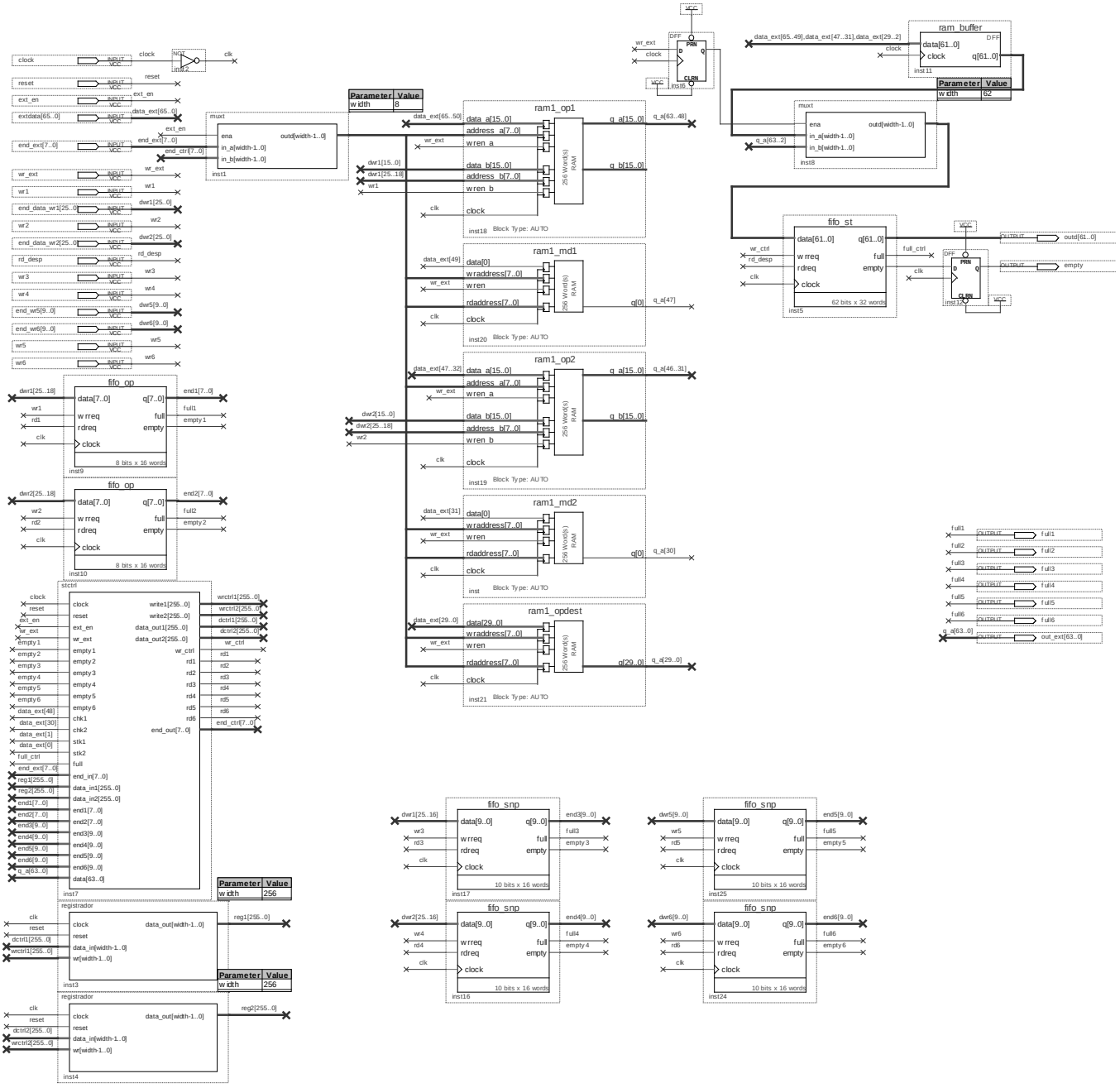
ANEXO 3

DIAGRAMA EM BLOCOS DO ELEMENTO PROCESSADOR (EP)



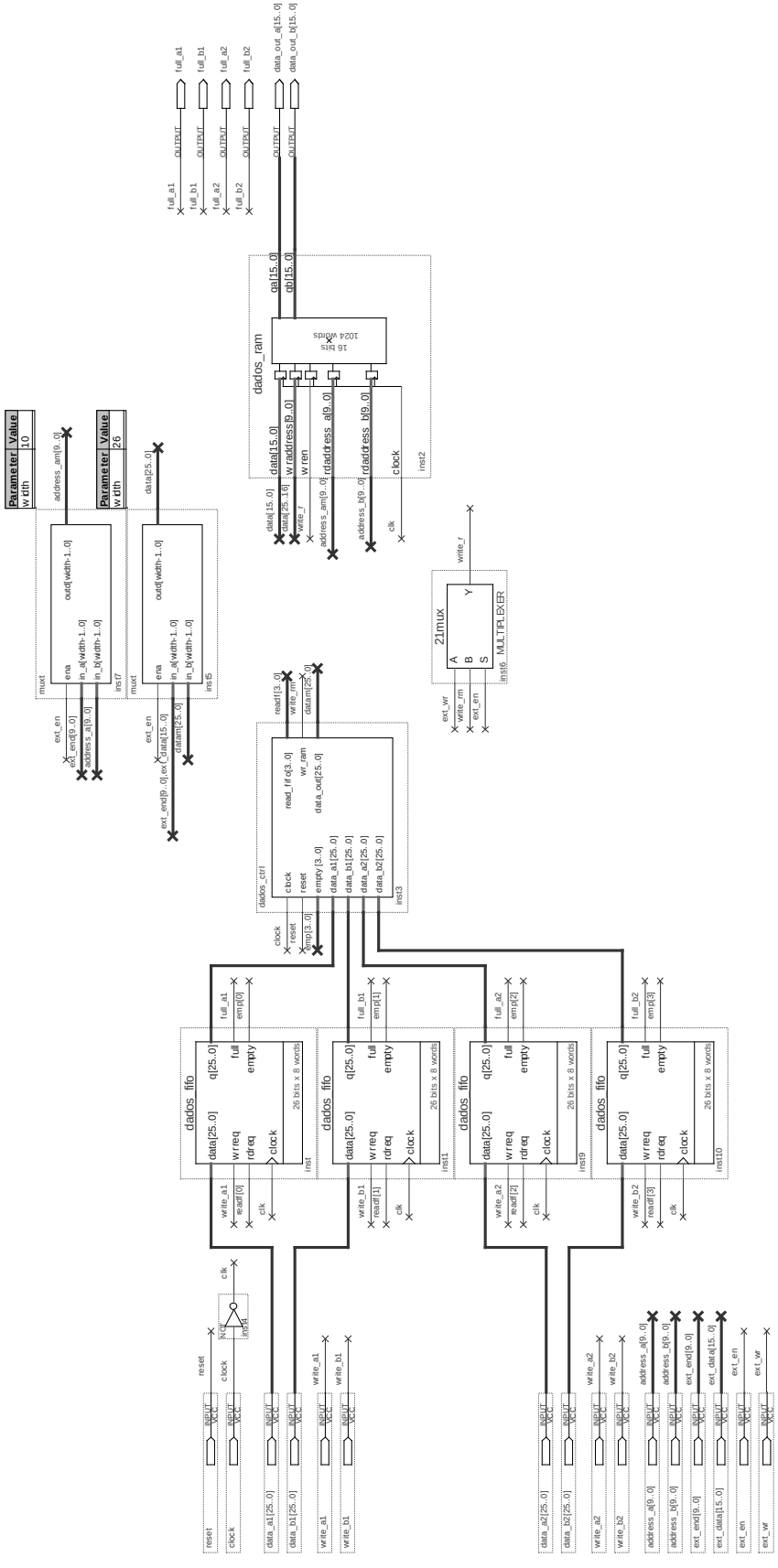
ANEXO 4

DIAGRAMA EM BLOCOS DA MEMÓRIA DE *TEMPLATES* (MT)



ANEXO 5

DIAGRAMA EM BLOCOS DA MEMÓRIA DE DADOS (MD)



ANEXO 6

LISTAGEM DO CÓDIGO VHDL DA UNIDADE DE DESPACHO (UD)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity despacho is
	port
	(
		clock, reset		: in	std_logic;
		temp_in			: in	std_logic_vector(3 downto 0);
		full_ep			: in	std_logic_vector(3 downto 0);
		inicio_in		: in	std_logic;
		n_ep_in			: in	unsigned(1 downto 0);
		empty			: in	std_logic;
		dm_a			: in	std_logic;
		dm_b			: in	std_logic;
		desp_in			: in	std_logic;
		rd_ram			: out	std_logic;
		wr_ep			: out	std_logic_vector(3 downto 0);
		fim			: out	std_logic;
		desp_out		: out	std_logic
	);
end despacho;

architecture despacho of despacho is

begin

	process (clock, reset, inicio_in)
		variable cont_ep		: integer range 0 to 3;
		variable n_ep			: unsigned(1 downto 0);
		type estados is (inicio, dados, memoria, stop);
		variable estado : estados;
	begin
		if reset = '1' then
			n_ep := n_ep_in;
			cont_ep := 0;
			wr_ep <= conv_std_logic_vector(0,4);
			fim <= '0';
			rd_ram <= '0';
			desp_out <= '0';
			estado := inicio;
		elsif clock'event and clock = '1' then
		case estado is
			when inicio =>
				if inicio_in = '1' then
					estado := dados;
					rd_ram <= '1';
				end if;
			when dados =>
				if empty = '0' then
					if temp_in = x"F" then -- Se instrucao STOP
						estado := stop;
					else
						if dm_a = '1' or dm_b = '1' then
							rd_ram <= '0';
							wr_ep <= "0000";
							desp_out <= '1';
							estado := memoria;
						else
							if full_ep(cont_ep) = '0' then -- se fila nao cheia
								for i in 0 to 3 loop -- write em apenas 1 EP
									if i = cont_ep then
										wr_ep(i)<='1';
									else
										wr_ep(i)<='0';
									end if;
								end loop;
								if cont_ep = n_ep then
```

```
                                                                                cont_ep := 0; -- incremento de cont_ep
                                                                        else
                                                                                cont_ep := cont_ep + 1;
                                                                        end if;
                                                                        rd_ram <= '1';

                                                                else
                                                                        rd_ram <= '0';
                                                                        wr_ep <= "0000";
                                                                end if;
                                                        end if;
                                                end if;
                                        else
                                                wr_ep <= "0000";
                                        end if;

                                when memoria =>
                                        if full_ep(cont_ep) = '0' then -- se fila nao cheia
                                                if desp_in = '1' then
                                                        desp_out <= '0';
                                                        for i in 0 to 3 loop -- write em apenas 1 EP
                                                                if i = cont_ep then
                                                                        wr_ep(i)<='1';
                                                                else
                                                                        wr_ep(i)<='0';
                                                                end if;
                                                        end loop;
                                                        if cont_ep = n_ep then
                                                                cont_ep := 0; -- incremento de cont_ep
                                                        else
                                                                cont_ep := cont_ep + 1;
                                                        end if;
                                                        rd_ram <= '1';
                                                        estado := dados;
                                                end if;

                                        end if;

                                when stop =>
                                        fim <= '1';
                                        rd_ram <= '0';
                                        wr_ep <= "0000";
                        end case;
                        end if;
                end process;
end despacho;
```

ANEXO 7

LISTAGEM DO CÓDIGO VHDL DA UNIDADE DE ARMAZENAMENTO (UA)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity arm is
port
(
	clock, reset		:	in std_logic;
	used			:	in std_logic_vector(15 downto 0);
	data_in			:	in std_logic_vector(26 downto 0);
	full_ram		:	in std_logic;
	full_dados		:	in std_logic;
	rd_ep			:	out std_logic_vector(15 downto 0);
	wr_ram			:	out std_logic;
	wr_dados		:	out std_logic;
	data_out		:	out std_logic_vector(25 downto 0)
);
end arm;

architecture arm of arm is
	signal cont_ep	:	integer range 0 to 15;

begin
	process(clock, reset)
		type estados is (pronto, gravando);
		variable estado : estados;
	begin
		if reset = '1' then
			wr_ram <= '0';
			wr_dados <= '0';
			cont_ep <= 0;
			rd_ep <= conv_std_logic_vector(0,16);
			estado := pronto;

		elsif clock'EVENT and clock = '1' then
			case estado is
				when pronto =>
					wr_ram <= '0';
					wr_dados <= '0';
					if used(cont_ep) = '0' then
						for i in 0 to 15 loop
							if i = cont_ep then
								rd_ep(i) <= '1';
							else
								rd_ep(i) <= '0';
							end if;
						end loop;
						estado := gravando;
					else
						rd_ep <= conv_std_logic_vector(0,16);
						for i in 0 to 15 loop
							if used(i) = '0' then
								cont_ep <= i;
							end if;
						end loop;
					end if;
				when gravando =>
					rd_ep <= conv_std_logic_vector(0,16);
					data_out <= data_in(25 downto 0);
					if data_in(26) = '0' then
						if full_ram = '0' then
							wr_ram <= '1';
							wr_dados <= '0';
							estado := pronto;
						end if;
					else
						if full_dados = '0' then
							wr_ram <= '0';
```

```
                                                    wr_dados <= '1';
                                                    estado := pronto;
                                                end if;
                                            end if;

                                end case;
                        end if;
                end process;
end arm;
```

ANEXO 8

LISTAGEM DO CÓDIGO VHDL DA UNIDADE LÓGICA-ARITMÉTICA (ULA)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

--Operações

-- Aritméticas
-- 0000 - Soma
-- 0001 - Subtração
-- 0010 - Multiplicação
-- 0011 - Divisão

-- Lógicas
-- 0100 - and
-- 0101 - or
-- 0110 - not
-- 0111 - xor
-- 1000 - Nor

-- Relacionais
-- 1001 - ==
-- 1010 - !=
-- 1011 - >=
-- 1100 - <
-- 1101 - Condicional

-- 1110 - Duplica
-- 1111 - Stop (afeta apenas unidade de controle)

entity ep is
generic (   n           : integer:= 16          );
port
(
        clock, reset        :       in      std_logic;
        q_a                 :       in      unsigned ( 59 downto 0 );
        full                :       in      std_logic;
        qcte                :       in      std_logic_vector(n-1 downto 0);
        empty_in            :       in      std_logic;
        wrout               :       out     std_logic_vector(3 downto 0);
        rd_fin              :       out     std_logic;
        destino             :       out     std_logic_vector(26 downto 0); --16bits de dados, 10bits de endereco e 1bit DM
        numerador           :       out     std_logic_vector(n-1 downto 0)
);
end ep;

architecture ep of ep is
begin
        process (clock, reset)
                variable enddest1, enddest2     :       unsigned (9 downto 0); -- endereco destino 1 e 2
                variable pdest1, pdest2         :       std_logic; -- port destino 1 e 2
                variable mdest1, mdest2         :       std_logic; -- bit para local de armazenamento
                variable dest                   :       std_logic_vector (n-1 downto 0); -- valores a serem gravados no destino
                variable temp                   :       std_logic_vector (n-1 downto 0); -- registro temporario da saida
                variable op1, op2               :       unsigned (n-1 downto 0);
                variable operacao               :       unsigned (3 downto 0);
                type STATE_TYPE is (inicio, decod, calculo, saida1, saida2, divisao);
                variable state: STATE_TYPE;

        begin
                if reset = '1' then
                        rd_fin <= '0';
                        wrout <= "0000";
                        dest := x"0000";
                        temp := x"FFFF";
                        state := inicio;

                elsif clock'EVENT and clock = '1' then
```

```
case state is
	when inicio =>
		wrout <= "0000";
		if empty_in = '0' then
			rd_fin <= '1';
			state := decod;
		else
			rd_fin <= '0';
			state := inicio;
		end if;

	when decod =>
		temp := x"FFFF";
		wrout <= "0000";
		rd_fin <= '0';
		enddest1 := q_a(23 downto 14);
		enddest2 := q_a(11 downto 2);
		pdest1 := q_a(13);
		pdest2 := q_a(1);
		mdest1 := q_a(12);
		mdest2 := q_a(0);
		if (q_a(27 downto 24) = 3) then
			numerador <= conv_std_logic_vector(q_a(59 downto 44),16);
			destino(15 downto 0) <= conv_std_logic_vector(q_a(43 downto 28),16);
			state := divisao;
		else
			operacao := q_a(27 downto 24);
			op1 := q_a(59 downto 44);
			op2 := q_a(43 downto 28);
			state := calculo;
		end if;

	when calculo =>
		case operacao is
			-- Aritméticas
			when x"0" => -- SOMA
				dest := conv_std_logic_vector(op1 + op2,n);
			when x"1" => -- SUBTRAÇÃO
				dest := conv_std_logic_vector(op1 - op2,n);
			when x"2" => -- MULTIPLICAÇÃO
				dest :=conv_std_logic_vector(op1 * op2,n);

			-- Lógicas
			when x"4" => -- AND
				for i in 0 to n-1 loop
					dest(i):= op1(i) and op2(i);
				end loop;
			when x"5" => -- OR
				for i in 0 to n-1 loop
					dest(i):= op1(i) or op2(i);
				end loop;
			when x"6" => -- NOT
				for i in 0 to n-1 loop
					dest(i):= not op1(i);
				end loop;
			when x"7" => -- XOR
				for i in 0 to n-1 loop
					dest(i):= op1(i) xor op2(i);

				end loop;
			when x"8" => -- NOR
				for i in 0 to n-1 loop
					dest(i):= op1(i) nor op2(i);

				end loop;

			-- Relacionais
			when x"9" => -- IGUAL ==
				if op1 = op2 then
					dest := conv_std_logic_vector(1,n);
				else
					dest := conv_std_logic_vector(0,n);

				end if;
			when x"A" => -- DIFERENTE !=
				if op1 /= op2 then
					dest := conv_std_logic_vector(1,n);
```

```
                                        else
                                                dest := conv_std_logic_vector(0,n);

                                        end if;
                                when x"B" => -- MAIOR IGUAL >=
                                        if op1 >= op2 then
                                                dest := conv_std_logic_vector(1,n);
                                        else
                                                dest := conv_std_logic_vector(0,n);

                                        end if;
                                when x"C" => -- MENOR <
                                        if op1 < op2 then
                                                dest := conv_std_logic_vector(1,n);
                                        else
                                                dest := conv_std_logic_vector(0,n);

                                        end if;

                                when x"D" => -- CONDICIONAL
                                        dest := conv_std_logic_vector(op2,n);

                                        if conv_std_logic_vector(op1,n) = conv_std_logic_vector(1,n) then
                                                enddest1 := conv_unsigned(0,10);
                                        else
                                                enddest2 := conv_unsigned(0,10);
                                        end if;

                                when x"E" => -- DUPLICA
                                        dest := conv_std_logic_vector(op1,n);

                                when others =>
                                        dest := conv_std_logic_vector(0,n);
                        end case;
                        if temp = dest then
                                state := saida1;
                        else
                                temp := dest;
                        end if;

                when saida1 =>
                        if full = '1' then
                                state := saida1;
                        else
                                if enddest1 /= conv_unsigned(0,10) then
                                        destino(26) <= mdest1;
                                        destino(25 downto 16) <= conv_std_logic_vector(enddest1,10);
                                        destino(15 downto 0) <= dest;
                                        if enddest1(1) = '0' then
                                                if pdest1 = '0' then
                                                        wrout <= "0001";
                                                else
                                                        wrout <= "0010";
                                                end if;
                                        else
                                                if pdest1 = '0' then
                                                        wrout <= "0100";
                                                else
                                                        wrout <= "1000";
                                                end if;
                                        end if;
                                end if;
                                if enddest2 /= conv_unsigned(0,10) then
                                        state := saida2;
                                else
                                        if empty_in = '0' then
                                                rd_fin <= '1';
                                                state := decod;
                                        else
                                                state := inicio;
                                        end if;
                                end if;
                        end if;

                when saida2 =>
                        if pdest1 = pdest2 and full = '1' then
                                state := saida2;
```

```
                                        else
                                                destino(26) <= mdest2;
                                                destino(25 downto 16) <= conv_std_logic_vector(enddest2,10);
                                                destino(15 downto 0) <= dest;
                                                if enddest2(1) = '0' then
                                                        if pdest2 = '0' then
                                                                wrout <= "0001";
                                                        else
                                                                wrout <= "0010";
                                                        end if;
                                                else
                                                        if pdest2 = '0' then
                                                                wrout <= "0100";
                                                        else
                                                                wrout <= "1000";
                                                        end if;
                                                end if;
                                                if empty_in = '0' then
                                                        rd_fin <= '1';
                                                        state := decod;
                                                else
                                                        state := inicio;
                                                end if;
                                        end if;

                                when divisao =>
                                        dest := qcte;
                                        if dest = temp then
                                                state := saida1;
                                        else
                                                temp := dest;
                                                state := divisao;
                                        end if;

                        end case;
                end if;
        end process;
end ep;
```

REFERÊNCIAS BIBLIOGRÁFICAS

Accelchip. Disponível em: http://www.accelchip.com. Acessado em agosto 2008.

Adaptor version 3.1/3.2. Disponível em: http://www.liv.ac.uk/HPC/CompilerBenchmarks/node20.html. Acessado em agosto 2008.

ADÁRIO, A.M.S., ROEHE, E.L., BAMPI, S. Dynamically reconfigurable architecture for image processor applications**.** In: **Proceedings of Design Automation Conference**. New York, NY, USA: ACM, p. 623-628, 1999.

Adelante Technologies Inc. Disponível em: http://www.adelantetech.com. Acessado em junho 2008.

AGGARWAL, N., RANGANATHAN, P., JOUPPI, N.P., SMITH, J.E. Isolation in commodity multicore processors. **IEEE Computer**, v. 40, n. 6, p. 49-59, june, 2007.

Altera Corporation. Disponível em: http://www.altera.com. Acessado em agosto 2008.

ANDREWS, D., NIEHAUS, D., ASHENDEN, P. Programming models for hybrid CPU/FPGA chips. **IEEE Computer**, v. 37, n. 1, p. 118-120, 2004.

ARVIND, NIKHIL, R.S. Executing a Program on the MIT togged-token dataflow architecture. **IEEE Transactions on Computer**, v. 34, n. 3, p. 300-318, march, 1990.

BARAK, A. What´s mosix?. Disponível em: http://www.mosix.com/tit_whatis.html. Acessado em agosto 2008.

BECKER, J. **A Partitioning Compiler for Computers with Xputer-based Accelerators**. Ph. D. Thesis. Kaiserslautern University, Computer Science Department, 1997.

BECKER, J., PIONTECK, T., GLESNER, M. An application-tailored dynamically reconfigurable hardware architecture for digital baseband processing. In: **Proceedings of 13th Symposium on Integrated Circuits and Systems Design**, Manaus, AM, p. 342-346, september, 2000.

BECKER, J.; HARTENSTEIN, R.; HERZ, M.; NAGELDINGER, U. Parallelization in co-compilation for configurable accelerators. In: **Proceedings of Asia and South Pacific Design Automation Conference**. Yokohama, Japan, p. 23-33, february 10-13, 1998.

BELL, G., GRAY, J. What´s next in high-performance computing? **Communications of the ACM**, v. 25, p. 91-95, february, 2002.

BELL, G., GRAY, J., SZALAY, A. Petascale computational systems. **IEEE Computer**, v. 39, n. 1, p. 110-112, january, 2006.

BERGER, A.S. **Hardware and Computer Organization – the Software Perspective**. Oxford, UK: Newnes, 2005.

BITTNER Jr, R.A., ATHANAS, P.M., MUSGROVE, M.D. Colt: an experiment in wormhole run-time reconfiguration. In: **Proceedings of SPIE Photonics East**, Boston, MA, USA, p. 187-195, november, 1996.

BOUWENS, F., BEREKOVIC, M., KANSTEIN, A., GAYDADJIEV, G. Architectural exploration of the ADRES coarse-grained reconfigurable array. **Lecture Notes in Computer Science**, v. 4419, p. 1-12, 2007.

BROOKSHEAR, J.G. **Ciência da Computação: uma visão abrangente**, 7a edição. Porto Alegre, RS: Bookman, 2005.

BUDIO, M., ARTIGAS, P.V., GOLDSTEIN, S.C. Dataflow: a complement to superscalar. In: **Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software**, Austin, TX, USA, p. 102-111, march 20-22, 2005.

BUELL, D., EL-GHAZAWI, T., GAJ, K., KINDRATENKO, V. High-performance reconfigurable computing. **IEEE Computer**, v. 40, n. 3, p. 23-27, march, 2007.

CARDOSO, J.M.P. Self loops and reconfigurable dataflow arrays. In: **Proceedings of International Workshop on Systems, Architecture, Modeling and Simulation. Lecture Notes in Computer Science,** v. 3133, Samos, Greece, p. 234-243, july, 2004.

CARDOSO, J.M.P., VÉSTIAS, M.P. Architectures and compilers to support reconfigurable computing. **ACM Crossroads Student Magazine,** v. 5-3, p. 15-22, july, 1999.

CARLSTROM, J., BODÉN, T. Synchronous dataflow architecture for network processors. **IEEE Micro**, v. 24, n. 5, p. 10-18, september-october, 2004.

CARTER, N. **Arquitetura de Computadores**. Porto Alegre, RS: Bookman, 2003.

CASAVANT, T.L., TURDIK, P., PLASIK, F. (Eds). **Parallel Computer Theory and Practice.** Los Alamitos, CA, USA: IEEE Computer Society Press, 1996.

CAVIN, R., HUTCHBY, J.A., ZHIRNOV, V., BREWER. J.E., BOURIANOFF, G. Emerging research architectures. **IEEE Computer**, v. 41, n. 5, p. 33-37, may, 2008.

Celoxica Inc - technology backgrounder. Disponível em: http://www.celoxica.com/press/presskit/default.asp. Acessado em agosto 2008.

CENTODUCATTE, P.C. Arquiteturas avançadas de computadores: uma introdução. In: **Anais da I Jornada de Estudos en Computação de Piracicaba e Região**, Piracicaba,SP, p. 135-153, setembro, 2000.

CHAPARO, P., GONZÁLEZ, J., MAGKLIS, G., CAI, Q., GONZÁLEZ, A. Understanding the thermal implications of multicore architectures. **IEEE Transactions on Parallel and Distributed Systems**, v. 18, n. 8, p. 1055-1065, august, 2007.

CHARBOUILLOT, S., PÉREZ, A., FRONTE, D. A programmable hardware cellular automaton: example of data flow transformation. **VLSI Design**, v. 2008, article id160728, p. 1-7, 2008.

CHERBAKA, M.F. **Verification and Configuration of a Run-Time Reconfigurable Custom Computing Integrated Circuit for DSP Applications.** M.Sc Thesis. Virginia Polytechinic Institute and State University, 1996.

COMPTON, K. **Architecture Generation of Customized Reconfigurable Hardware**. Ph.D Thesis. Nortwestern University, ECE Department, 2003.

COMPTON, K., HAUCK, S. Reconfigurable computing: a survey of systems and software. **ACM Computing Survey**, v. 34, n. 2, p. 171-210, june, 2002.

DEHON, A., WAWRZYNEK, J. Reconfigurable computing: what, why and implications for design automation. In: **Proceedings of Design Automation Conference**, New York, NY, USA: ACM, p. 610-615, 1999.

Design Suiet, DK. Disponível em: http://www.celoxica.com/products/dk/default.asp. Acessado em agosto 2008.

DINIZ, P., HALL, M., PARK, J., SO, B., ZIEGLER, H. Automatic mapping of C for FPGAs with the DEFACTO compilation and synthesis system. **Microprocessor and Microsystems**, v. 29, p. 51–62, 2005.

DONG, Y., DOU, Y., ZHOU, J. Optimized Generation of Memory Structure in Compiling Window Operations onto Reconfigurable Hardware. **Lecture Notes in Computer Science – Reconfigurable Computing: Architectures, Tools and Applications**, v. 4419, Berlin-Heidelberg: Springer-Verlag, p. 110 – 121, 2007.

DONGARRA, J., FOSTER, I., FOX, G., GROPP, W., KENNEDY, K., TORCZON, L., WHITE, A. **Sourcebook of Parallel Computing**. San Francisco, CA, USA: Morgan & Kaufmann, 2003.

DUTT, N., CHOI, K. Configurable processors for embedded computing. **IEEE Computer**, v.36, n. 1, p. 120-123, january, 2003.

EKPANYAPONG, M., HEALY, M., LIM, S.K. Placement for configurable dataflow architecture. In: **Proceedings of Asia and South Pacific Design Automation Conference**, Shanghai, China, v. 2, p. 1127-1130, january 18-21, 2005.

EL-REWINI, H., ABD-EL-BARR, M. **Advanced Computer Architecture and Parallel Processing**. New York, NY, USA: John Wiley & Sons, 2005.

FERLIN, E.P. **Avaliação de Métodos de Paralelização Automática**. Dissertação (Mestrado em Física Computacional). Universidade de São Paulo, Instituto de Física de São Carlos, 1997.

FERLIN, E.P. O avanço tecnológico dos processadores e sua utilização pelo software. **Revista da Vinci**, v. 1, n. 1, p. 43-60, 2004.

FERLIN, E.P. O processo de paralelização automática de programas sequenciais em programas paralelos. In: **Anais do IV Simpósio de Pesquisa e Extensão em Tecnologia**, Natal, RN, p. 188-190, novembro, 1998.

FERLIN, E.P., LOPES, H.S., LIMA, C.R.E., CICHACZEWSKI, E. Reconfigurable parallel architecture for genetic algorithms: application to the synthesis of digital circuits. **Lecture Notes in Computer Science**, v. 4419, p. 326-336, 2007.

FERLIN, E.P., LOPES, H.S., LIMA, L.C., JORY, C., SILVA JÚNIOR, P.G. Arquitetura paralela reconfigurável de alto desempenho aplicada à métodos numéricos. In: **Proceedings of 4th International Information and Telecommunication Technologies Symposium**, Florianópolis, SC, CDROM, dezembro, 2005.

Fluent bechmark. Disponível em: http://www.fluent.com/software/fluent/fl5bench/flbench_6.1/index.htm. Acessado em junho 2008.

Fluente – software – fluent benchmarks – small class transionic flow though a rotor. Disponível em: http://www.fluent.com/software/fluent/fl5bench/flbench_6.2.x/problems/fl5s3.htm. Acessado em junho 2008.

Fluente news – spring 98 – parallel processing on NT – another first. Disponível em: http://www.fluent.com/about/news/newsletters/98v7i1/a11.htm. Acessado em junho 2008.

GALANIS, M.D., DIMITROULAKOS, G., GOUTIS, C.E. Speedups and energy reductions from mapping DSP applications on an enbededd reconfigurable system. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, v. 15, n. 12, p. 1362-1366, december, 2007.

GHAZAWI, T.E., ARABY, E.E., HUANG, M., GAJ. K., KINDRATENKO, V., BUELL, D. The promise of high-performance reconfigurable computing. **IEEE Computer**, v. 41, n. 2, p. 69-76, february, 2008.

GOLDSTEIN, S.C., SCHMIT, H., BUDIU, M., CADAMBI. S., MOE, M., TAYLOR, R.R. Piperench: a reconfigurable architecture and compiler. **IEEE Computer**, v. 33, n. 4, p. 70-77, april, 2000.

GOODACRE, J., SLOSS, A.N. Parallelism and the ARM instructions set architecture. **IEEE Computer**, v. 38, n. 7, p. 42-50, july, 2005.

GRAY, J., BELL, G., SZALAY, A. Petascale computational systems. **IEEE Computer**, v. 39, n. 1, p. 110-112, 2006.

GURD, J. R., KIRKHAM, C. C., WATSON, I. The manchester prototype dataflow computer. **Communications of the ACM**, v. 28, n. 1, p. 34-52, january, 1985.

HALL, M.W., HARVEY, T., KENNEDY, K., MCINTOSH, N., MCKINLEY, K.S., OLDHAM, J. D., PALECZNY, M., ROTH, G. Experiences using the ParaScope editor: an interactive parallel programming tool. In: **Proceedings of Symposium on Principles and Practice of Parallel Programming**, San Diego, CA, USA, p. 33-43, may, 1993.

HAMMOND, L., NAYFEH, B.A., OLUKOTUN, K. A single-chip multiprocessor. **IEEE Computer**, v. 30, n. 9, p. 79-85, september, 1997.

HARDNETT, C.R., JAYARAJ, A., PALEM, K.V., YALAMANCHILI, S. Compiling stream kernels for polymorphous computing architectures. In: **Proceedings of 12th International Conference on Parallel Architectures and Compilation Tecniques**, New Orleans, LA, USA, CDROM, september 27 – October 1, 2003.

HARTENSTEIN, R. A decade of reconfigurable computing: a visionary retrospective. In: **Proceedings of Conference on Design, Automation and Test in Europe**. New York, NY, USA: IEEE Press, p. 642-649, 2001.

HARTENSTEIN, R. The digital divide of computing. In: **Proceedings of ACM International Conference on Computing**, p. 357-362, 2004.

HARTENSTEIN, R., KRESS, R. A datapath synthesis system for the reconfigurable datapath architecture. In: **Proceedings of Asia and South Pacific Design Automation Conference**, Chiba, Japan, p. 479 - 484, august 29 – september 1, 1995.

HARTENSTEIN, R., KRESS, R., REINING, H. A reconfigurable data-driven ALU for Xputers. In: **IEEE Workshop on FPGAs for Custom Computing Machines**, Napa, USA, CDROM, april, 1994.

HENNESSY, J.L., PATTERSON, D.A. **Computer Architecture: a quantitative approach**, 4th edition. San Francisco, CA, USA: Morgan & Kaufmann, 2006.

HENNING, J.L. SPEC CPU2000: measuring CPU performance in the new millennium. **IEEE Computer**, v. 33, n. 7, p. 28-35, july, 2000.

HERBORDT, M.C., VANCOURT, T., GU, Y., SUKHWANI, B., CONTI, A., MODEL, J., DISABELLO, D. Achieving high performance with FPGA-based computing. **IEEE Computer**, v. 40, n. 3, p. 50-57, march, 2007.

HONG, B., PRASANNA, V. Adaptive allocation of independent tasks to maximize throughput. **IEEE Transactions on Parallel and Distributed Systems**, v. 18, n. 10, p. 1377-1392, october, 2007.

HUTCHINGS, B., BELLOWS, P., HAWKINS, J., HEMMERT, S., NELSON, B., RYTTING, M. A CAD suite for high-performance FPGA design. In: **Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines**, p. 12-24, april, 1999.

JHDL FPGA CAD Tools - Brigham Young University. Disponível em: http://www.jhdl.org. Acessado em maio 2008.

JIANG, Y.C., WANG, J.F. Temporal partitioning data flow grapas for dynamically reconfigurable computing. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, v. 15, n. 12, p. 1351-1361, december, 2007.

JIMENEZ, M., LLABERIA, J.M., FERNANDEZ, A. Register tiling in nonrectangular iteration spaces. **ACM Transactions on Programming Languages and Systems**, v. 24, n. 4, p. 409-453, july, 2002.

KATZ, D.S., SOME, R.R. NASA advances robotic space exploration. **IEEE Computer**, v. 36, n. 1, p. 52-62, january, 2003.

KAVI, K.M., GIORGI, R., ARUL, J. Scheduled dataflow: execution paradigm, architecture, and performance evaluation. **IEEE Transactions on Computers**, v. 50, n. 8, p. 834-846, august, 2001.

LALL, N., CIGAN, E. Plug and play design methodologies for FPGA – based signal processing. **FPGA and Programmable Logic Journal**, Techfocus Media, March 8, 2005.

LE, T.T., HUU, T.C. Advances in parallel computing for the year 2000 and beyond. In: **Proceedings of Vietnamese Technical International Conference**, San Jose State University, CDROM, july 17-19, 1997.

LILJA, D.J. Exploiting the parallelism available in loops. **IEEE Computer**, v. 27, n. 2, p. 13-26, february, 1994.

LIMA, F.G., GUNTZEL, J.L.A. Componentes Programáveis. In: **Anais da 2ª Escola de Microeletrônica da SBC-Sul**, Torres, RS, p. 71-95, julho, 2000.

LIU, Y., FURBER, S. A low power embedded dataflow coprocessor. In: **Proceedings of IEEE Computer Society Annual Symposium on New Frontiers in VLSI Design**, p. 246-247, may 11-12, 2005.

LODI, A., TOMA, M., CAMPI, F., CAPPELLI, A., CANEGALLO, R., GUERRIERI, R. A VLIW processor with reconfigurable instruction set for embedded applications. **IEEE Journal of Solid-State Circuits**, v. 38, n. 11, p. 1876-1886, 2003.

MARTINS, C.A.P.S., ORDONEZ, E.D.M., CORRÊA, J.B.T., CARVALHO, M.B. Computação reconfigurável: conceitos, tendências e aplicações. In: **Anais do XXII Jornadas de Atualização em Informática**, SBC, p. 339-388, 2003.

MERKEY, P. Beowulf – introduction & overview. Disponível em: www.beowulf.org/intro.html. Acessado em junho 2008

MOORE, N., CONTI, A., LEESER, M., KING, L.S. Vforce: an extensible framework for reconfigurable supercomputing. **IEEE Computer**, v. 40, n. 3, p. 39-49, march, 2007.

Motorola Inc. **PowerPC 601 RISC Microprocessor User's Manual**. Phoenix, AR, USA: Motorola, 1993.

MPI - message passing interface. Disponível em: http://www-linux.mcs.anl.gov/mpi/. Acessado em junho 2008.

MURALI, S., ATIENZA, D., MELONI, O., CARTA, S., BENINI, L., MICHELI, G. DE., RAFFO, L. Synthesis of predictable networks-on-chip-based interconnect architectures for chip multiprocessors. **IEEE Transactions on Very Large Scale Integration (VLSI)**, v. 15, n. 8, p. 869-880, august, 2007.

MURDOCCA, M.J., HEURING, V.P. **Computer Architecture and Organization**. New York, NY, USA: John Wiley & Sons, 2007.

MYJAK, M.J., DELGADO-FRIAS, J.G. A medium-grain reconfigurable architecture for DSP: VLSI design, benchmark mapping and performance. **IEEE Transactions on Very Large Scale Integration (VLSI)**, v. 16, n. 1, p. 14-23, january, 2008.

NAGELDINGER, U. **Coarse – Grained Reconfigurable Architecture Design Space Exploration.** Ph.D Thesis. University of Kaiserslautern, Computer Science Department, 2001.

NAJJAR, W. A., BOHM, W., DRAPER, B. A., HAMMES, J., RINKER, R., BEVERIDGE, J. R., CHAWATHE, M., ROSS, C. High-level language abstraction for reconfigurable computing. **IEEE Computer**, v. 36, n. 8, p. 63-69, august, 2003.

Ocapi: modeling and mapping. Disponível em: http://www.imec.be/design/ocapi/. Acessado em agosto 2008.

ORDONEZ, E.D.M., SILVA, J.L. **Computação Reconfigurável: Experiências e Perspectivas**. Marilia, SP: Fundação de Ensino Eurípedes Soares da Rocha (FEESR), 2000.

PACT XPP Technologies Inc. **XPP-III Processor Overview – White Paper**. Version 2.0.1, Disponível em: http://www.pactxpp.com. 2006.

PAGE, I. Reconfigurable processor architectures. **Microprocessors and Microsystems**, v. 20, n. 3, p. 185-196, may, 1996.

Parafrase-2, what is? Disponível em: http://www.csrd.uiuc.edu/parafrase2/p2-main.html. Acessado em junho 2008.

PARK, H., FAN, K., MAHLKE, S., OH, T., KIM, H., KIM, H.S. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In: **Proceedings of PACT**, Toronto, Canadá, p. 633-637, october 25-29, 2008.

PATTERSON, D.A., HENNESSY, J.L. **Organização e Projeto de Computadores**, 3a edição. São Paulo, SP: Campus Elsevier, 2005.

PELLERIN, D., THIBAULT, S. **Practical FPGA Programming in C**. Upper Saddle River, NJ, USA: Prentice Hall, 2005.

PERRETTO, M., FERLIN, E.P. FATOSS – sistema escalonador tolerante à falhas para cluster de computadores. In: **Proceedings of 4th International Information and Telecommunication Technologies Symposium**, Florianópolis, SC, CDROM, dezembro, 2005.

PETERSEN, R.J. **An Assessment of the Suitability of Reconfigurable Systems for Digital Signal Processing.** M.Sc Thesis. Brighan Young University, Electrical and Computer Engineering Department, 1995.

PFISTER, G.F. **In Search of Clusters**, 2nd edition. New York, NY, USA: Prentice Hall, 1998.

PILLEMENT, S., SENTIEYS, O., DAVID, R. DART: a functional-level reconfigurable architecture for high energy efficiency. **EUROSIP Journal on Embedded Systems**, v. 2008, article id562326, p. 1-13, 2008.

PVM: Parallel Virtual Machine. Disponível em: http://www.csm.ornl.gov/pvm/. Acessado em agosto 2008.

QUENOT, G., COUTELLE, C; SEROT, J., ZAVIDOVIQUE, B. Implementing image processing applications on a real-time architecture. In: **Proceedings of IEEE Workshop on Computer Architectures for Machine Perception**, New Orleans, MI, USA, p. 34-42, december, 1993.

RUTZIG, M.B., BECK, A.C.S., CARRO, L. Transparent dataflow execution for embedded applicatios. In: **Proceedings of IEEE Computer Society Annual Symposium on VLSI**, Porto Alegre, RS, p. 47-54, march 9-11, 2007.

SALOMAO, C., ALVES, V., CHAVES FILHO, E.C. HiPCrypto: a high performance VLSI cryptographic chip. In: **Proceedings of IEEE ASIC Conference**, Rochester, NY, USA, p. 7-13, September 13-16, 1998.

SANCHEZ, E., HAENNI, J.O., BEUCHAT, J.L., STAUFFER, A., PEREZ-URIBE, A., SIPPER, M. Static and dynamic configurable systems. **IEEE Transactions of Computers**, v. 48, n. 6, p. 556-564, june, 1999.

SANKARALINGAM, K., NAGARAJAN, R., LIU, H., KIM, C., HUH, J., BURGER, D., KECLER, S.W., MOORE, C.R. Exploiting ILP, TLP, and DLP with polymorphous TRIPS architecture. In: **Proceedings of 30th Annual International Symposium on Computer Architecture**, San Diego, CA, USA, p. 422 - 433, june 9-11, 2003.

SATO, L.M., MIDORKAWA, E.T., SENGER, H. Introdução a programação paralela e distribuída. In: **Anais do XV Jornada de Atualização em Informática,** Recife, PE, p. 1-56, 1996.

SCHNEIER, B. Other block ciphers. **Applied Cryptography**, New York, NY, USA: John Wiley & Sons, p. 319-325, 1996.

SCHWAN, K., MUKHERJEE, B. Operating systems for parallel machines. In: **Parallel Computers**, Los Alamitos, CA, USA: IEEE Press, p. 305-348, 1996.

SCROFANO, R., GOKHALE, M.B., TROUW, F., PRASANNA, V. Accelerating molecular dynamics simulations with reconfigurable computers. **IEEE Transactions of Parallel and Distributed Systems**, v. 19, n. 6, p. 764-778, june, 2008.

SHIBATA, Y., MIYAZAKI, H., LING, X., AMANO, H. HOSMII: a virtual hardware integrated with DRAM. In: **Proceedings of 5th Reconfigurable Architectures Workshop**, Orlando, FL,USA, p. 85 - 90, march 30, 1998.

SILBERSCHARTZ, A., GALVIN, P.B., GAGNE, G. **Sistemas Operacionais com Java**, 7a edição. São Paulo, SP: Campus Elsevier, 2008.

SILC, J., ROBIC, B., UNGERER, T. **Processor Architecture**. Berlin-Heidelberg: Spring-Verlag, 1999.

SIMA, D., FOUNTAIN, T., KACSUK, P. **Advanced Computer Architectures: A Design Space Approach**. Harlow, England: Addison-Wesley, 1997.

SINGH, H.S., LEE, M.H., LU, G., KURDAHI, F.J., BAGHERZADEH, N., CHAVES FILHO, E.M. MorphoSys: an integrated reconfigurable system for data-parallel

computation-intensive applications. **IEEE Transactions on Computers**, v. 49, n. 5, p. 465-481, may 2000.

SIPPER, M., SANCHEZ, E. Configurable chips meld software and hardware. **IEEE Computer**, v. 33, n. 1, p. 120-121, january, 2000.

SISAL language tutorial. Disponível em: http://www2.cmp.uea.ac.uk/~jrwg/Sisal/. Acessado em julho 2008.

STALLINGS, W. **Computer Organization and Architecture**, 7th edition. Upper Siddle River, NJ, USA: Pearson Prentice Hall, 2006.

Stanford SUIF - The Stanford SUIF compiler group. Disponível em: http://suif.stanford.edu/. Acessado em maio 2008.

STONE, H.S. **High-Performance Computer Architecture**, 3rd edition. Reading, MA, USA: Addison-Wesley, 1993.

SWANSON, S., MICHELSON, K., SCHWERIN, A., OSCIN, M. WaveScalar. In: **Proceedings of 36th IEEE/ACM International Symposium on Microarchitecture**, p. 291-302, 2003.

SystemC: welcome. Disponível em: http://www.systemc.org. Acessado em agosto 2008.

TANENBAUM, A.S. **Organização Estruturada de Computadores**, 4a edição. Rio de Janeiro, RJ: LTC, 2001.

TANENBAUM, A.S. **Organização Estruturada de Computadores**, 5a edição. São Paulo, SP: Pearson, 2007.

TANENBAUM, A.S., VAN STEEN, M. **Sistemas Distribuídos: princípios e paradigmas**, 2a edição. São Paulo, SP: Pearson Prentice Hall, 2007.

TEIFEL, J., MANOHAR, R. An asynchronous dataflow FPGA architecture. **IEEE Transactions on Computers**, v. 53. n. 11, p. 1376-1392, november, 2004.

TOCCI, R.J., WIDMER, N.S., MOSS, G. **Digital Systems: principles and applications**, 10th edition. Upper Siddle River, NJ, USA: Pearson Prentice Hall, 2006.

TRIPP, J.L., GOKHALE, M.B., PETERSON, K.D. Trident: from high-level language to hardware circuitry. **IEEE Computer**, v. 40, n. 3, p. 28-37, march, 2007.

TURLEY, J. Triscend E5 reconfigures microcontrollers. **Microprocessor Report**, v. 12, n. 15, p. 12-13, 1998.

VAHEY, M., GRANACKI, J., LEWINS, L., DAVIDOFF, D., DRAPER, J., STEELE, C., GROVES, G., KRAMER, M., LACOSS, J., PRAGER, K., KULP, J., CHANNELL, C. MONARCH: a first generation polymorphic computing processor. In: **Proceedings of**

10th Annual Workshop on High Performance Embedded Computing, Lexington, MA, USA, CDROM, september 19-21, 2006.

VARNAGIRYTE, B., ZAMELIS, A., OLSEN, O., KOCH, P., WOLF, O., KAZANAVICIUS, E. A practical approach to DSP code optimization using compiler/architecture. **Ultragarsas**, v. 43, n. 2, p. 28-33, 2002.

VEEN, A.H. Dataflow machine architecture. **ACM Computing Surveys**, v. 18, n. 4, p. 365-398, december, 1986.

VEMURI, R.R., HARR, R.E. Configurable computing: technology and applications. **IEEE Computer**, v. 33, n. 4, p. 39-40, april, 2000.

VERMA, P., SINGHAL, L. **Mini Project Report SUIF2VHDL Converte**r. Indian Institute of Technology, 1999.

VILLASENOR, J., MANGIONE-SMITH, W.H. Configurable computing. **Scientific American**, v. 276, p. 54-59, june, 1997.

VOIGT, S.O., TEUFEL, T. Dynamically reconfigurable dataflow architecture for high-performance digital processing on multi-FPGA platforms. In: **Proceedings of International Conference on Field Programmable Logic and Applications**, Amsterdam, Netherlands, p. 633-637, august 27-29, 2007.

WOLFE, M. **High Performance Compilers for Parallel Computing**. Redwood City, CA, USA: Addison-Wesley, 1996.

WU, B., PENG, C. System-on-chip design with dataflow architecture. In: **Proceedings of 8th International Conference on Computer Supported Cooperative Work in Design, Xiamen, China**, v. 2, p. 712-716, may 26-28, 2004.

WU, K., KANSTEIN, A., MADSEN, J., BEREKOVIC, M. MT-ADRES: multithreading on coarse-grained reconfigurable architecture. **Lecture Notes in Computer Science**, v. 4419, p. 26-38, 2007.

YIANNACOURAS, P., STEFFAN, J. G., ROSE, J. Exploration and customization of FPGA-based soft processors. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 26, n. 2, p. 266-277, february, 2007.

ZDNet UK. Disponível em: http://www.zdnet.com. Acessado em junho 2008.

RESUMO:

Os problemas de engenharia cada vez mais exigem grandes necessidades computacionais, principalmente em termos de capacidade de processamento, sendo que o tempo de execução é um dos pontos-chave em toda essa discussão. Neste sentido o processamento paralelo surge como um elemento decisivo, pois possibilita uma redução do tempo de processamento em decorrência da execução paralela das operações. Outro fator importante é a questão da computação reconfigurável que possibilita combinar o desempenho do *hardware* com a flexibilidade do *software,* permitindo o desenvolvimento de sistemas extremamente complexos e compactos. Este trabalho tem por objetivo apresentar uma proposta de uma arquitetura paralela reconfigurável baseada em fluxo de dados (*dataflow*), que aproveita a potencialidade tanto do processamento paralelo quanto da computação reconfigurável, e que proporciona uma rápida adequação da máquina paralela ao problema a ser resolvido, garantindo um alto desempenho e uma grande flexibilidade de adaptar o sistema paralelo à aplicação desejada. Esta arquitetura visa explorar o paralelismo existente entre as operações envolvidas nos cálculos numéricos, baseando-se no grafo de fluxo de dados do problema a ser solucionado. A arquitetura é composta por uma unidade de controle, responsável por todo o controle dos Elementos Processadores (EPs) e o fluxo de dados entre eles, e de vários EPs que efetivamente realizam a execução da operação. Ao contrário da computação temporal ou sequencial, a computação espacial aproveita e muito a disponibilidade dos vários EPs presentes na arquitetura, garantindo um maior desempenho, além do que a arquitetura pode facilmente ser reorganizada, adaptado-se à aplicação, e isto garante uma flexibilidade na classe de problemas computacionais que podem ser executados nesta arquitetura.

PALAVRAS-CHAVE

Arquitetura de Computador, Processamento Paralelo, Computação Reconfigurável.

ÁREA/SUB-ÁREA DE CONHECIMENTO

1.03.00.00 – 7 Ciência da Computação

1.03.04.02 – 9 Arquitetura de Sistemas de Computação

2008

N°: 39

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Administração
Baixar livros de Agronomia
Baixar livros de Arquitetura
Baixar livros de Artes
Baixar livros de Astronomia
Baixar livros de Biologia Geral
Baixar livros de Ciência da Computação
Baixar livros de Ciência da Informação
Baixar livros de Ciência Política
Baixar livros de Ciências da Saúde
Baixar livros de Comunicação
Baixar livros do Conselho Nacional de Educação - CNE
Baixar livros de Defesa civil
Baixar livros de Direito
Baixar livros de Direitos humanos
Baixar livros de Economia
Baixar livros de Economia Doméstica
Baixar livros de Educação
Baixar livros de Educação - Trânsito
Baixar livros de Educação Física
Baixar livros de Engenharia Aeroespacial
Baixar livros de Farmácia
Baixar livros de Filosofia
Baixar livros de Física
Baixar livros de Geociências
Baixar livros de Geografia
Baixar livros de História
Baixar livros de Línguas

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo