CRD: Um Co-processador Reconfigurável Dinamicamente para Melhoria de Desempenho

Felipe Joffre Romano Renon

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

CRD: Um Co-processador Reconfigurável Dinamicamente para Melhoria de Desempenho

Felipe Joffre Romano Renon[1]

5 de Novembro de 2004

Banca Examinadora:

- Paulo Cesar Centoducatte (Orientador)
- Manuel Euzebio de Lima
  Centro de informática - UFPE
- Rodolfo Jardim Azevedo
  IC - UNICAMP
- Sandro Rigo (Suplente)
  PUC-CAMPINAS

[1]**Apoio financeiro do Programa Nacional de Micro-eletrônica**

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Renon, Felipe Joffre Romano

R295c CRD: co-processador reconfigurável dinamicamente para melhoria de desempenho / Felipe Joffre Romano Renon -- Campinas, [S.P. :s.n.], 2004.

Orientador : Paulo Cesar Centoducatte

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Arquitetura de computadores. 2. Hardware – Arquitetura. 3. Hardware – Linguagens descritivas. 4. Microprocessadores. I. Centoducatte, Paulo Cesar. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 05 de novembro de 2004, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Manoel Eusébio de Lima
CI - UFPE

Prof. Dr. Rodolfo Jardim de Azevedo
IC – UNICAMP

Prof. Dr. Paulo Cesar Centoducatte
IC – UNICAMP

CRD: Um Co-processador Reconfigurável Dinamicamente para Melhoria de Desempenho

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Felipe Joffre Romano Renon e aprovada pela Banca Examinadora.

Campinas, 5 de Novembro de 2004.

Paulo Cesar Centoducatte (Orientador)

Ricardo Pannain (Co-orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

A minha avó Odette (*in memoriam*),
com carinho

Resumo

O desempenho de sistemas computacionais tem sido um requisito recorrente para um grande número de aplicações. Porém, nem sempre as soluções tradicionais para se melhorar o desempenho como por exemplo: o aumento na freqüência de operação dos processadores, a utilização de processamento paralelo etc, podem ser viáveis técnica ou economicamente, principalmente em se tratando de um sistema dedicado. Uma alternativa para a melhoria de desempenho em tais sistemas é a identificação dos trechos da aplicação que são executados de forma pouco eficientes por software e implementá-los diretamente em hardware. Os candidatos naturais para esta abordagem são os laços interiores, que normalmente são pequenos e responsáveis por grande parte do tempo de execução e, que quando implementados em hardware, não fazem uso de uma grande área de silício.

Neste trabalho propomos um co-processador reconfigurável, mapeado em memória, denominado **Co-processador Reconfigurável Dinamicamente** *(CRD)*, capaz de executar trechos de códigos pouco eficientes em software, tais como laços internos (kernels), diretamente em hardware. Com o intuito de reduzir a área ocupada pelo co-processador, diminuindo desta forma o custo do sistema, o *CRD* é dotado de uma unidade de reprogramação, que permite reutilizar os recursos disponíveis para implementar diferentes trechos de programa em hardware em uma mesma instância de execução.

Os trechos de programas escolhidos para serem executados diretamente em hardware (no *CRD*) são aqueles responsáveis pela maior parte do tempo de execução do programa como um todo. O uso desta técnica mostrou um ganho total, no tempo de execução dos programas do benchmark *DSPStone* de até 20 vezes.

Abstract

Performance has beem a current requirement for a great number of applications. However, in some cases, the traditional solutions to improve performance, like: increase frequency of processor's operation, parallel processing etc, can be applied, or to be viable economically, when the improvement object is a embedded system. An alternative solution that can be adopted is to identify the blocks in source code inefficient when implemented in software and to implement them in the hardware directly. Natural candidates are the inner loops, thats normally are small and responsible for great parte of the execution time and that implemented in the hardware doesn't use great silicon area.

In this work we propose a reconfigurable coprocessor system mapped in memory called **CRD**, capable to execute inefficient codes in software, such as internal loops (kernels), directly in the hardware. With intention to reduce the filled area for the ASIC, reducing by this way the price of the system, it has a reprogrammable unit inside of this, destined to fill the lack of memory that is not being more used for a hardware instruction, for other that it will be used in the future.

The parts of chosen programs to be executed in the hardware are those responsible ones mostly of the time of program execution. The use of this technique shows a total speedup of up to 20 times, in the execution time of the *DSPstone* benchmark programs.

Agradecimentos

Um agradecimento todo especial à minha família, principalmente aos meus pais, Honorino e Leonor, e à meu tio Claudino, pelo apoio total e irrestrito, participando comigo nesta caminhada, me fortalecendo e apoiando em vários momentos de decisão.

Ao meu orientador, Paulo Cesar Centoducatte, pelas dicas, idéias e correções que foram preciosíssimas no decorrer do trabalho. Assim como a confiança prestada durante o programa de estágio docente no ano 2001.

A meu co-orientador Ricardo Pannain, pela confiança e apoio prestado.

Aos meus colegas de convívio diário dentro do instituto de computação, tanto aos que já obtiveram o título de mestre e doutor, assim como os que estão a recebê-lo.

Ao CNPq, pelo apoio financeiro (Projeto Nacional de Microeletrônica), à FAPESP pelos equipamentos fornecidos (Processo n^o 2000/5083-9) e ao LSC pelo ótimo ambiente de trabalho que me foi oferecido.

À PIComponentes, Altera Corporation e Mentor Graphics Corp. pelos equipamentos e softwares fornecidos ao Instituto de Computação em seus respectivos Programas Educacionais.

Sumário

Resumo viii

Abstract ix

Agradecimentos x

1 Introdução 1

1.1 Organização do Trabalho 5

2 Trabalhos Relacionados 6

2.1 Arquiteturas Híbridas 10

2.1.1 *Hardware* Reconfigurável para Processamento *Streaming* 13

2.1.2 Projetos Baseados na Modificação de Interconexões 14

2.2 Comparação entre Características Computacionais 15

2.2.1 Desempenho de Várias Aplicações em Sistemas Reconfiguráveis . . 17

2.3 Motivação 20

2.4 Proposta do Trabalho 21

3 Ambiente de Desenvolvimento 23

3.1 O Processador NIOS 24

3.1.1 Periféricos 24

3.1.2 Terminal 27

3.1.3 Placa de Prototipagem 27

4 O Co-processador Reconfigurável Dinamicamente - CRD 32
4.1 Implementação do CRD . . . 34
4.1.1 Banco de Registradores . . . 36
4.1.2 Contador de Endereço de Instrução . . . 36
4.1.3 Contador de Laço de Programa . . . 36
4.1.4 Memória de Programa . . . 37
4.1.5 Comparadores dos Contadores de Instrução e de Laço de Programa 38
4.1.6 Módulo de Programação . . . 38
4.1.7 Bloco Básico . . . 39
4.2 Acesso ao CRD . . . 43
4.2.1 Palavra de Controle . . . 44
4.2.2 Total de Recursos Utilizados . . . 45

5 Linguagem CRDL 47
5.1 Utilização da *CRDL* . . . 48
5.2 Codificação da Linguagem . . . 51
5.2.1 Manipulação de Dados em Memória . . . 52
5.2.2 Criando uma Instrução no *CRD* . . . 52
5.2.3 Realizando a Chamada da Nova Instrução . . . 52
5.3 C como Linguagem de Programação do CRD . . . 53
5.4 Caminho Crítico da Implementação . . . 56
5.5 Latência de Leitura e Escrita . . . 57
5.5.1 Overhead em Operações com Vetores e Matrizes . . . 58

6 Programação de Instruções no *CRD* 60
6.1 Particionamento Hardware/Software . . . 60
6.2 Implementação por Granularidade . . . 63
6.2.1 Implementação de Blocos de Instruções no *CRD* . . . 67
6.3 Implementação de Laços no CRD . . . 71
6.4 Trabalhando com as Limitações do *CRD* . . . 74
6.5 Explorando Paralelismo no CRD . . . 76

7 Resultados **78**

7.1 DSPStone . . . 78
7.2 Dot_Product . . . 80
7.3 Complex_multiply . . . 82
7.4 Real_Update . . . 84
7.5 Biquad_one_section . . . 85
7.6 Fir . . . 87
7.7 Fir2dim . . . 90
7.8 MAT1x3 . . . 94
7.9 Complex_update . . . 95
7.10 N_Complex_Updates . . . 97
7.11 Convolution . . . 101
7.12 Lms . . . 103
7.13 Matrix1 . . . 108
7.14 N_Real_Updates . . . 110

8 Conclusões e Trabalhos Futuros **114**

8.1 Conclusões . . . 114
8.2 Trabalhos futuros . . . 116
8.2.1 Dificuldades encontradas . . . 118
8.3 Contribuições . . . 119

Referências Bibliográficas **120**

A Diretivas de compilação para a tomada dos dados **130**

B Programas benchmark DSPstone **132**

B.1 Dot_Product . . . 132
B.2 Complex_multiply . . . 134
B.3 Real_update . . . 135
B.4 Biquad_one_section . . . 136
B.5 Matrix1 . . . 138

B.6 Complex_Update 140
B.7 N_Complex_Updates 142
B.8 Convolution 144
B.9 Fir 145
B.10 Fir2dim 147
B.11 Lms 150
B.12 Mat1x3 153
B.13 N_Real_Updates 154

Lista de Tabelas

2.1 Classificação de Sistemas reconfiguráveis 17
2.2 Aplicações utilizando FPGAs com seus respectivos speedups 18
2.3 Desempenho para estimação de movimento em diversas plataformas 19
2.4 Ganhos de desempenho entre diferentes sistemas reconfiguráveis 20

3.1 Características básicas do processador NIOS 24
3.2 Descrição dos registradores do "NIOS Timer" 25
3.3 Características principais da FPGA APEX20K200E 29

6.1 *Profile* do *Benchmark* SPECINT 62
6.2 *Profile* do *Benchmark* MediaBench 62
6.3 *Profile* dos *Benchmarks* Secuirty e NetBench 63

7.1 Programas que compõem o benchmark DSPStone 79
7.2 Número de ciclos gastos na execução do *kernel* DOT_PRODUCT 81
7.3 *Speed-up* obtido para o *kernel* DOT_PRODUCT 81
7.4 Número de ciclos gastos na execução do *kernel* COMPLEX_MULTIPLY . . 83
7.5 *Speed-up* obtido para o *kernel* COMPLEX_MULTIPLY 83
7.6 Número de ciclos gastos na execução do *kernel* REAL_UPDATE 84
7.7 *Speed-up* obtido para o *kernel* REAL_UPDATE 85
7.8 Número de ciclos gastos na execução do *kernel* BIQUAD_ONE_SECTION . 86
7.9 *Speed-up* obtido para o *kernel* BIQUAD_ONE_SECTION 86
7.10 Número de Ciclos gastos na execução do *kernel* FIR 88
7.11 *Speed-up* obtido para o *kernel* FIR 88
7.12 Equações do número de ciclos envolvidos na computação do *kernel* FIR . . 89

7.13 Número de Ciclos gastos na execução do *kernel* FIR2DIM 91
7.14 *Speed-up* obtido para o *kernel* Fir2Dim 91
7.15 Equações do número de ciclos envolvidos na computação do *kernel* FIR2DIM 93
7.16 Número de ciclos gastos na computação do *kernel* MAT1x3 94
7.17 *Speed-up* obtido para o *kernel* MAT1x3 94
7.18 Número de ciclos gastos na execução do *kernel* COMPLEX_UPDATE . . . 96
7.19 *Speed-up* obtido para o *kernel* COMPLEX_UPDATE 96
7.20 Número de ciclos gastos na execução do *kernel* N_COMPLEX_UPDATE . 98
7.21 *Speed-up* obtido para o *kernel* N_COMPLEX_UPDATE 98
7.22 Equações que descrevem a computação do *kernel* N_COMPLEX_UPDATE 101
7.23 Número de ciclos gastos na computação do *kernel* CONVOLUTION 101
7.24 *Speed-up* obtido para o *kernel* CONVOLUTION 102
7.25 Equações que descrevem a computação do *kernel* CONVOLUTION 103
7.26 Número de ciclos gastos na computação do *kernel* LMS 104
7.27 Atrasos envolvidos na instrução de *hardware* LMS 104
7.28 *Speed-up* obtido para o *kernel* LMS 105
7.29 Equações do número de ciclos envolvidos na computação do *kernel* LMS . 106
7.30 Número de ciclos gastos na execução do *kernel* MATRIX1 108
7.31 *Speed-up* obtido para o *kernel* MATRIX1 109
7.32 Equações que descrevem o comportamento do *kernel* MATRIX1 110
7.33 Número de ciclos gastos na execução do *kernel* N_REAL_UPDATE 110
7.34 *Speed-up* obtido para o *kernel* N_REAL_UPDATE 111
7.35 Equações que descrevem a computação do *kernel* N_COMPLEX_UPDATE 113

8.1 *Speed-up* obtido nos programas do DSPStone 115
8.2 *Speed-up* obtido com o sistema *NIOS & CRD* no *benchmark* DSPStone . . 117

Lista de Figuras

3.1 Codificação do mapeamento de alguns periféricos no NIOS 26
3.2 Estrutura para manipulação do timer do sistema 27
3.3 Funções destinados a habilitar e retornar o valor do contador 28
3.4 Utilização das funções de tomada de tempo para um bloco de código . . . 29
3.5 Figura ilustrativa do kit *Excalibur* 30

4.1 Sistema utilizando o *CRD* mapeado em memória 33
4.2 Sinais de acionamento do CRD 34
4.3 Diagrama funcional do CRD 35
4.4 Organização da memória de programação interna ao *CRD*. 37
4.5 Instrução de 3 ciclos, repetida n vezes 38
4.6 Módulo de programação 39
4.7 Unidades funcionais dispostas no Bloco Básico 40
4.8 Palavra de controle do Bloco Básico 41
4.9 Palavra de controle do módulo de programação 42
4.10 Endereçamento da memória de programa. 43
4.11 Palavra de controle do *CRD*. 44

5.1 Exemplo de código misto submetido ao pré-processador da *CRDL* 49
5.2 Codificação de uma nova instrução em *CRDL* 53
5.3 Código na linguagem C, gerado pelo pré-processamento da *CRDL* 54
5.4 Código *CRDL* para implementação de um laço 55
5.5 *Datapath* para realizar a operação de multiplicação de uma variável 57
5.6 Overhead de leitura e gravação de instruções no *CRD* 58

5.7 Comparativo na execução do bloco de código "A = A * A" 59
5.8 Trecho de código responsável por estrutura simples de laço na liguagem C e sua transcrição para a linguagem CRDL 59

6.1 Tempo de Execução da Função Potência 64
6.2 Implementação da função $f(x)$ em hardware x software 65
6.3 Código exemplo em C 66
6.4 *Datapath* para Execução da Linha(9) do Código Exemplo 67
6.5 Transcrição de código para a *CRDL* 68
6.6 *Datapath* equivalente a um trecho do código exemplo 69
6.7 *Speed-up* obtido para n Execuções Consecutivas de um Bloco de Instruções 70
6.8 N chamadas a instrução *BLOCO1* 71
6.9 Curva Comparativa de Desempenho entre *Software e Hardware* 72
6.10 Particionamento do Kernel "matrix1x3" 73
6.11 Código Exemplo da utilização da técnica: evolução da variável de indução . 75
6.12 Código exemplo da técnica de distribuição de laços 76

7.1 Tempo de execução do *kernel* DOT_PRODUCT 82
7.2 Tempo de execução para o *kernel* COMPLEX_MULTIPLY 83
7.3 Tempo de execução do *kernel* REAL_UPDATE 85
7.4 Tempo de execução do *kernel* BIQUAD_ONE_SECTION 87
7.5 Tempo de execução do *kernel* FIR 89
7.6 *Speed-up* máximo relativo do *kernel FIR* 90
7.7 Curvas que descrevem o comportamento do *kernel* FIR2DIM 92
7.8 Tempo de execução em nanosegundos do *kernel* FIR2DIM 92
7.9 Speedup máximo relativo em função do número de iterações do laço 93
7.10 Tempo de execução do *kernel* MAT1X3 em nanosegundos 95
7.11 Tempo de execução do *kernel* COMPLEX_UPDATE 97
7.12 Tempo obtido pela execução do *kernel* N_COMPLEX_UPDATE 99
7.13 Interpolação dos dados para o *kernel* N_COMPLEX_UPDATE 100
7.14 *Speed-up* máximo relativo em função do número de iterações do laço 100
7.15 Tempo de execução do *kernel* CONVOLUTION 102

7.16 Etapas de execução do algoritmo true LMS 103
7.17 Modificações necessárias para a execução do novo bloco de código LMS . . 105
7.18 Tempo de execução do *kernel* LMS 106
7.19 *Speed-up* máximo relativo do *kernel* LMS (Dinâmico) 107
7.20 *Speed-up* máximo relativo do *kernel* LMS (Estático) 107
7.21 Curva Característica da execução do *kernel* "matrix1" no *NIOS* 109
7.22 Curva Característica da execução do *kernel* "matrix1" no *NIOS & CRD* . 109
7.23 Tempo de execução do *kernel* Matrix1 para diferentes sistemas 111
7.24 *Speed-up* máximo relativo em função do número de iterações do laço 112
7.25 Tempo de execução do *kernel* N_Real_updates 112
7.26 *Speed-up* máximo relativo em função do número de iterações do laço 113

8.1 Gráficos dos *speed-ups* obtidos para o benchmark DSPStone 116

Capítulo 1

Introdução

O desenvolvimento dos sistemas computacionais eletrônicos, tem experimentado um enorme crescimento desde seu surgimento, há pouco mais de 50 anos. Isto se deve à constante evolução nos processos tecnológicos utilizados para a sua implementação (principalmente CMOS), à evolução das ferramentas de auxílio ao projeto e a inovação no projeto, em si, dos sistemas (Arquitetura). Inicialmente a forma de computação dominante era o uso de grandes máquinas denominadas *Mainframes*[1], que custavam milhares de dólares e necessitavam grande suporte técnico. O surgimento dos microprocessadores no início dos anos 70 e sua constante evolução proporcionou uma grande popularização dos sistemas computacionais o que nos permite hoje em dia comprar, por pouco mais de US$1,000.00, um computador pessoal com mais recursos e poder computacional do que os computadores comerciais da década de 80. O resultado direto dessa evolução, onde temos por um lado aumento de desempenho e por outro redução dos custos, levou ao uso de sistemas computacionais nos mais diferenciados produtos utilizados nas atividades cotidianas, como telecomunicações (telefonia fixa e móvel), veículos de transporte (controle de freios, combustível, estabilidade de vôo etc), eletrodomésticos (controle inteligente de ar condicionados, microondas, geladeiras, máquinas de lavar etc), entretenimento, equipamento médico e hospitalares (tomografia computadorizada etc) entre outras.

Este novo uso dos sistemas computacionais, denominados de sistemas embarcados [10], tem imposto diversos desafios a evolução dos sistemas computacionais tais como portabi-

[1] Computadores de grande custo e porte, capaz de suportar centenas ou milhares de usuários simultaneamente

lidade, baixo consumo de energia, baixa disponibilidade de memória, confiabilidade, I/O intensivo e curto tempo de projeto. Buscando atender a estes requisitos de projeto, os projetistas tem usado em suas soluções diversos tipos de processadores (RISC, VLIW, DSP, ASIP) e em muitos casos o uso simultâneo de mais de um processador de tipos diferentes.

Estima-se que aproximadamente 79% dos microprocessadores produzidos em todo o mundo têm como destino os sistemas embarcados e que programadores escrevem 5 vezes mais código para sistemas embarcados do que para computadores convencionais [56].

Em um grande número de aplicações, onde requisitos como área de silício, potência e desempenho são essenciais, a integracão de todo o sistema em uma única pastilha tem se mostrado uma soluadequada. Ao sistema resultante desta integracão dá-se o nome de *SOC* (System-on-a-chip) [30]. Neste novo cenário, alguns desafios devem ser vencidos. Por um lado, se respeitada a lei de Moore, pela qual o número de transistores disponíveis em um chip dobra a cada 18 meses [59] e a demanda de mercado exigindo cada vez aplicações mais sofisticadas, é possível implementar estas novoas aplicações em sistemas SOC. Por outro lado, este mesmo mercado exige cada vez mais um menor tempo de projeto o que implica em dizer que sistemas com milhões de transistores devem ser projetados em um espaço de tempo de poucos meses [54]. Uma solução, que vem sendo adotada, para enfrentar estas novas demandas do mercado é o projeto do sistema utlizando-se o paradigma de "projeto baseado em plataformas". Uma plataforma é uma arquitetura de *hardware* e software específica para um domínio de aplicação, mas altamente parametrizável (no número de componentes de cada tipo, na estrutura de comunicação, no tamanho da memória, nos tipos de dispositivos de E/S etc.). Uma segunda abordagem é o projeto baseado em IPcores (Intellectual Property) que consiste em projetar os sistemas, usando-se diversos *cores* previamente projetados e validados escolhidos dentre aqueles fornecidos por diversos "fabricantes". Estas estratégias viabilizam o reuso de componentes (ou núcleos) [9] previamente desenvolvidos e testados, o que reduz o tempo de projeto.

Em muitos casos, as necessidades de desempenho e/ou de processamento em tempo real exigidas pelas aplicações não podem serem alcançadas com soluções baseadas em um único processador, exigindo soluções que utilizem: arquiteturas paralelas ou distribuídas [12, 41, 52, 65, 63] baseadas em vários processadores de propósito geral; arquiteturas

baseadas em processadores dedicados à aplicação específica (ASIPs) [95]; arquiteturas reconfiguráveis etc. Assim, com o objetivo de satisfazer os novos requisitos dos sistemas (alto desempenho, baixo custo, baixo consumo de energia, alta disponibilidade etc) tem-se observado o surgimento de novos modelos e estilos de computação.

De uma forma geral, pode-se classificar os tipos de computação para sistemas embarcados em dois grandes grupos: aqueles baseados em software, representados por soluções implementadas em sistemas que utilizam processador(es) programável(eis) de propósito geral e aqueles baseados em *hardware*, representados pelos sistemas que utilizam processadores especificamente projetados para executarem determinada tarefa. Ambas as soluções são bem delimitadas quanto as vantagens e desvantagens em sua utilização quanto ao custo, tempo de projeto, consumo, desempenho e flexibilidade.

No paradigma baseado em software, vê-se claramente uma grande vantagem quando os requisitos principais são flexibilidade e custo. Nesta abordagem tem-se um processador de propósito geral e um código contendo instruções que serão executadas neste processador. Segundo Pressman [67] e Yourdon [99] o processo para este desenvolvimento consiste na descrição e análise do problema e no projeto da solução. Ao final desta etapa, o desenvolvedor possui a solução do problema e deve saber quais os paradigmas de programação e computação que melhor se adequam a solução proposta. Esta solução é então representada por um algoritmo. A codificação deste algoritmo em uma linguagem é então interpretada ou compilada gerando um programa executável para um determinado tipo e nível arquitetural. A execução de software garante flexibilidade ao sistema, permitindo que várias soluções possam ser implementadas, e portabilidade, pois uma vez que a solução do problema está descrita em uma linguagem, pode-se transpor a execução para diversas arquiteturas diferentes, visto que um compilador é quem faz a tradução do código fonte para o código executável. Porém, a grande limitação deste modelo de computação está no baixo desempenho, ou melhor, no desempenho inferior ao que se poderia atingir utilizando a abordagem por *hardware*.

Já no paradigma de *hardware* específico, ou fixo, observam-se vantagens quando o requisito é desempenho, porém, em contrapartida nota-se uma sub-utilizacão do sistema em tarefas não tão usuais. A abordagem baseada em *hardware* consiste em projetar um sistema específico para o problema que se deseja computar. Esta abordagem permite

sistemas com alto desempenho e de acordo com a tecnologia utilizada implementam, geralmente, um único algoritmo. Projetos de *ASIC's* (Application Specific ICs) tiveram grande impulso com o aumento da densidade dos circuitos integrados, assim como o avanço de ferramentas de projeto *VLSI* (*Very Large Scale Integration*). A falta de flexibilidade e o custo são os grandes inconvenientes desta abordagem.

Apesar do exposto, os fatores custo, desempenho e flexibilidade, muitas vezes também não são satisfatórios.

É curioso observar que, comparando a abordagem em *hardware* e em software, em geral, o ponto forte de uma é o ponto fraco da outra e vice-versa. Assim, torna-se interessante a busca por uma solução mista que possa aglutinar os pontos fortes destas duas abordagens. Uma possível solução é particionar o sistema de forma que as tarefas que demandem menos desempenho sejam executadas em um (ou mais) processadores de propósito geral e as tarefas que demandem alto desempenho sejam executadas em um *hardware* específico. Para reduzir a área ocupada pelo *hardware* específico pode-se, ainda, utilizar um dispositivo programável (FPGA - *Field Programmable Gate Array*) que sob demanda pode ser programado para implementar diversas atividades. A esta nova abordagem denomina-se computação reconfigurável. O uso deste tipo de computação permite alcançar desempenhos comparáveis aos desempenhos das soluções baseadas em *ASICs* dedicados com flexibilidade comparáveis às soluções implementadas com processadores de propósito geral, uma vez que se tem várias tarefas sendo realizadas em *hardware* e este pode ser reprogramado para incorporar atualizações no sistema.

Conforme [68], "*Computação reconfigurável representa uma nova idéia em filosofia de computação, na qual algum agente de* hardware *de propósito geral é configurado para realizar uma tarefa específica, mas pode ser reconfigurado sob demanda para realizar outras tarefas específicas*".

Segundo Hartenstein [34], a proposta de *hardware* reconfigurável, estará cada vez mais presente em sistemas embarcados, onde a enorme vantagem de se incluir a reconfiguração de *hardware* é a possibilidade extra de personalização de um chip, e a atenuação do enorme custo de máscaras que as tecnologias nano-métricas apresentam.

Neste trabalho propomos um co-processador reconfigurável, mapeado em memória denominado **Co-processador Reconfigurável Dinamicamente** *(CRD)*, capaz de exe-

cutar trechos de códigos pouco eficientes em software, tais como laços internos (kernels), diretamente em *hardware*. Com o intuito de reduzir a área ocupada pelo co-processador, diminuindo desta forma o custo do sistema, o *CRD* é dotado de uma unidade de re-programação, que permite reutilizar os recursos disponíveis para implementar diferentes trechos de programa em *hardware* em uma mesma instância de execucão.

1.1 Organização do Trabalho

Este trabalho está organizado da seguinte forma: no próximo capítulo, são apresentados alguns trabalhos realizados sob a óptica da computacão reconfigurável, bem como alguns conceitos inerentes a área. No capítulo 3 é apresentado o ambiente utilizado para o desenvolvimento do protótipo do *CRD*, dando uma ênfase maior à apresentação do processador *NIOS*, utilizado como processador host do sistema. No capítulo 4, são apresentados em detalhes a arquitetura do *CRD*. No capítulo 5 é apresentada a linguagem híbrida *CRDL*, desenvolvida como parte deste trabalho e utilizada para programação do co-processador. No capítulo 6, são apresentados formas de se obter melhores ganhos de desempenho na utilização do *CRD*, como a utilização de paralelismo e variação da granularidade como critério para o particionamento entre *hardware* e software, além de algumas técnicas de implementação para contorno dos limites físicos do co-processador.

No capítulo 7, são apresentados os ganhos obtidos na utilização do *CRD* em conjunto com o processador de propósito geral *Nios*, a partir de testes realizados em bancada. Os dados, são ainda confrontados com o tempo de execução obtido por diversos *DSPs*.

Por fim, no capítulo 8, apresentamos as principais dificuldades envolvidas na elaboração deste trabalho, bem como algumas conclusões e trabalhos futuros.

No final deste trabalho, é adicionado sob forma de anexo, o código fonte dos programas pertencentes ao DSPStone para identificação do bloco transcrito em *hardware*, delimitados pela diretiva *START_PROFILING* e *END_PROFILING*.

Capítulo 2

Trabalhos Relacionados

Em um sistema reconfigurável o *hardware* disponível para execução de uma dada tarefa pode ser modificado no tempo, adaptando-se para melhor executar a tarefa. Esta transformação (configuração) do *hardware* pode se dar de duas formas: uma de maneira estática, quando a configuração do *hardware* é realizada antes do início da execução da tarefa e não sofre mais alterações até o término da mesma e outra dinâmica, quando as adaptações no *hardware* são realizadas *on-the-fly*[1] durante a execução da tarefa. A configuração dinâmica, pode ainda, ser classificada como total, onde o dispositivo encerra a execução de determinada computação para promover a configuração de uma próxima instrução e parcial, onde a execução na unidade reconfigurável se dá juntamente com a configuração de uma próxima computação. Infelizmente com a tecnologia atual, a configuração de um dispositivo programável (FPGA) gasta tempos medidos em milisegundos o que restringe o campo de aplicação de sistemas reconfiguráveis dinamicamente, mesmo quando se usa os dispositivos reconfiguráveis mais novos que admitem reconfiguração parcial. Uma solução que pode ser adotada para se contornar esse problema introduzido pelo tempo de programação do dispositivo programável é usar a programação estática, onde alguns trechos do programa são escolhidos para serem implementados diretamente em *hardware*, sendo programados no dispositivo antes do início da execução da tarefa. O dispositivo não sofre mais alteração até o fim da execução. Nesta abordagem os dispositivos programáveis podem ser previamente particionados em trechos que terão aplicação específica, seja como componentes de hw ou aceleradores de software. Estes pequenos

[1] em tempo de execução

trechos do programa para execução direta em *hardware* são, em geral, responsáveis pela maior parte do tempo de execução do mesmo.

Embora se encontrem diversas referências e trabalhos sobre arquiteturas reconfiguráveis, a grande maioria ainda está direcionado à pesquisa. Alguns trabalhos de maior interesse e resultados na área são abordados em seguida, muitos deles contribuíram fortemente para a elaboração do projeto apresentado nesta dissertação.

SPLASH2

SPLASH2 [11] é um dos primeiros trabalhos apresentados na literatura que possui um protótipo implementado em uma placa contendo diversas FPGA's (a versão atual contém 17 FPGAs Xilinx 4010 [97]) conectada à uma SPARCStation SBus[1]. O *SPLASH2* utiliza o conceito de reprogramação estática, isto é, uma vez definida a aplicação, a sua solução por *hardware* é programada nas FPGAs e durante a execução da tarefa o *hardware* não é mais reconfigurado. O *hardware* é programado para executar toda a tarefa, sem alternância de execução de parte da tarefa em *hardware* e parte da tarefa na estação de trabalho. Nos primeiros trabalhos, o *SPLASH2* foi utilizado para executar uma ampla diversidade de tarefas e os seus resultados comparados com aqueles obtidos pela execução das mesmas tarefas, implementadas em software, executadas em uma SPARC10. Para o cálculo de distâncias de edição em cadeias de DNA [11, 50] obteve-se um ganho acima de 2500 vezes. Para aplicações de computação gráfica (filtro da mediana aplicado a uma imagem em tons de cinza) obteve-se ganhos de 140 vezes. Estes resultados despertaram um grande interesse pela computação reconfigurável em diversos centros de pesquisas. Atualmente, pesquisas usando o *SPLASH2* vêm sendo realizadas no *Supercomputing Research Center*.

PRISM Machines

O *PRISM Machines* [5, 93] também foi, um dos primeiros protótipos com computação reconfigurável apresentado na literatura e foi desenvolvido no *Virginia Polytechnin Institute*. A versão desse protótipo, **PRISM-I** (Processor Reconfiguration through Instruction-Set Metamorphosis), consistia de uma placa com 4 Xilinx 3090 [97], conectadas a um sistema baseado em um processador Motorola 68010 [61]. É um *hardware* reconfigurável baseado

em uma placa multi-FPGA, como no sistema *Splash*. A diferença no entanto, da-se pelo fato da transparência existente entre o *host*[2] e o protótipo. Para o desenvolvimento de aplicações foi desenvolvido um compilador C, que com alguma assistência do programador, é feita a extração do trecho de código (trechos menos eficientes) a ser implementado em *hardware*. Este compilador distribui a execução do processamento entre o *host* e o dispositivo reconfigurável, os programas compilados rodam parcialmente no processador do *host* e parcialmente na placa FPGA.

Com o intuito de manter uma maior aproximação entre a unidade reconfigurável e o processador host, de forma a reduzir atrasos ocasionados pela interconexão das unidades, foi desenvolvido o **PRISM-II** [93], composto por um processador *host*(*AMD29050*) [2] e a placa (*3 Xilinx 4010*).

Para ambas as máquinas, o tamanho das FPGAs limitam os laços internos a pequenas funções. Para estes pequenos laços implementados, conseguiu-se *speed-ups* de 7 a 86 vezes quando comparados com o host e de 2 a 25 vezes por FPGA.

Spyder machine

Outro protótipo muito similar ao *PRISM*, é o *Spyder machine*, desenvolvido na *École Polytechnique Fédérale*. O *Spyder machine* [43] estende um processador customizado com três Xilinx 4010 atuando como unidade de execução reconfigurável. Neste projeto, o programador se torna responsável pelo particionamento do programa entre o processador principal e a unidade de reconfiguração

DISC

Usando a mesma abordagem que os trabalhos anteriores, porém procurando uma forma de manter uma maior integração entre processador principal e unidade reconfigurável, pesquisadores da *Brigham Young University*, criaram uma máquina híbrida chamada de **DISC**, abreviação de *Dynamic Instruction Set Computer* [40, 94]. Sugerem neste trabalho manter em uma mesma FPGA o processador principal e a unidade reconfigurável.

A FPGA utilizada é particionada em duas áreas, uma implementa o módulo de controle global estático e outra o espaço de instruções dinâmicas. A unidade de controle estático é

[2] computador hospedeiro

sempre executada na FPGA como uma unidade global não volátil. Seu propósito está diretamente ligado ao gerenciamento de memória, recursos e canais de comunicação. Antes de uma aplicação DISC iniciar, a unidade de controle global é carregada para o *hardware* e o espaço de instruções dinâmicas é limpo. Conforme o espaço de instruções for sendo preenchido, módulos de instruções são removidos para alocar *hardware* adicional para novos módulos de instruções. Para reduzir o *overhead*[3] de configuração, a FPGA é parcialmente configurada.

Duas versões deste protótipo foram desenvolvidas, o *DISC* original e o *DISC II*. Na primeira versão, o *DISC* utiliza dois *National Semiconductor CLAy31*, o segundo utiliza três, reservando o terceiro apenas para implementar o processador principal, tornando o dispositivo mais poderoso.

Apesar dos dois protótipos não possuírem um compilador, a tarefa de reconfiguração é desempenhada por uma ferramenta de síntese para FPGAs usuais. O mecanismo de reconfiguração do *DISC* é tratado como uma pequena cache que inicia a gravação da programação na FPGA, na ocorrência de um *miss*, que gera um *stall* no sistema com a duração necessária à reconfiguração.

Para um conjunto de aplicações baseadas em filtros utilizados em processamento de imagens como a convolução bidimensional, foi alcançado um desempenho de 80 vezes em relação a um sistema utilizando um processador de propósito geral na computação do mesmo algoritmo.

Kestrel

Outro sistema baseado em FPGAs, fazendo uso de um co-processador como unidade reconfigurável, que merece destaque é a unidade reconfigurável *Kestrel* [51], desenvolvida na *University of Califórnia Santa Cruz*, consistindo em um processador paralelo reconfigurável. O *Kestrel* tem como principal objetivo, oferecer ganhos significativos em análises de sequências moleculares (formada por cadeias de proteínas ou aminoácidos) e outras aplicações em que o paralelismo de dados pode ser aplicado de forma eficiente. Dotado de processadores elementares que podem ser configurados de forma a obter um *array* sistólico, o projeto em questão, com um total de 512 processadores elementares, obteve

[3] na grande maioria das vezes, atraso decorrente à transferência de dados

ganhos significativos em aplicações na área de biologia computacional.

O protótipo deste co-processador, com um total de 512 processadores elementares, realiza comparações de seqüências de 20 a 40 vezes mais rápido que estação de trabalho com freqüência de 433 Mhz.

2.1 Arquiteturas Híbridas

Em computação reconfigurável com a utilização de FPGAs, como apresentado anteriormente, o dispositivo reconfigurável fica sem uso em grande parte do programa, principalmente quando não executa um número considerável de repetições. Para suprir esta deficiência e tentar um aumento no desempenho destes sistemas, é apresentado o modelo de *computação reconfigurável híbrida*, que consiste no acoplamento de um dispositivo reconfigurável a um processador, com o intuito de explorar um pouco as duas arquiteturas. A utilização de uma unidade reconfigurável separada do processador pode acarretar problemas quanto a compatibilidade de sinais e conexão de dados, fazendo com que desta forma a adoção de um processador de propósito geral como processador *host* se torne inviável. A literatura apresenta como um tipo de arquitetura híbrida, processadores de propósito geral com alterações em seu *datapath*, de maneira a diminuir atrasos decorrentes à configuração ou programação e aumentar o desempenho de possíveis instruções desenvolvidas em *hardware*. Esta abordagem cria um sistema específico integrado, diferente dos que utilizam uma unidade reconfigurável externa. Estes últimos, apesar de apresentarem perdas referentes à *overheads* e comunicação de dados, ganham em portabilidade devido à generalidade do canal de comunicação de dados e comandos.

Com esta organização, o *hardware* reconfigurável seria utilizado para executar os laços mais internos (*kernels*) de uma aplicação, enquanto o processador tradicional ficaria responsável pelo processamento da massa de código restante.

PRISC

Alguns projetos como o **PRISC** [75], *Programable Instruction Set Computers*, desenvolvido na *Universidade de Harvard*, sugeriam pequenas modificações em um datapath de um processador RISC, de forma que um novo *opcode*, de tamanho igual a 11 bits, fosse

desenvolvido com a finalidade de implementar as instruções da unidade reconfigurável. Com o novo *opcode* criado, poderiam ser gerenciadas 2048 novas instruções programadas na unidade reconfigurável adicional. Foram desenvolvidas diversas ferramentas em software, como um compilador, destinado a determinar, converter, otimizar e programar, as sequências de instruções pouco eficientes em software.

Uma grande limitação, está no fato de que o protótipo *PRISC* é capaz de realizar apenas funções combinacionais. Outra limitação imposta é que as novas instruções possuam no máximo dois operandos e produzam uma única saída, assim instruções complexas não podem ser implementadas diretamente no *hardware*. Os resultados apresentados na literatura são referentes à simulações e apresentam ganhos de 9% a 91% em relação a um processador de propósito geral.

MorphoSys

Utilizando o *TinyRISC* [85], o projeto *MorphoSys* [31, 49] modifica o datapath do processador host de forma a manter uma matriz de unidades reconfiguráveis, de tamanho 8x8. Com a grande finalidade de acelerar algoritmos de computação gráfica, como compressão de dados (codificação e decodificação em aplicações de vídeo que utilizam padrão MPEG [73]) e aplicações ATR (Reconhecimento Automático de Padrões) além de criptografia. Todas as unidades reconfiguráveis possuem uma ULA (Unidade lógica e Aritmética) [37] de 16 bits, dois multiplexadores que realizam a seleção das entradas das ULAs, uma unidade de deslocamento (*shift*) e um banco de registradores. A matriz é fisicamente construída de modo a permitir uma ampla interligação entre todos os registradores de todas as unidades de uma linha ou coluna.

Na análise da codificação e decodificação do padrão MPEG, em aplicações de reconhecimento automático de padrões e em um algoritmo de criptografia foi reportado ganhos de 53 vezes quando comparadores a um PENTIUM [42] II 233Mhz[4].

GARP

Notadamente a arquitetura **Berkeley GARP** [35, 91, 92], apresenta como grande diferencial entre as demais arquiteturas, um pequeno *DMA* (Acesso Direto à Memória) [37]

[4]Ver Comparação entre características computacionais para maiores detalhes - Seção 2.2

interno a unidade reconfigurável, sendo capaz de acelerar operações que necessitem trocas, leituras e escritas em memória.

Para realizar sua reconfiguração e a transferência de dados entre os registradores do processador de propósito geral, foram necessárias algumas modificações na estrutura interna do processador principal (*datapath*), sendo necessária a criação de novas instruções para estes propósitos. A maneira adotada para a configuração de uma nova instrução no *hardware* foi o desenvolvimento de um software que recebe uma descrição comportamental da configuração desejada, de forma textual, e retorna um arquivo para ser incluído no código fonte do programa que fará uso desta nova operação.

Para a execução do algoritmo DES (*Data Encryption Standard*), a literatura reporta ganhos de desempenho na ordem de 19 vezes, e de 17 vezes na execução de um filtro *dithering* em uma imagem.

Chimaera, ConCISe

Outros trabalhos que utilizam unidade reconfigurável como unidade do processador, e que merecem destaque são o **Chimaera** [7, 45] desenvolvido na *Northwestern University*, e o **ConCISe** [39] desenvolvido no *Philips Research Laboratories*. Suas unidades reconfiguráveis, assim como no *PRISC*, são vistas como uma unidade funcional de um processador, onde as entradas de dados são obtidas através do *register file* [37] e os resultados armazenados de volta. Porém, devido a falta de qualquer registrador interno, a unidade funcional reconfigurável não suporta a implementação de laços no circuito.

Para todos estes sistemas apresentados, o método de configuração empregado foi o mesmo utilizado com sucesso pelo *DISC*, as configurações são executas através de um gerenciador de funcionalidade semelhante a uma memória cache.

Quando aplicadas a unidade funcional reconfigurável *Chimaera*, *speed-ups*[5] de 2 a 4 vezes foram reportados para computações de propósito geral. Em aplicações de compressão (Compress Benchmark) o sistema apresentou um ganho de desempenho de 1.1 vezes.

Quando submetidos a aplicações como o algoritmo *life*[6], onde é possível uma optimização paralela agressiva, foram alcançadas taxas de até 160 vezes [45].

[5] ganho de desempenho

[6] inventado por J. H. Conway, em 1969, o algoritmo *life* implementa um autômato celular

NAPA100

O *NAPA100* [33] (*National Adaptative Processing Architecture*) é um sistema que possui em um mesmo circuito integrado a unidade reconfigurável, processador e memória, reduzindo assim problemas de comunicação e largura da banda de dados, entre o processador principal e a unidade reconfigurável.

Um fato inovador consiste na arquitetura de sua unidade reconfigurável que aceita mais de uma configuração ao mesmo tempo, podendo executar uma nova configuração na unidade de controle simultaneamente com a execução de outra.

Através da linguagem *NAPA-C* [78], pedaços de código escritos em alto nível são convertidos em bits que são responsáveis pela reprogramação da unidade reconfigurável. A linguagem apresentada pelos autores oferece uma grande semelhança com a linguagem *C*, apresentando diretivas para estruturação da unidade reconfigurável, como *#pragma* que limita o bloco que deve ser implementado na unidade reprogramável.

Resultados pouco expressivos foram demonstrados neste tipo de abordagem, raramente com *speed-ups* superiores a 2 vezes.

2.1.1 *Hardware* Reconfigurável para Processamento *Streaming*

Seguindo uma outra linha, algumas pesquisas tem se dedicado ao desenvolvimento de *hardware* reconfigurável para aplicações do tipo *stream*[7]. Alguns exemplos de aplicações *streams* incluem filtro de audio e vídeo, compressão, descompressão e criptografia, assim como problemas que possam ser decompostos em operações de posições contíguas de memória, tais como manipulação de vetores.

O sistema **OneChip-98** [15] é uma arquitetura de reconfiguração estática, orientada a *stream*. Neste projeto somente uma entrada de *stream* e uma saída simples são suportados. Existem também restrições quanto ao tamanho da *stream*, tornando o sistema muito limitado.

O **RaPiD** [38, 19, 28, 29] é um projeto desenvolvido na *University of Washington* que busca a mesma abordagem de implementação. Sua unidade reconfigurável é composta de uma unidade funcional de 16 bits e os registradores são conectados por uma rede

[7] Dados em ordem sequencial, processados geralmente através de uma FIFO

configurável. Utiliza-se neste projeto suporte a múltiplos streams, onde pelo menos 3 são suportados. Procurou-se, para um maior desempenho das aplicações submetidas ao sistema, a formação de arranjos sistólicos para grande massa de dados.

São numerosas as aplicações do tipo *streams* que podem ser decompostas em uma sequência de pequenas operações, igualmente atuantes, fazendo com que algumas delas possam ser executadas em unidades reconfiguráveis de tamanhos variáveis, com sua velocidade de execução proporcional à quantidade de *hardware* reconfigurável disponível.

Os trabalhos mais interessantes nesta área tem sido desenvolvidos na *Carnegie Mellon University*, sendo a unidade reconfigurável **PipeRench** [?, 48] o principal trabalho neste segmento. A proposta principal deste trabalho é um *pipeline* [37] onde todos os estágios são reconfiguráveis. Cada estágio físico do pipeline do PipeRench é formado por 16 processadores elementares, onde cada um contém 8 registradores de 8 bits e um bloco lógico programável em SRAM. Para a programação de cada estágio do pipeline são necessários 672 bits, previamente armazenados na memória de configuração do chip. A entrada e saída de dados entre o pipeline e o exterior é feita através de memórias FIFO. A arquitetura do PipeRench, contudo, não permite trabalhar com problemas que não possam ser adequadamente representados sob a forma de *streams*.

Apresentado em [6], o **Wormhole** é dotado de uma arquitetura completamente reconfigurável através de um *stream* que contém um cabeçalho com informações de reconfiguração do *datapath*, incluindo a reconfiguração do caminho dos dados (*crossbar routing*) e das unidades funcionais. Em seguida ao cabeçalho aparecem os dados a serem processados que trilham o caminho configurado.

2.1.2 Projetos Baseados na Modificação de Interconexões

Alguns dos projetos mencionados, diferentemente daqueles nos quais faziam das FPGAs a base da sua unidade reconfigurável, criaram novas unidades para seus processadores. Outros estudos porém, têm concentrado esforços na matriz de reconfiguração, sem especificar exatamente como ela é conectada com a memória ou com o processador principal.

O projeto experimental da *NEC*, chamado de **SOP** [98] (*Sea Of Processor*), sugere reotimizar os blocos lógicos de uma FPGA, uma versão do *array* reconfigurável foi desen-

volvido no *Tokyo Institute of Technology.*

Existem diversos outros trabalhos que empenham-se em modificar o projeto das FPGAs existentes de acordo com suas necessidades ou sugerir novos métodos de interligação entre unidades, entre eles podemos citar: **CHESS**, descrito por Marshall et al. [55], **MATRIX** desenvolvido no *Massachussets Institute of Technology* [22], assim como diversas outras estruturas e idéias como **ICARUS** [8], **Kumar** [47], **Nimble** [53] e **Riley-2** [70],

A partir dos anos 90, várias companhias como a *Virtual Computer Corporation* (`www.vcc.com`), *Alpha Data Parallel Systems* (`www.alphadata.co.uk`) e *Nallatech Limited* (`www.nallatech.com`), lançaram no mercado placas *plug-in* contendo pequenas quantidades de FPGAs, às vezes apresentando apenas uma. Estas placas quando "plugadas" em uma expansão, como por exemplo em um barramento PCI em uma estação de trabalho, funcionam tipicamente como um acelerador genérico para aplicações diversas.

2.2 Comparação entre Características Computacionais

De acordo com algumas classificações e conceitos pertinentes a computação reconfigurável, é apresentada uma tabela comparativa (Tabela 2.2) contendo os principais trabalhos realizados na área. A reconfiguração, que é a possibilidade de alterar a configuração do dispositivo inúmeras vezes se necessário, classifica-se em:

- Reconfiguração estática ou dinâmica

  Como visto anteriormente, chama-se reconfiguração estática aquela reconfiguração que é realizada com o dispositivo no estado inativo ou parado. A reconfiguração dinâmica é aquela realizada com o dispositivo no seu funcionamento normal.

- Reconfiguração parcial ou total

  Define-se reconfiguração total quando é necessário reconfigurar todos os recursos reconfiguráveis do dispositivo. A parcial restringe-se a configuração de apenas algumas partes do dispositivo.

- Reconfiguração em tempo de execução ou compilação

Define-se reconfiguração em tempo de execução quando existe a possibilidade de se reconfigurar o dispositivo durante a fase de execução ou processamento (operação do dispositivo). A reconfiguração em tempo de compilação limita-se a rearranjar o dispositivo no início da fase de execução ou processamento (normalmente durante a fase de compilação).

- Reconfiguração única ou múltipla

  Define-se reconfiguração múltipla quando se faz necessário mais de um padrão de configuração (bits de configuração), armazenados no dispositivo. A reconfiguração única utiliza apenas um padrão.

- Reconfiguração local ou remota

  Diferencia-se o tipo de reconfiguração, entre local ou remota, pela proximidade (localização física) do padrão de configuração e dispositivo

Quanto ao tipo de interface, local e remota, definida pelo tipo de conexão entre o host e o dispositivo reprogramável, que pode ser feita através de cabo ou memória PROM.

Um sistema reconfigurável pode, ainda ser classificado como:

- Monoprocessador
- Multiprocessador: Neste caso, ainda podem extender-se à:
  1. SIMD - Single Instruction Multiple Data
  2. MIMD - Multiple Instruction Multiple Data
- VLIW - Very-Long Instruction Word;
- Pipelined - Também conhecido como paralelismo em nível de instrução.

Muito empregado na área, o termo granularidade define o tamanho do grão da implementação lógica do dispositivo reconfigurável, distingue-se entre fina, média e grossa ou, em nível de bits, instrução ou bloco de instruções.

Sistema	Granularidade	Programabilidade	Reconfiguração	Interface	Modelo	Domínio
DPGA [21]	Fino	Múltipla	Dinâmica	Remota	Monoproc.	Computação em nível de bits
GARP	Fino	Múltipla	Estática	Local	Monoproc.	Computação em nível de bits
Splash	Fino	Múltipla	Estática	Remota	Monoproc.	Computação em nível de bits
DEC PeRLe-I [64]	Fino	Simples	Estática	Remota	Monoproc.	Computação em nível de bits
Chimaera	Fino	Simples	Estática	Local	Monoproc.	Computação em nível de bits
OneChip	Fino	Simples	Estática	Local	Monoproc.	Controladores, Aceleradores
DISC	Fino	Simples	Dinâmica	Local	Monoproc.	propósito geral
PADDI [80]	Grosso	Simples	Estática	Remota	VLIW, SIMD	Aplicações DSP
MATRIX	Grosso	Múltipla	Dinâmica	Não definido	MIMD	Não definido
RaPiD	Grosso	Simples Estática	Maioria	Remota	Arranjo Linear	Arranjos sistólicos
Remarc [58]	Grosso	Múltipla	Estática	Local	SIMD	Aplicação de dados paralelos
RAW [90]	Misto	Simples	Estática	Local	MIMD	propósito geral
PipeRench	Misto	Múltipla	Dinâmica	Remota	Com Pipeline	Dados e DSP
Morphosys	Misto	Múltipla	Dinâmica	Local	SIMD	Dados paralelos

Tabela 2.1: Classificação de Sistemas reconfiguráveis

2.2.1 Desempenho de Várias Aplicações em Sistemas Reconfiguráveis

Durante a revisão bibliográfica deste trabalho procurou-se fazer uma breve comparação de desempenho, entre a maioria dos sistemas apresentados, porém a forma em que são apresentados os resultados são tanto quanto questionáveis. Para se ter uma boa base de comparação seria necessário ter uma equivalência entre tecnologia, padronização do algoritmo utilizado, exclusão de optimização de código etc. Em alguns artigos estudados o ganho de desempenho obtido com a implementação em *hardware* é ilusória, um tratamento é dado antes da inclusão do código em *hardware*, o que já pode gerar ganhos

significativos através de otimizações inerentes em alguns compiladores. No capítulo de resultados deste trabalho, é apresentado o código compilado com e sem diretivas de otimização, em alguns casos apenas com a utilização de chaves de otimização se consegue um ganho de desempenho razoável em determinados trechos de programas.

Conforme apresentado anteriormente, o sistema Splash apresentou *speed-ups* de até 2500% com relação a uma *Workstation* SPARC10, para o cálculo de distância de edição de cadeias de *DNA* e um ganho de até 140 vezes para a execução de um filtro mediana em uma imagem em tons de cinza, utilizando para comparação a mesma estação. São resultados realmente motivadores e não é surpreendente que tenham se mantido inalcançáveis por vários anos. Cabe a pergunta: *como foi medido o desempenho?*

Isto se deve, contudo, ao fato do sistema *Splash 2*, contendo 17 FPGAs e um *SPARC10*, ser construído em volta de um microprocessador em um único circuito integrado. Da mesma forma que uma placa especial com 17 processadores *SPARCs* tendem a ser mais rápidos que um único *SPARC10*. Quando o *speed-up* acima é dividido pelo número de FPGAs ativas utilizadas, o *speed-up* por chip cai para um fator de 147 vezes para o cálculo da distância de edição no caso de cadeias de *DNA* e abaixo de 10 para o filtro mediana em tons de cinza. Estes número são ainda interessantes mas coloca os speed-up por FPGA em uma melhor perspectiva.

Algumas aplicações recentes em FPGAs tem seu melhor *speed-up* limitado entre 10 e 30 vezes por chip, contendo somente um pequeno número de aplicações onde se encontra valores acima destes patamares, conforme pode se observar na tabela 2.2.

Aplicação	Dispositivo	Comparação	Speed-up por chip	referência
Reconhecimento de alvos militares	Splash-2	HP 770 (110Mhz)	7.5	[69]
Comparação genética	VME Card	SPARC 20	7.9	[60]
Reconhecimento de alvos por I.R.	Placa PCI 16 Xilinx 4020	Pentium (180)Mhz	1.24	[44]

Tabela 2.2:
Algumas aplicações utilizando FPGAs e seus respectivos speedups por chip

No sistema *MorphoSys*, também foram reportados ganhos de desempenho notáveis [31],

na análise da codificação e decodificação do padrão MPEG, onde se enfatiza as fases de estimativa de movimento em aplicações de reconhecimento automático de padrões e em um algoritmo de criptografia. Os resultados obtidos pelo *MorphoSys* quando confrontados diretamente ao desempenho obtido por uma *Workstation* Pentium II 233Mhz e um processador de sinais digitais TMS320C64X [84], são apresentados na tabela 2.3.

Processador	Ciclos de relógio
DSPTMS320C64X	2100
Pentium II	29000
MorphoSys	540

Tabela 2.3: Desempenho para estimação de movimento em diversas plataformas

Em ciclos de máquina, o MorphoSys apresenta ganhos muito significativos em relação às outras duas abordagens, porém quando se mede desempenho é imprescindível que se leve em consideração o tempo destinado a executar uma tarefa. O TMS320C64X é um DSP capaz de trabalhar em uma frequência de operação superior a 600Mhz, enquanto o Morphosys é limitado a 100Mhz. Nestas condições não se obtém um ganho de desempenho em uma abordagem em relação a outra e sim um decréscimo, ou um *speed-up* menor que um quando o MorphoSys é confrontado com o TMS320C64X. Analogamente, temos o processador de propósito geral Pentium [42] que mesmo operando em uma frequência maior ainda possui um desempenho menor que o MorphoSys sendo executado a 100Mhz. Comparativamente há um ganho de aproximadamente 20 vezes entre os dois sistemas.

O que o autor do artigo relata, no entanto, é que o algoritmo que está sendo executado no MorphoSys é diferente daquele que está sendo executado nas outras abordagens.

Há várias técnicas para se implementar esta aplicação, sendo a mais conhecida a do algoritmo BM (*Blocking Matching*), pois ele permite um projeto melhor em *hardware*. A implementação usada no Morphosys baseou-se no algoritmo FSBM (*Full Search Block Matching*), este facilmente paralelizável e portanto explorando a grande característica do sistema.

Pode-se encontrar, na literatura, resultados obtidos pelos diferentes tipos de sistemas reconfiguráveis além dos apresentados anteriormente, porém não existe uma padronização no tipo de algoritmo utilizado, instâncias de entrada para o problema a ser resolvido,

tecnologia empregada entre outros aspectos. A tabela 2.4 apresenta alguns sistemas com seus respectivos ganhos de desempenho.

Sistema	*Speed-up* em relação ao host	*Speed-up* por FPGA
PRISM	7 a 86	2 a 25
Kestrel	20 a 40	N.A
PRISC⋆	1.09 a 1.91	N.A
GARP*dagger*	até 24	N.A
NAPA100	até 2	N.A
Chimaera	de 2 a 4	N.A

Tabela 2.4: Ganhos de desempenho entre diferentes sistemas reconfiguráveis

O "N.A" apresentado na tabela significa "*Não se Aplica*", ou não foi encontrada nenhuma referência para a categoria.

2.3 Motivação

Pelas duas últimas décadas, o projeto de processadores de propósito geral, tem concentrado esforços em aplicações *stand-alone* e não em *real time*. Porém aplicações multimídia estão agora começando a exercer grande impacto no *Workload*Carga da aplicação no espaço de trabalho do processador destes sistemas de computação [24] [46]. Tecnologia de componentes multimídia como vídeo-conferência, visualização de gráficos 3D, animação, simulação realística, reconhecimentos de voz, entre outros, estão cada vez mais ganhando espaço.

Tradicionalmente, aplicações de processamento de sinais digitais são desempenhadas unicamente por um processador *DSP*, que é especializado para realizar um intenso número de operação existentes em vários algoritmos de processamento de sinais.

No entanto, apesar de existirem processadores especializados com arquitetura optimizada para desenvolver tarefas de processamento de sinais digitais, alguns pesquisadores afirmam que a utilização conjunta de um processador DSP, para o processamento específico de sinais e um processador de propósito geral para outras aplicações contidas no contexto do *Workload*. Esta abordagem, no entanto, aumenta a área ocupada, o custo e complexidade do sistema.

Contudo, a importância desta tecnologia, serviços e aplicações, está cada vez mais ganhando consideração por desenvolvedores de microprocessadores. Alguns processadores como o Trimedia [87] da Philips, Multimedia Signal Processor da Sansung, Multimedia Extensions (MMX) da Intel [42], *MIPS* Digital Media eXtension (MDMX) da Mips [36, 81] e o Visual Instruction Set (VIS) da Sun [79], possuem rotinas em *hardware* para funções multimídia. Além disto, o número de fabricantes de CPUs de propósito geral que oferecem versões de CPUs adicionadas a algumas funcionalidades de processadores DSPs, para acelerar o processamento de audio e vídeo, também não pára de crescer [83]. Apesar de incorporarem a funcionalidade de alguns processadores DSPs aos processadores de propósito geral, como instruções para manipulação de som e vídeo, esta união não corresponde aos mesmos ganhos se utilizados separadamente [82].

2.4 Proposta do Trabalho

Cada vez mais estão aparecendo diferentes abordagens de se melhorar o desempenho de sistemas embarcados, o que nos motivou a buscar uma outra alternativa, mais eficiente, para acelerar a execução de programas e que não tenha um impacto considerável no tempo e custo de projeto do sistema. A solução adotada foi propor uma variante da abordagem processador de propósito geral associado a *ASICs*, que reduz significativamente o tempo de projeto, substituindo o uso do circuito de aplicação específica por um co-processador, previamente projetado e verificado, na forma de um *IP*, pronto para ser utilizado em um *SOC*.

Apesar da abordagem da utilização de co-processadores eliminar o problema da dificuldade na reusabilidade de componentes, apresentada por Reinaldo Bergamashi [9] esta apresenta um outro fator mais preocupante. A área ocupada pelo *hardware* extra utilizado para acelerar a execução do programa cresce com o número de funções diretamente implementadas em *hardware* e em muitos casos, devido a requisitos de consumo e custos, é o fator limitante do projeto. Como forma de se enfrentar esse problema propomos neste trabalho um co-processador que utiliza um *hardware* de tamanho fixo e dinamicamente reconfigurável, capaz de executar um grande número de comandos comuns a linguagem de alto nível, incluindo laços do tipo **for**.

Como exemplo de aplicações que podem ser beneficiadas com o desenvolvimento de tal método podemos citar a comparação de cadeias genéticas [50], criptografia [66, 86], processamento de sinais digitais [3, 20] , processamento de imagens [4, 16], entre outros.

Capítulo 3

Ambiente de Desenvolvimento

Neste trabalho foi desenvolvido um Co-Processador Reconfigurável Dinamicamente (*CRD*) capaz de ser configurado para executar um conjunto de funções com o objetivo de acelerar a computação realizada por um processador de propósito geral. Para a prototipagem e implementação do CRD, foi utilizado o kit *EXCALIBUR* da *Altera*, composto por uma FPGA da linha APEX (APEX 20K200E) disposta em uma placa de prototipagem e o processador *Soft-Core* NIOS [18], proprietário da Altera, projetado especificamente para lógica programável.

O Kit EXCALIBUR, possui ainda a ferramenta de projeto (entrada, síntese, simulação etc) *Quartus II*, um compilador C/C++, macro assembler, linker, debugger e bibliotecas de domínio público, que através de um desktop gera código executável para o processador *NIOS*.

O *CRD* foi descrito utilizando-se a linguagem de descrição de hardware *VHDL* (Very High Speed Integrated Circuit **H**ardware **D**escription **L**anguage) [23, 72] sintetizado com o software Leonardo da Mentor Graphics [57] e integrado ao soft-core *NIOS* com o Quartus II da *Altera*. Também foi desenvolvido uma linguagem intermediária denominada *CRDL* para auxiliar o usuário do sistema integrado *NIOS & CRD*, a realizar as modificações no código fonte C/C++ da aplicação, para programar e disparar a execução de uma função na unidade reprogramável *CRD*.

3.1 O Processador NIOS

O processador *NIOS* é um *Soft-Core* de um processador *RISC* [37] de propósito geral, otimizado para lógica programável. O processador *NIOS* possui um pipeline de 5 estágios e pode ser configurado para usar palavras de 16 ou 32 bits. A tabela 3.1 apresenta as principais características da CPU *NIOS* para palavras de 16 e de 32 bits.

Arquitetura da CPU do NIOS

Detalhes	CPU 32 Bits	CPU 16 Bits
Tamanho do barramento de dados	32	16
Largura da ULA	32	16
Tamanho dos registradores internos	32	16
Tamanho do barramento de endereço	33	17
Tamanho de uma instrução	16	16
Células lógicas	1700	1100
F_{max} *(EP20K200E)*	Acima de 50Mhz	Acima de 50Mhz

Tabela 3.1: Características básicas do processador NIOS

O banco de registradores do processador NIOS, pode ser configurado e conter um total de 128, 256 ou 512 registradores. Os programas podem acessar os registradores utilizando uma janela deslizante de 32 registradores.

3.1.1 Periféricos

Na configuração do processador *NIOS*, alguns periféricos podem ser incorporados ao sistema, tais como:

- *Timers* programáveis de 32 bits
- UARTs (Universal Asynchronous Receiver/Transmitter
- DMA
- Controlador de *refresh* de memória dinâmica (Versão II)
- Entradas e Saídas paralelas de 1 a 32 bits
- Interface Serial - "3 wire"
- Memórias RAM e ROM *on chip*

Muitos dos dispositivos são implementados internamente ao processador *NIOS*, e outros necessitam de um barramento para comunicação entre CPU e periférico.

Timer Programável

No presente trabalho, os *timers* programáveis foram utilizados para a medida do tempo de execução das instruções pelo processador *NIOS*.

O timer programável de 32 bits, pode ser controlado através de escritas ou leituras em seus registradores (Tabela 3.2) pelo processador *NIOS*, definindo modos e valores de contagem. O Timer gera um pedido de interrupção que pode ser mascarada por um bit de controle interno.

A1 a A0	Register name	Descrição dos bits do registrador													
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Status															RUN	TO
1	Control													Stop	Start	CONT	ITO
2	Period(L)	Time-out-Period (Bits 15 - 0)															
3	Period(H)	Time-out-Period (Bits 31 - 16)															
4	Snap(L)	Time-out Counter Snapshot (Bits 15 - 0)															
5	Snap(H)	Time-out Counter Snapshot (Bits 31 - 16)															

Tabela 3.2: Descrição dos registradores do "NIOS Timer"

Cada *timer* é gerado por um *MegaWizard Plug-In*[1] na ferramenta *Quartus II*. Um sistema completo pode conter diversos *timers* (limitados pela capacidade da FPGA) mapeados em diferentes endereços, e utilizando diferentes entradas do vetor de interrupção do processador.

O processador *NIOS* executa os seguintes tipos de controle sobre um *timer* reprogramável:

- Carregar o valor da contagem, através da escrita dos registradores *PeriodL* e *PeriodH*;
- Início e parada da contagem pelas escritas de 1s no "start" e "stop" bits no registrador de controle;
- Habilitar ou desabilitar interrupções (bit iTO) no registrador de controle;
- Programar o modo de operação (contínuo ou por pulso) através da escrita do bit "cont" no registrador de controle;

[1] Ferramenta que atua como assistente no software de síntese

Pelo fato do *NIOS timer* poder trabalhar tanto em sistemas de 16 ou 32 bits, todos os registradores internos ao contador possuem largura de 16 bits. Em sistemas de 32 bits é necessário executar duas operações de escritas separadas nos dois registradores de 16 bits (*PeriodL* e *PeriodH*) para a programação de um valor de contagem de 32 bits.

Durante a execução do *"MegaWizard Plug-In"* para a criação do *timer* embarcado, é necessário que se forneça o endereço base, criado para ele e o tipo da interrupção vetorada associada ao mesmo. Estes valores são então utilizados automaticamente, pela ferramenta Quartus II, para gerar o mapeamento dos periféricos e futura utilização via software, através de arquivos em código C (*headers*). Conforme é apresentado na figura 3.1, estipulou-se endereço base para o timer1 em $440h$ e 25 para o tipo da interrupção. Sempre que for necessário a utilização do periférico via software, o nome definido pelo header poderá ser utilizado.

```
1  #define na_uart1          ((np_uart *)  0x00000400)
2  #define na_uart1_irq                    26
3  #define na_timer1         ((np_timer *) 0x00000440)
4  #define na_timer1_irq                   25
```

Figura 3.1: Código resultante do Mapeamento dos principais periféricos utilizados na prototipagem do sistema

O acesso ao conjunto de registradores internos ao timer (status, controle etc) mostrados na tabela 3.2 pode ser realizado utilizando-se a estrutura *np_timer* (codificada em C) mostrada na figura 3.2.

Para determinar a quantidade de ciclos gastos na execução de um trecho de instruções são utilizadas duas funções denominadas *"start_tick"* e *"get_tick"*. A primeira serve para programar o valor inicial do *timer* e disparar sua contagem que é realizada a cada novo ciclo de clock, a segunda bloqueia-o e lê o valor contido em seu registrador de contagem[2] indicando quantos ciclos se passaram desde o seu disparo, retornando este valor por uma variável. Um exemplo de uso destas funções pode ser visto no trecho de código mostrado na figura 3.3.

[2]apontado na estrutura da figura 3.2 como *np_timerperiodl* e *np_periodoh*, onde a primeira significa a parte baixa e a segunda a parte alta de uma variável de contagem de 32 bits

```
1   // Timer Registers
2   typedef volatile struct
3           {
4           int np_timerstatus;  // read only, 2 bits (any write to clear TO)
5           int np_timercontrol; // write/readable, 4 bits
6           int np_timerperiodl; // write/readable, 16 bits
7           int np_timerperiodh; // write/readable, 16 bits
8           int np_timersnapl;   // read only, 16 bits
9           int np_timersnaph;   // read only, 16 bits
10          } np_timer;
```

Figura 3.2: Estrutura para manipulação do timer do sistema

O valor retornado por *"get_tick"* é então subtraído do valor LATENCY, que é o número de ciclos necessários à execução da chamada da função e armazenamento dos valores dos registradores e do endereço de retorno na pilha.. A latência entre a chamada das duas funções é de 24 ciclos de clock.

Para determinar o número de ciclos gastos na execução de um trecho de programa no *NIOS* ou no *CRD*, basta chamar a função *"start_tick"* imediatamente antes do trecho do programa e a funcão *"get_tick"* imediatamente após. A figura 3.4 ilustra o uso de "start_tick" (linha 1) e "get_tick" (linha 4) na determinação do número de ciclos gastos na execução do laço, compreendidos entre as linhas 2 e 3.

3.1.2 Terminal

Através de uma porta serial, configurada como componente, consegue-se além de carregar o software que se deseja executar no processador NIOS, interagir com o processador através de um desktop. Pode-se realizar comandos de entradas e saídas durante a execução do código pelo processador, através de instruções de alto nível como o *printf e getch*, possibilitando o controle e monitoramento do programa que está sendo executado pelo *NIOS* no Kit *EXCALIBUR*.

3.1.3 Placa de Prototipagem

O Kit *EXCALIBUR*, mostrado na figura 3.5, é uma placa de prototipagem composta por:

```
1  void start_tick(np_timer *timer)
2  {
3        timer->np_timerperiodh = PERIOD >> 16;
4        timer->np_timerperiodl = PERIOD & 0xffff;
5        timer->np_timercontrol = (timer->np_timercontrol & 3)
6                               + np_timercontrol_start_mask;
7  }
8
9  long get_tick(np_timer *timer)
10 {
11       long valor;
12
13       timer->np_timercontrol = (timer->np_timercontrol & 3)
14                              + np_timercontrol_stop_mask;
15       timer->np_timersnapl = 0;
16       valor = (timer->np_timersnapl & 0x0ffff)
17               + ((long)timer->np_timersnaph << 16);
18       return PERIOD-valor-LATENCY;
19 }
```

Figura 3.3: Funções destinados a habilitar e retornar o valor do contador

- FPGA APEX 20K200E, dispositivo com 484 pinos encapsulado em BGA (*Ball-grid array*) (Tabela 3.3). O soft-core NIOS, dependendo de quais periféricos são instanciados, quando sintetizados pelos software *Quartus II*, ocupa em média, 25% a 35% dos recursos disponíveis (LEs, ESBs, LUT) desta FPGA.

- Uma placa de desenvolvimento que permite dois métodos distintos para a configuração do dispositivo APEX : A conexão JTAG, que poderá ser utilizada com o software *Quartus II* através do cabo de programação *ByteBlaster*, ou um controlador de configuração que configura o dispositivo APEX quando o mesmo é alimentado, através do arquivo *hexout* armazenado na memória *flash*.
- 1 MByte (512K x 16-bit) em memória *flash*.
- 256 KBytes de SRAM (em 2 chips de 64K x 16-bit).
- Lógica On-Board para configurar o dispositivo APEX a partir da memória *flash*.
- Uma porta serial RS-232.

```
1    start_tick(timer);                  // Begin Profiling
2    for (f=0;f<2;f++)
3         *p_z  += *p_a++ * *p_b++ ;
4   v1 = get_tick(timer);                // End Profiling
5   printf("\nClocks => %ld\n",v1);      // Print
```

Figura 3.4: Utilização das funções de tomada de tempo para um bloco de código

- Quatro botões do tipo *push-botton.*
- 1 Chave do tipo DIP(*Dual in-line package*) de 8 bits.
- 2 displays de 7 segmentos.
- Conector JTAG (*Joint Test Action Group*) para o programador *ByteBlaster.*
- Oscilador de 33Mhz.
- Circuito *Power-on Reset.*
- Circuito regulador de tensão (Entrada 9V não regulada).

Utilização máxima de Portas lógicas	526.000
Utilização típica de Portas lógicas	211.000
LEs	8.320
ESBs	52
Máximo de bits de RAM	106.496
Máximo de macrocélulas	832
Máximo de pinos de I/O	382

Tabela 3.3: Características principais da FPGA APEX20K200E

Memória

Para o desenvolvimento do sistema foi utilizado a memória *SRAM* existente na placa de prototipagem. Por ser uma memória de acesso rápido, as instruções de escritas e leituras efetuadas pelo processador *NIOS* foram configuradas de forma a não necessitarem *Wait States*, tornando possível completar uma leitura ou escrita na memória em apenas um ciclo de clock.

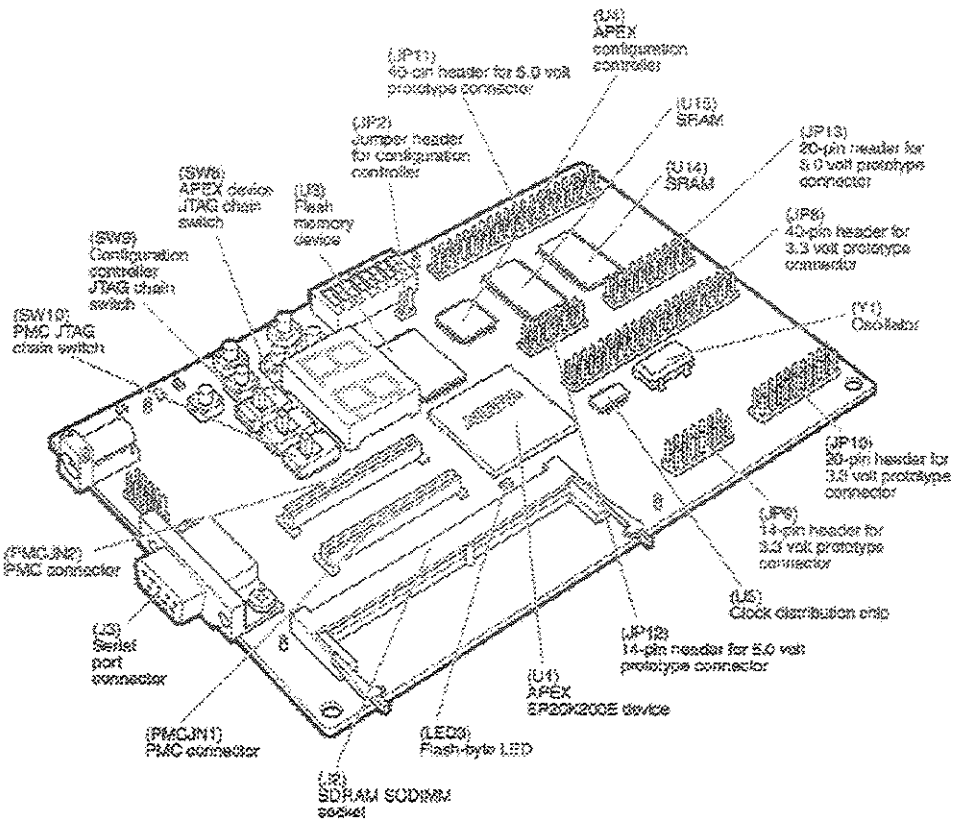


Figura 3.5: Figura ilustrativa do kit *Excalibur*

Existe a possibilidade de se utilizar uma memória tipo *flash*, disponível no KIT caso se deseje manter o código armazenado na placa de prototipagem, caso contrário deve-se carregar o programa a cada execução do mesmo.

Compilador GCC

A escolha do GCC se baseou principalmente nos seguintes fatores:

- **Acessibilidade**: O compilador GCC é parte integrante do Kit Excalibur, ferramenta utilizada para a prototipagem do projeto.
- **Estabilidade** : O compilador já passou por um conjunto suficiente de testes para garantir um padrão mínimo de estabilidade em todos os seus componentes.

- **Portabilidade** : Ainda não existe uma arquitetura reconfigurável padrão no mercado, nem mesmo um processador dominante no mercado de sistemas dedicados. Por isso, precisamos que a ferramenta possa ser facilmente portada para outras plataformas.

Com o uso de um compilador existente para diferentes plataformas e extremamente estável, será possível, no futuro, experimentar o uso do *CRD* com diversas outras CPU's.

Capítulo 4

O Co-processador Reconfigurável Dinamicamente - CRD

Uma técnica que pode ser utilizada para aumentar o desempenho de um dado programa é a utilização de um component de *hardware* especificamente projetado para executar diretamente os trechos do programa responsáveis pela maior parte de seu tempo de execução. Esta técnica pode ser implementada usando uma das duas possíveis abordagens básicas. A primeira consiste em dotar o *datapath* do processador com o *hardware* (e instruções) necessário à execucão de cada trecho crítico. Esta abordagem tem as seguintes restricões: i) O projeto do *hardware* extra depende do processador, podendo ser totalmente diferenciados para processadores de diferentes arquiteturas; ii) O processador deve ser reprojetado e sintetizado; iii) Não pode ser utilizada, por exemplo, em um projeto de um SOC que utiliza um *core* de um processador proprietário. A segunda abordagem, que não apresenta as restricões listadas acima, consiste em projetar um co-processador para executar os trechos de programas, e que seja acionado a partir de instruções já existentes no conjunto de instruções do processador de propósito geral. Esta abordagem tem a vantagem de ser independente do processador, uma vez que o co-processador pode ser, por exemplo, mapeado no espaço de memória do sistema e ser acionado a partir de instruções de escrita e leitura em memória, comuns a todos os processadores de propósito geral.

A solução adotada neste trabalho foi a implementação de um Co-processador Reconfigurável Dinamicamente (**CRD**), mapeado em memória, que pode ser usado com diferentes processadores e é capaz de ser configurado para executar diversos trechos de

programas, incluindo laços inteiros. A figura 4.1 mostra como o *CRD* pode ser utilizado com o processador *NIOS* da *Altera*.

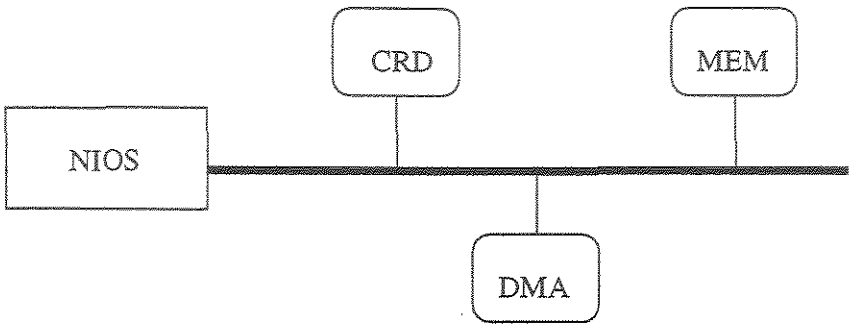


Figura 4.1: Esquema de um sistema baseado no processador *NIOS* usando o *CRD* mapeado em memória

O CRD possui duas faixas de endereços distintas, denominadas endereço de registradores e endereço de programação. A faixa de endereçamento de registradores é utilizada para armazenar os valores a serem operados pelo *CRD*. Estes valores podem ser constantes, endereços de memória ou índices para manipulação de vetores. Nos endereços de programação são armazenadas as palavras de programação responsáveis por definirem as funcionalidades do *datapath* do *CRD*. A figura 4.2 mostra um diagrama dos sinais envolvidos na transferência de controle da execução da CPU *NIOS* para o *CRD*. Uma leitura na faixa de endereços de programação, realizada através dos sinais de leitura (#Readn_PRG), seleção (SEL_PRG) e do endereço da instrução (Addr_in_PRG) faz com o que o *CRD* ative o sinal *SLEEP*, colocando o processador *NIOS* em *HALT*[1]. Desta forma o processador torna-se mestre do barramento e dá início à execução da instrução definida pelas palavras de programação. Após a execução da instrução, o resultado válido é colocado nas linhas de dados (Data_Out) e então o sinal de *SLEEP* é colocado em nível lógico zero, fazendo desta forma, com que o processador *NIOS* assuma o controle do barramento e realize a leitura do valor entregue pelo co-processador *CRD*.

Com o processador de propósito geral em *HALT*, o CRD durante a execução de uma instrução pode usar os valores armazenados em seus registradores internos como operandos ou como endereços para acessos de leitura ou escrita a dados na memória ou outros

[1] Estado de espera

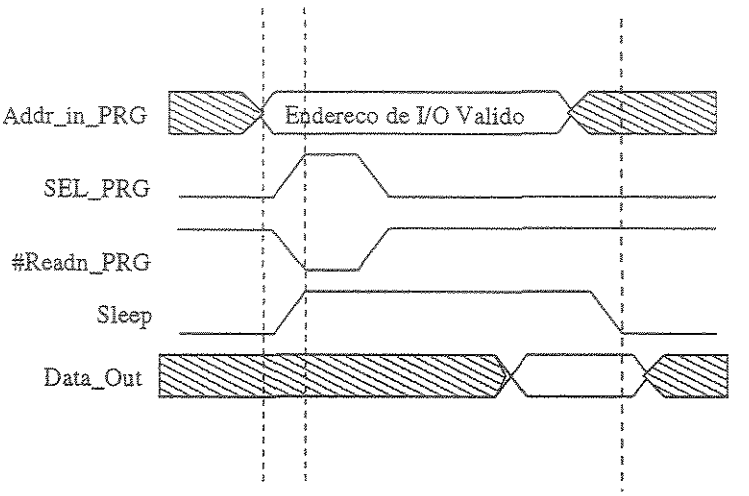


Figura 4.2: Sinais de acionamento do CRD

dispositivos diretamente ligados ao barramento. O processamento a ser realizado é definido através das palavras de programação, previamente escritas na memória interna do CRD (memória de programação) pelo processador, imediatamente antes da execução da instrução (programação dinâmica), ou armazenadas durante a carga do programa (programação estática). O CRD manterá o processador em estado de espera pelo período necessário à execucão da instrucão, determinado nas palavras de programação.

Ao final do processamento o CRD armazena o valor de retorno da funcão computada em um registrador, mapeado no mesmo endereço de leitura da memória de programação que disparou o processamento, e então o CRD desativa o sinal de *SLEEP* tornando novamente ativo o processador de propósito geral. Neste ponto o processador termina a leitura que disparou a execucão da instrucão e obtém um valor retornado pelo *CRD*.

4.1 Implementação do CRD

O *CRD* é composto por 7 blocos funcionais principais, interligados como mostrado na figura 4.3.

O módulo de programacão (BASCELL - compreendido pela linha pontilhada) é o bloco efetivamente responsável pela implementacão de um novo *datapath*. É neste bloco que unidades como deslocadores, registradores, multiplexadores, unidade lógica e aritmética

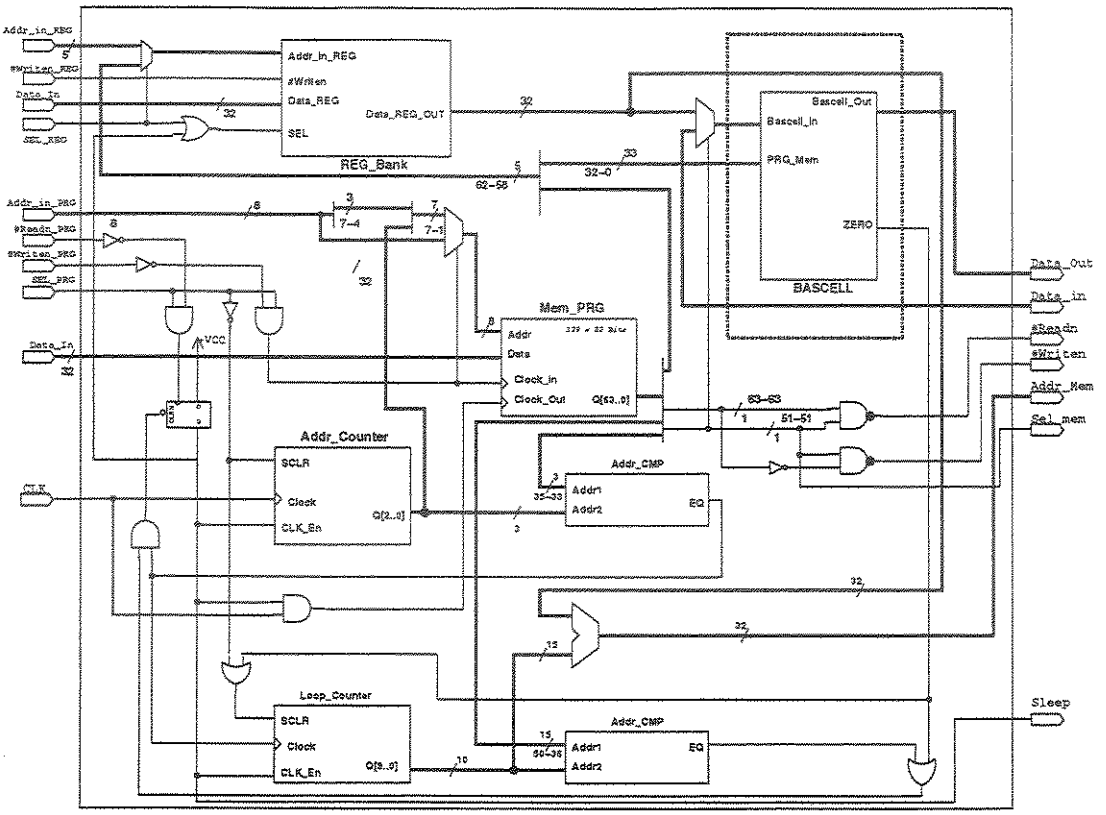


Figura 4.3: Diagrama funcional do CRD

podem ser rearranjadas de forma semelhante a uma PAL[2], para implementarem uma dada funcionalidade. Na configuracão de uma instrucão, a definicão de quais sinais serão entradas de um determinado bloco funcional é realizada por meio de uma rede de multiplexadores, dispostos estrategicamente no *CRD* de forma a permitir diversas possibilidades de interligações entre os blocos funcionais. Os bits utilizados na seleção dos multiplexadores são armazenados na unidade Mem_PRG que é a memória de programacão do *CRD*. As funcionalidades implementadas em cada um dos 7 principais blocos (REG_Bank, Addr_Counter, Loop_Counter, Mem_PRG, Addr_CMP, Loop_CMP e BASCELL) que compõe o *CRD* e mostrados na figura 4.3 estão descritos a seguir.

[2]Programmable Array Logic

4.1.1 Banco de Registradores

O Banco de registradores do *CRD*, denominado no diagrama de blocos funcionais (Figura 4.3), como REG_Bank, é um banco de registradores interno ao co-processador, usado para armazenar valores de operandos ou de endereços de memória (onde se encontram os operandos) para uso pelo *CRD* no processamento da função correntemente em execução. REG_Bank possui 32 registradores de 32 bits e quando usados para armazenar endereços de memória podem ser utilizados como endereços base para acesso a elementos de vetores. Também podem ser utilizados para armazenar valores temporários na execução de uma dada função. A leitura dos dados contidos no REG_Bank só pode ser efetuada internamente ao *CRD* e a escrita pode ser realizada tanto pela CPU, quanto por outros dispositivos externos ao co-processador e pelo próprio *CRD*.

4.1.2 Contador de Endereço de Instrução

O módulo *Addr_Counter* na figura 4.3, é um contador de 3 bits utilizado no controle da execução de uma instrução que pode ter a duração máxima de até 8 ciclos. O endereço da palavra de controle que é usada para configurar o *datapath* em um dado ciclo de execução de uma instrução é aquele usado pela CPU para disparar a execução da instrução (endereço da memória de programação do CRD lido) concatenado com o valor de *Addr_counter*. Desta forma *Addr_counter* contém, para cada ciclo o deslocamento, em relação ao endereço inicial, onde se encontra a palavra de programação do datapath (enderecos múltiplos de 8). A figura 4.4 mostra de forma esquemática como é formado o endereço para acesso à memória de programação do *CRD*, bem como a organização da mesma.

4.1.3 Contador de Laço de Programa

É um contador utilizado na execução de laços pelo *CRD* (Loop_Counter na figura 4.3) cujo valor armazenado, corresponde à iteração corrente. Este valor deve, no fim da execução de cada iteração, ser comparado com o número total de iterações do laço para avaliar se mais uma iteração deve ser executada ou não.

O número total de iterações de um laço é fornecido ao *CRD* durante a programação

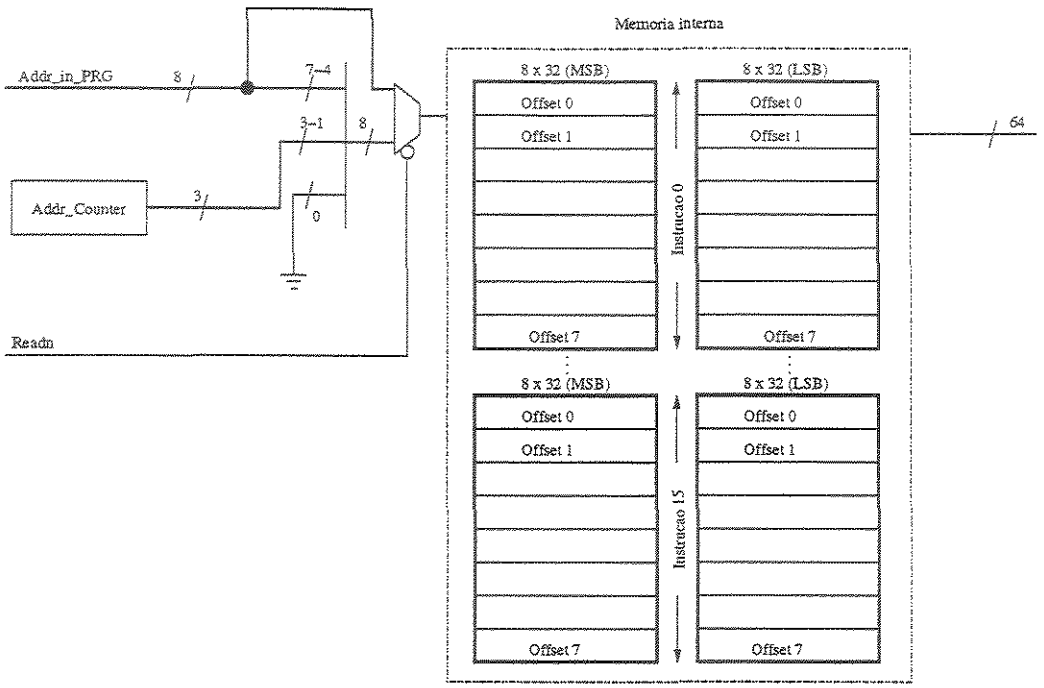


Figura 4.4: Organização da memória de programação interna ao *CRD*.

através das palavras de controle, de acordo com a instrução a ser implementada. Esse valor deverá ser zero para os casos onde a instrução não represente um laço. Na implementação atual, o Loop_counter é um contador de 10 bits, que permite lacos com até 1024 iteracões. A figura 4.5 mostra o comportamento de Loop_Counter para a execução de um laço cuja execução do corpo do mesmo gasta 3 ciclos de clock.

4.1.4 Memória de Programa

Mem_PRG é uma memória, interna ao *CRD*, de 128 palavras de 64 bits onde são armazenadas as configurações de até 16 instruções. Para cada instrução são reservadas 8 palavras, o que possibilita a implementação de instruções que sejam executadas em até 8 ciclos de relógio (1 palavra para cada ciclo). Para manter a compatibilidade com processadores de 32 bits, a Mem_PRG é vista externamente como uma memória de 256 palavras de 32 bits (Figura 4.4), facilitando, desta forma, a escrita das palavras de controle pela CPU. A palavra de memória de endereço interno i (palavra de 64 bits) corresponde às palavras de memória de endereços externos 2_i e $2_{(i+1)}$ (palavras de 32 bits) para $i_{0,...,127}$.

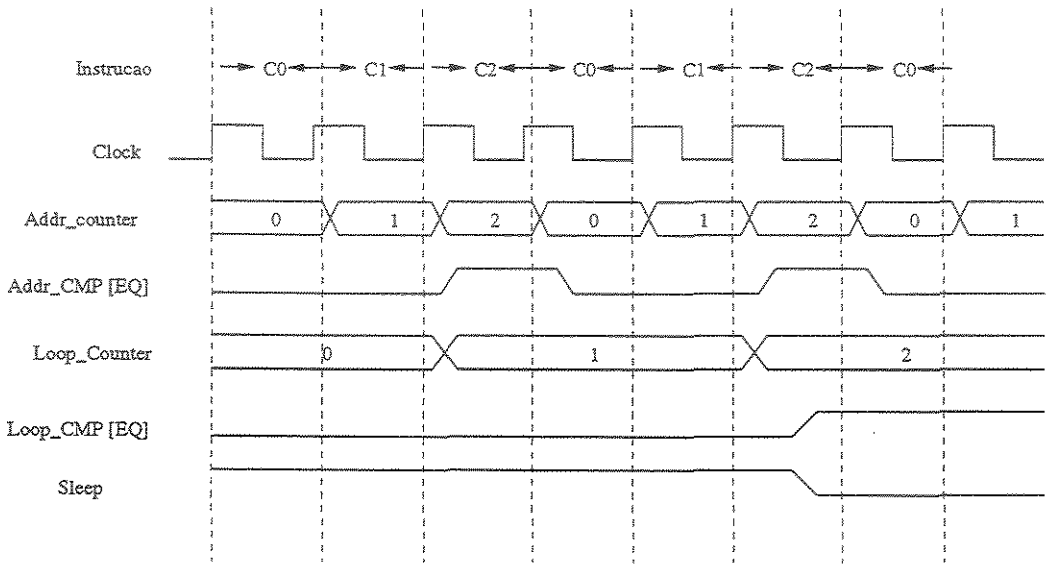


Figura 4.5: Instrução de 3 ciclos de clock, repetida n vezes (no exemplo n é igual a 2).

4.1.5 Comparadores dos Contadores de Instrução e de Laço de Programa

Addr_CMP é um comparador de 3 bits que a cada ciclo de relógio compara o valor do Addr_counter (offset do endereco de memória de programa) com o número de ciclos da instrução, contida na palavra de controle (figura 4.11). O resultado desta comparação é utilizado para terminar a execução da instrução ou para executar mais um ciclo da instrução corrente.

Loop_CMP é um comparador de 10 bits que é utilizado de forma similar à *Addr_CMP*, com objetivo de determinar o término da execução de um laço ou executar mais uma iteração.

4.1.6 Módulo de Programação

O módulo de programação (BASCELL - compreendida pela linha pontilhada na figura 4.3, mostrado na figura 4.6) é composto por um bloco básico e um conjunto de multiplexadores. Estes multiplexadores estão dispostos de forma a tornar possível um grande número de combinações entre os elementos do bloco básico, através de interconexões entre entrada e saída das diferentes unidades do mesmo. Este módulo permite programar um *hardware*

capaz de executar uma função equivalente àquela executada por um pequeno bloco de instruções do processador de propósito geral.

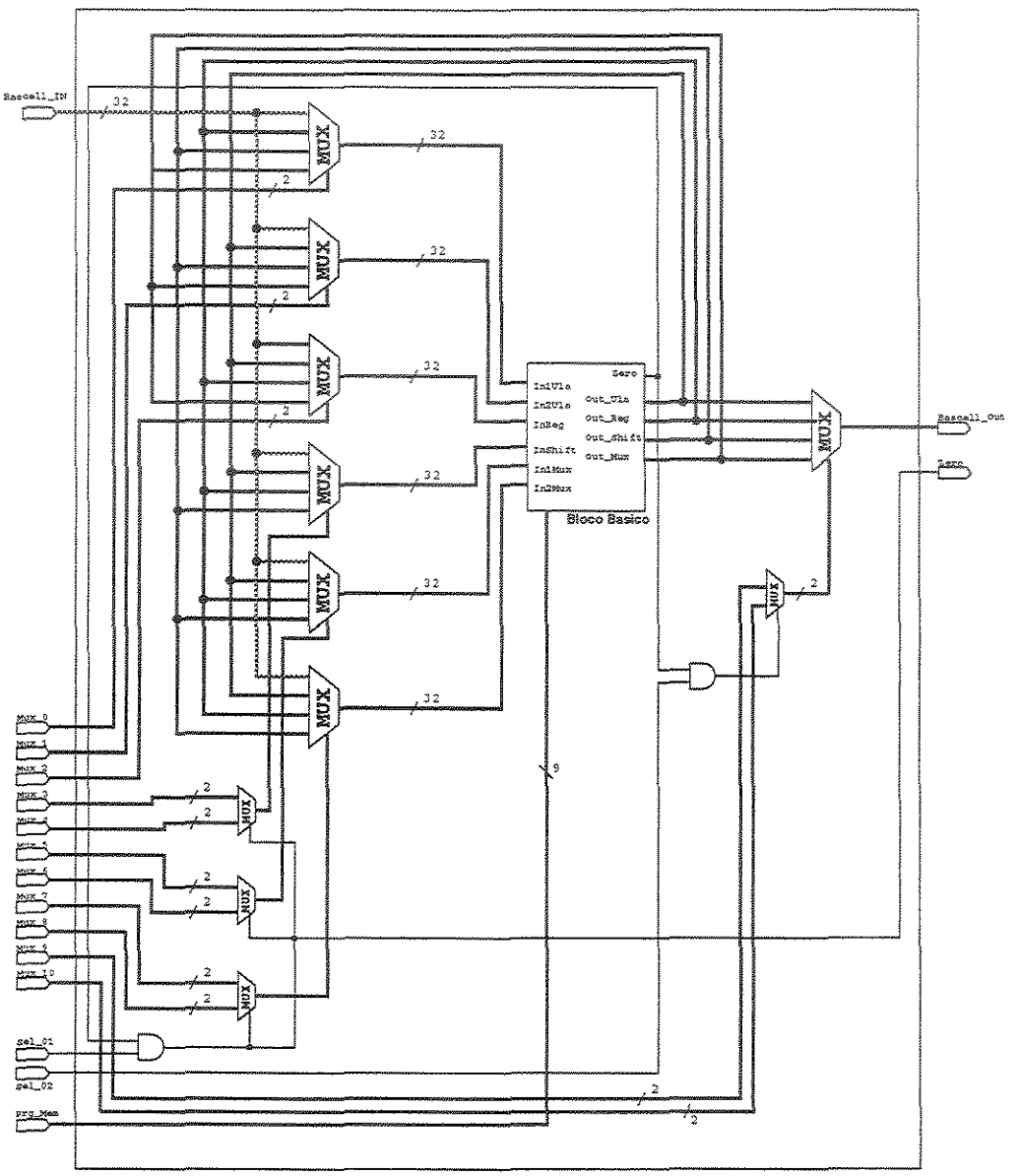


Figura 4.6: Módulo de programação

4.1.7 Bloco Básico

Denomina-se Bloco Básico ao conjunto de unidades funcionais, listadas abaixo, com as quais é possível implementar a maioria das operações que normalmente ocorrem em um

programa típico. A figura 4.7 mostra de forma esquemática o bloco básico.

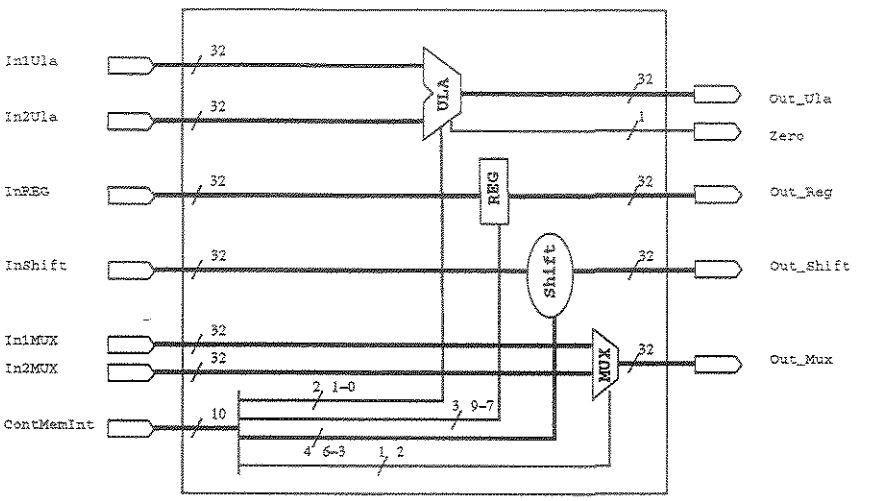


Figura 4.7: Unidades funcionais dispostas no Bloco Básico

São as seguintes as unidades funcionais do bloco básico:

- *Unidade lógica e aritmética* de 32 bits - A *ULA* do CRD é capaz de realizar um pequeno grupo de operações[3], tais como:
  - Soma entre dois operandos;
  - Subtração entre dois operandos;
  - Multiplicação em ponto fixo de dois operandos de 32 bits;
  - Negação - Nega o conteúdo do operando;
  - Compara se os dois operandos são iguais
  - Compara se o primeiro operando é maior que o segundo operando
  - *BIT0* - Retorna o valor do Bit menos significativo (*LSB*)
- *Shift* de 16 bits - Este bloco funcional executa deslocamentos à esquerda de até 16 bits;
- *Banco de registradores* - Com 4 registradores de 32 bits.

[3]É possível a codificacão de mais de 4 instrucões na ULA, com apenas 2 bits devido as saídas independentes ZERO e RESULT onde, por exemplo, em uma operacão de soma é realizado simultaneamente a uma operacão de comparacão.

- *Multiplexador* - Este multiplexador de duas entradas pode ser utilizado para implementar em *hardware*, saltos condicionais.

A palavra de controle do Bloco Básico é composta por um vetor de 10 bits, conforme mostrado na figura 4.8, onde:

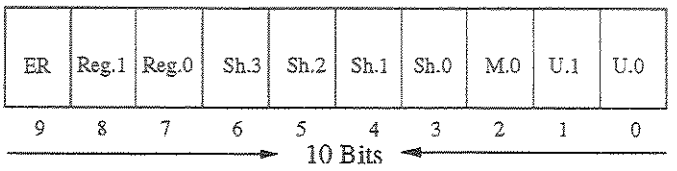


Figura 4.8: Palavra de controle do Bloco Básico

- bits U.1 e U.0 : Seleciona a operação a ser executada pela *ULA*.
- bit M.0 : Define a saída do multiplexador.
- bits SH.0 - SH.3 : Definem o deslocamento a ser realizado pela unidade *Shift*.
- bits R.0 E R.1 : Formam a entrada do decodificador do banco de registradores. Estes dois bits indicam qual registrador interno ao bloco básico está sendo utilizado.
- bit ER : É o bit de *"Enable"* do registrador, escolhido através da palavra *R.0 e R.1*.

Um conjunto de instruções do processador nativo (NIOS) pode ser mapeado em uma instrução multiciclo do *CRD* de até 8 ciclos, onde a operação a ser realizada, em cada ciclo é determinada pela palavra de programação que aciona uma unidade funcional do Bloco Básico (figura 4.8). Pode-se aumentar o desempenho do *CRD* na execucão de um conjunto de instruções do processador nativo mapeando este conjunto em um número menor de ciclos, fazendo com que em um ciclo seja ativado mais do que um bloco funcional do Bloco Básico. Por exemplo, um *shift* e uma operacão da *ULA* que podem ser executados em paralelo, se não houver dependências de dados entre estas funções.

Nos casos onde há dependência de dados, pode-se acelerar a execucão configurando-se as unidades funcionais para executarem em série, onde a saída de uma unidade funcional é a entrada de outra em um próximo ciclo de relógio. Este último tipo de programação requer do programador um conhecimento mais profundo da temporização dos sinais do

hardware (*CRD*) de forma a prover o tempo necessário à execucão de toda a operação, para assim, determinar a freqüência de operação do *CRD*.

O módulo de programação (figura 4.6) possui como entrada, uma palavra de dados de *32 bits* e outra de controle de *33 bits*. A saída é composta por uma palavra de *32 bits*, proveniente de uma das saídas do bloco básico e direcionada, através dos multiplexadores, para a saída da unidade. Ainda faz parte da saída, deste módulo, o sinal *ZERO* que pode ser utilizado como sinal de controle para a unidade de processamento.

A palavra de controle do módulo de programação é composta por *33 bits* (Figura 4.9), distribuídos da seguinte forma :

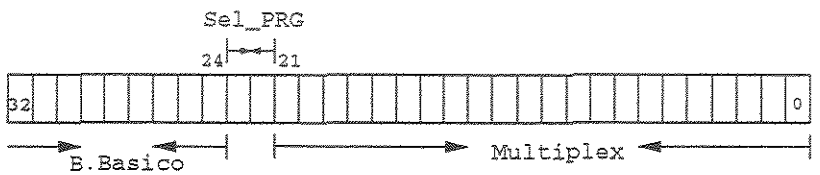


Figura 4.9: Palavra de controle do módulo de programação

- multiplex : Este campo é composto por *22 bits*. É ele quem define a seleção da saída dos 7 principais multiplexadores contidos no módulo de programação. Os campos Mux_{0-2} (apresentados na figura 4.6) mapeiam diretamente 3 multiplexadores de 4 entradas, enquanto que os 16 bits restantes servem como entrada, dois a dois, para multiplexadores menores de 2 entradas, as quais são selecionadas, pelo sinal *"ZERO"* proveniente do bloco básico (figura 4.7).

- Sel_PRG : Este campo é o que define se o sinal de *ZERO*, proveniente do bloco básico deve ser usado para a manipulação dos dados. Através deste campo de controle consegue-se implementar instruções de execução condicionais (*if then else*) e laços.

- bloco básico : Palavra que contém os 10 bits de controle das unidades internas ao bloco básico (figura 4.8).

4.2 Acesso ao CRD

O programador tem acesso ao *CRD* realizando operações de escrita ou de leitura. A operação de escrita pode ser realizada tanto para escrever na memória de programação durante a configuracão de uma nova instrução a ser executada pelo *CRD*, quanto escrever no banco de registradores, permitindo ao programador passar dados, endereços etc, para ser usado na execução das instruções no *CRD*. A operação de leitura é usada para disparar a execução de uma instrução pelo co-processador e obter um valor retornado pela instrução. A instrução que deverá ser executada pelo *CRD* é aquela cuja programação inicia-se no endereço lido.

O endereçamento da memória de programação e do banco de registradores são realizados em dois espaços de endereçamento distintos reservados para uso do co-processador. O espaço de endereçamento reservado à memória de configuração possui 8 bits e o significado de cada bit pode ser visto na figura 4.10.

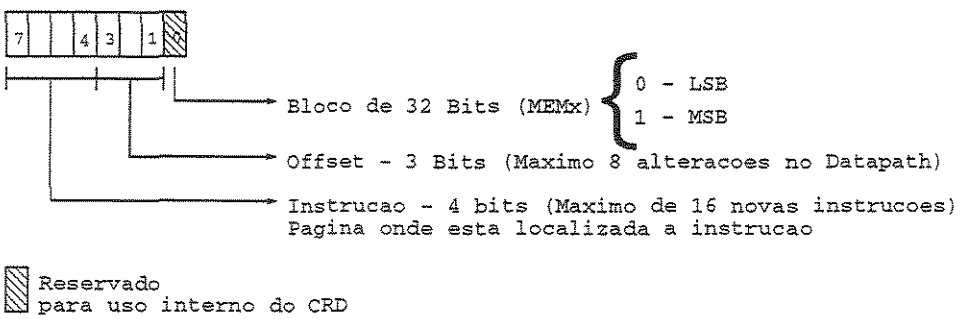


Figura 4.10: Endereçamento da memória de programa.

Em uma operação de escrita (programação de uma nova instrução no *CRD*) todos os 8 bits são utilizados. Como a memória de programação do *CRD* é vista externamente como sendo uma memória de 32 bits, o bit menos significativo é utilizado para indicar a palavra MSB(1) ou LSB(0) de programação, os 4 bits mais significativos endereçam uma das 16 páginas possíveis para programação das instruções e os 3 bits restantes indicam qual palavra de programação está sendo escrita (uma de 8 possíveis). Assim, para a programação de uma instrução que será executada em 8 ciclos é feita com 16 operações de escrita na memória de programação do *CRD*.

Em uma operação de leitura no espaço de endereçamento da memória de programacão,

internamente o *CRD*, só utiliza os 4 bits mais significativos do endereço fornecido pela *CPU*, uma vez que uma programação de uma instrução sempre tem início em um endereço do tipo XXXX0000. Durante a execução da instrução, pelo co-processador, o bit menos significativo permanece com o valor em zero (internamente a memória é vista como sendo de 64 bits de largura) e os outros 3 bits (*offset*) serão fornecidos pelo *Addr_Count*, que contém o valor zero no início da execução de uma instrução no *CRD*.

4.2.1 Palavra de Controle

A palavra de controle, mostrada na figura 4.11, é uma palavra de 64 bits, armazenada na memória de programação do *CRD*, que descreve o comportamento do *CRD* em determinado ciclo de execução do mesmo. Uma nova instrução no *CRD* é programada escrevendo-se de uma a, no máximo, oito destas palavras na memória de programação, onde cada palavra especifica o comportamento do *CRD* em cada ciclo de execução desta nova instrução.

Uma palavra de controle é escrita na memória de programacão do *CRD* por operações de escrita em memória, do processador (NIOS), no espaço de endereçamento destinado ao co-processador. Como a interface externa do *CRD* usa barramento de dados de 32 bits são necessárias duas operações de escritas do processador para gravar uma palavra de controle na memória de programacão do *CRD*.

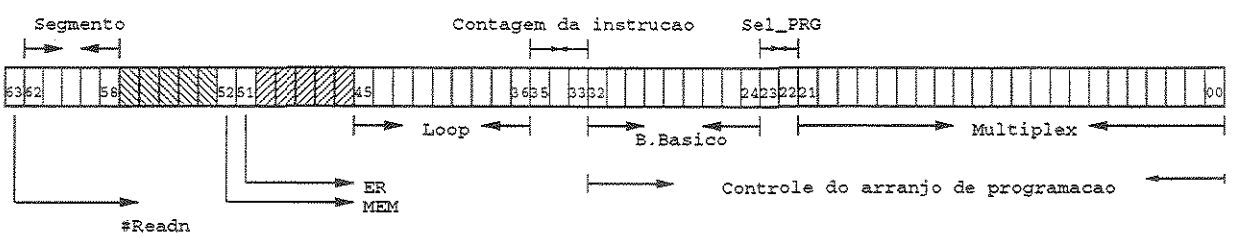


Figura 4.11: Palavra de controle do *CRD*.

Os bits hachurados na palavra de controle (Figura 4.11), compreendidos entre dois intervalos (bits 53 a 57 e 46 a 50) representam, por decisão de projeto, bits reservados ao sistema. Caso haja necessidade de se realizar uma expansão quanto ao número de bits de controle, utilizando como exemplo o acesso a dispositivos externos, ou aumento de tamanho da contagem máxima de um laço, estes bits podem ser utilizados. Os 33

bits menos significativos da palavra representam a palavra de controle do módulo de programação (Figura 4.9). A função dos demais bits está relacionada abaixo:

- CONTAGEM DA INSTRUÇÃO : Os 3 bits referentes a "contagem da instrução" indicam quantas vezes o *datapath* do *CRD* será reprogramado, ou seja, quantas palavras do controle são usadas para especificar a instrucão, ou ainda quantos ciclos serão necessários à execução da instrução no *CRD*.

- LOOP : quando os 10 bits (36-45) deste campo estiverem preenchidos com zero, indicam que a instrução não é um laço e deve ser executada uma única vez, se diferente de zero indica que a instrução é um laço e deve ser executada $n+1$ vezes, possibilitando a implementação de laços em até 2^{10} iterações. Laços com um número de iterações maiores do que 1024 iterações podem ser implementados quebrando sua execução em diversas vezes, ou modificando-se o *CRD* para usar os bits reservados para aumentar o número de iterações possíveis.

- MEM : O sinal MEM bit 51 da palavra de controle, indica se um determinado valor será acessado a partir de algum dispositivo externo ao *CRD*, por exemplo, a *memória*.

- SEGMENTO : O segmento (bits 58 a 62) determina qual registrador (do banco de registradores do *CRD*) contém o endereço base de determinado operando a ser buscado ou gravado em uma memória externa. Pelo fato do banco de registradores possuir 32 registradores, esta palavra é de 5 bits. A utilização de segmentação é muito útil quando se trabalha com manipulação de vetores ou dados seqüenciais.

- READN : O bit 63 da palavra de controle atua em conjunto com o bit 51(MEM). Este bit, determina se a operação a ser realizada pelo *CRD* na memória será uma escrita (READN=1) ou uma leitura (READN=0)

4.2.2 Total de Recursos Utilizados

Buscando melhor desempenho ao invés de melhor aproveitamento da área, foi reportado pela ferramenta de síntese que a implementação do *CRD* compreende aproximadamente

26% dos recursos de um dispositivo da família APEX (EP20K200E). Tornaram-se ocupados 2236 dos 8320 elementos lógicos, assim como 4352 de 106496 bits de RAM, contidos na FPGA.

Capítulo 5

Linguagem CRDL

Neste capítulo será apresentada uma linguagem (pré-processada) para a arquitetura *CRD*, cujo objetivo é auxiliar o desenvolvedor a programar (configurar) e executar instruções no *CRD*.

A programação de uma instrução, a ser executada no co-processador *CRD*, envolve a manipulação direta de bits na construção das palavras de controle e escritas em endereços pré-fixados, no espaçamento de memória do *NIOS* (endereços da memória de programação do *CRD*). A execução destas instruções envolve escritas nos endereços pré-fixados do banco de registradores do *CRD* e leituras no endereço de programação. Embora possa-se rearranjar os bits da palavra de programação do CRD manualmente, esta tarefa pode vir a tornar-se tanto complexa quanto demorada. Na grande maioria dos casos, um tempo maior é despendido na elaboração da seqüencia de bits que devem ser incorporados à palavra de programação do que propriamente ao projeto do datapath, o que pode reduzir o tempo para explorar diferentes abordagens e alternativas para uma melhor solução do problema.

Sem a utilização de ferramentas que automatizem o ambiente de desenvolvimento, o processo de projeto é muito suscetível a erros e pode levar a inconsistências entre as representações de software e *hardware*.

Os esforços para o desenvolvimento de uma nova arquitetura podem ser significativamente reduzidos pela introdução de uma ferramenta baseada na transcrição de uma descrição do *hardware* para novos conjuntos de instruções. A CRDL (*CRD language*) foi desenvolvida com o intuito de auxiliar o desenvolvedor na tarefa de transformar os blo-

cos de instruções pouco eficientes em uma nova instrução a ser executada em *hardware*, de forma rápida e eficiente. Uma linguagem intermediária pré-processada [62, 88] é responsável por gerar as palavras de controle necessárias para a reconfiguração, inicialização e chamada da nova instrução que terá sua execução desviada para o *CRD*. As palavras de controle são enviadas ao co-processador, como utilizado em [5, 70, 71], através de instruções de escrita e leitura em memória.

Uma das características principais da linguagem é o controle de passo por ciclos de relógio, tornando possível um melhor controle sobre as unidades do co-processador e sobre a descrição a cada ciclo da instrução. Cabe ainda a descrição do novo conjunto de instruções, o comportamento e verificação de inconsistências, como tempo máximo do caminho crítico[1] e utilização de unidades indisponíveis, uma vez que o programador tem total liberdade no rearranjo das unidades do co-processador.

5.1 Utilização da *CRDL*

Durante a compilação de um código fonte, o compilador utilizado para gerar o código objeto do sistema *NIOS & CRD*, deverá ser capaz de gerar dois tipos distintos de código. Um deles referente a execução no processador de propósito geral. De outro lado, é necessário a elaboração do código necessário à configuração, início e leitura dos resultados das partes do programa que serão executadas no *CRD*.

Um compilador C/C++ convencional pode, facilmente, gerar o código objeto referente as partes do programa que serão executadas no processador de propósito geral, porém não são capazes de gerar o código extra de configuração e controle necessários à transferência de execução do processador para o *CRD* e vice-versa. Pelo fato do *CRD* ser mapeado em memória, a tarefa de configuração e controle pode ser realizada através de escritas e leituras em memória. Assim se as tarefas de configuração e controle forem explicitadas na forma de escrita e leitura de memória no código C/C++, um compilador convencional poderá ser usado para gerar o código executável final.

A cada ciclo de execução do *datapath* que será executada em *hardware*, são necessários

[1] Para a síntese realizada em bancada o caminho crítico sempre se manteve abaixo do período de execução ao qual o processador estaria sendo submetido

64 bits para sua programação. A fim de facilitar a tarefa do usuário na configuração do *CRD*, a (*CRDL*) é capaz também de encapsular a complexidade de se manipular esta quantidade de bits em estruturas de alto nível.

Um compilador C/C++ típico, além das fases de análise sintática, semântica, geração e otimização de código comuns a maioria dos compiladores, possui ainda uma fase inicial denominada de pré-processamento cuja finalidade principal é a substituição de macros e a inclusão de arquivos de definições.

Durante a elaboração da linguagem, questionou-se sobre a inclusão de uma extensão a linguagem C/C++, para o pré-processamento. Uma forma viável de diferenciação entre os dois códigos de um programa fonte, seria a utilização de símbolos identificadores, que determinassem o início e fim dos blocos da nova linguagem. Estabeleceu-se a diretiva #CRDL para este propósito. Assim sendo, um programa desenvolvido para este sistema conterá o código C/C++, para o qual será gerado código que executará no processador de propósito geral, e um código *CRDL*, destinado a elaboração das palavras de programação e controle do *CRD*, para que o mesmo execute diretamente em *hardware* uma dada função (em geral um loop interno do código original). A este programa contendo comandos em C/C++ e comandos *CRDL* denominamos código misto. Um exemplo de código misto, onde chamadas a duas instruções (mult e loop32), previamente programadas na memória interna do *CRD* são realizadas pode ser observado na figura 5.1.

```
1   for (i=0;i<10;i++)
2   {
3     #CRDL
4         A = mult;
5     #CRDL
6         B = B[i]+A;
7   }
8   #CRDL
9     C = A * Loop32;
10  #CRDL
```

Figura 5.1: Exemplo de código misto submetido ao pré-processador da *CRDL*

O novo pré-processador quando recebe como entrada um código misto, tem a tarefa

de transformar o interior dos blocos demarcados pela diretiva #CRDL no seu equivalente C/C++, composto basicamente por leituras e principalmente escritas em determinados endereços específicos de memória. O próximo passo é então compilar o código retornado pelo pré-processador, totalmente em C/C++, de forma que o programa executável gerado possa ser executado no processador de propósito geral.

Neste trabalho foi utilizado como compilador C/C++ o compilador *GCC* [32] para o soft-core *NIOS*, distribuído pela empresa *ALTERA*.

A extração dos blocos *CRDL* do código misto, delimitados pela diretiva #CRDL, é realizada primeiramente pelo apontamento do lugar do código onde cada bloco foi extraído, para que após uma análise sintática e semântica do código retirado, o bloco retirado venha a ser substituído por instruções de escrita e leitura codificados na linguagem C. Pode-se enumerar a seguir as seguintes fases do pré-processamento da linguagem *CRDL*:

1. **Separação do código *CRDL*** : Todos os blocos que contém a diretiva *#CRDL* são copiados para um arquivo temporário e anotado sua localização no código misto.

2. **Verificação do código** : Análise léxica e sintática da linguagem *CRDL* e geração das palavras de configuração do *CRD* através de instruções de leitura e escritas em memória.

3. **Substituição do código *CRDL* pelo equivalente em C**: A substituição dos blocos *CRDL* do código misto por seu equivalente na linguagem C pode ser feito de duas formas. Uma delas é a substituição do código no ponto em que o trecho em linguagem *CRDL* foi retirado, ou pela disposição do código traduzido em uma biblioteca para que as escritas das palavras de configuração das novas instruções em *hardware*, sejam realizadas antes da execução do código final, definindo uma programação estática de instruções. Desta maneira a configuração de todas as novas instruções em *hardware* são realizadas na carga do programa, antes do início da execução do código do programa, permitindo que um número limitado de instruções, pela capacidade de memória interna do *CRD*,seja programado para sua execução. A outra, pela substituição anotada, a configuração da instrução no *CRD* é realizada no instante imediatamente antes de sua execução, configurando-se uma programação

dinâmica, permitindo que um número ilimitado de instruções seja programado para execução no *CRD*.

A principal diferença entre a programação estática e a dinâmica, além do número de instruções possíveis de serem programadas é o desempenho obtido na execução de cada instrução pelo *CRD*. O desempenho na programação estática, para cada uma das instruções, é substancialmente maior, uma vez que o *overhead* de programação, e desvio da execução do processador para o *CRD* é constituído somente pela execução das instruções de chamada para a execução em *hardware*, enquanto que na programação dinâmica o *overhead* é constituído pelo tempo gasto para a escrita da programação do *CRD* e pelo tempo necessário ao se disparar a execução da instrução no *CRD*.

Devido ao fato da memória de configuração do *CRD* ter tamanho limitado (máximo de 16 instruções), quando se escolhe a abordagem de substituição de código por inclusão de bibliotecas, uma maneira eficiente de se alocar as instruções na memória disponível no *CRD* é requerido. Em geral, este modo é utilizado para realizar a configuração das instruções mais executadas no *CRD*.

5.2 Codificação da Linguagem

A linguagem *CRDL* é uma variante da linguagem de programação C, por este motivo, acessível a maioria dos desenvolvedores. Com o intuito de possibilitar a exploração entre vários tipos de granularidade (nível de laço ou de instrução), ela permite manipular instruções simples ou multi-ciclo com a limitação máxima de 8 ciclos por instrução. O programador é o responsável direto por definir a quantidade de ciclos que uma instrução irá gastar. A palavra reservada *clock* indica ao pré-processador que as configurações necessárias para um ciclo foram terminadas e então ele passa a criar as palavras de programação para a configuração do *CRD*.

Apesar da limitação de projeto, de 8 ciclos de clock por instrução, é possível expandir o número de ciclos, para um determinado *datapath*, através da criação de duas instruções seqüenciais, onde a segunda seja chamada imediatamente após a execução da primeira, dando a impressão de que apenas uma instrução é executada. Obviamente, ganha-se uma

penalidade neste processo imposta pelo segundo overhead de chamada da instrução, que embora mínimo, existe.

5.2.1 Manipulação de Dados em Memória

A linguagem *CRDL* permite ao programador fazer uso das variáveis, ou seus respectivos endereços, utilizados no corpo do programa C/C++. Observe o código apresentado na figura 5.2. A segunda coluna da linha 9 indica que o valor da variável *A* deve ser passado em tempo de execução para o registrador zero do *CRD* (tratado como *overhead* de inicialização).

5.2.2 Criando uma Instrução no *CRD*

Para a configuração de uma nova instrução em *hardware* a palavra reservada *inst* deverá ser usada, como podemos ver no exemplo da figura 5.2, linha 2. A palavra reservada *inst* deve ser seguida pelo nome da instrução que será criada e o conteúdo delimitado por "{" e "}" pertencerá a sua execução.

5.2.3 Realizando a Chamada da Nova Instrução

A execução de uma instrução criada a partir da *CRDL* se dá toda vez que o nome definido através da palavra reservada *inst* for invocado. O valor de retorno da execução da instrução pode ser carregado em uma variável definida no código C/C++ no qual o código *CRDL* está inserido, conforme pode ser visto nas linhas 10 e 12 do código apresentado na figura 5.2.

Paradigma Orientado a Ciclo

O código mostrado na figura 5.2, explora a granularidade por instrução. A linguagem usa a palavra chave *clock* para indicar a transição do *datapath* corrente para o próximo (Figura 5.5), ou diferentes estágios de um pipeline (mudança na palavra de programação). Este pequeno pedaço de código quando submetido ao compilador *CRDL* gera como saída duas palavras de programação de 64 bits que rearranjam as unidades funcionais do *CDR* de forma a funcionar como o desejado. Estas palavras geradas pelo pré-processamento

```
1   A = A * A;
2   B = B * B;
3
4
5
6
7
8
9
10
11
12
13
```

(a)

```
#CRDL
inst mult1
{
   reg.1 <- regin.0;
   clock;
   reg.0 <- reg1 * regin.0
   clock
}
regin.0 <- A;
A = mult1;
regin.0 <- B;
B = mult1;
#CRDL
```

(b)

Figura 5.2: Codificação de uma nova instrução na linguagem *CRDL*. (a) codificação C original; (b) codificação mista *CRDL* e C

são gravadas na memória de programação do co-processador através de duas escritas de 32 bits.

A linguagem *CRDL* aponta possíveis incoerências ou limitações de tarefas por ciclo. Caso se resolva trocar a linha 6 com a linha 5 do código apresentado na figura 5.2, no pré-processamento do código, será detectado a impossibilidade de se realizar 2 leituras de 32 bits em um mesmo ciclo.

5.3 C como Linguagem de Programação do CRD

O programador, se desejar, pode fazer uso apenas da linguagem C/C++ para definir novas instruções em *hardware*, bem como realizar chamadas e armazenar dados nos registradores internos do *CRD*, para isto, as palavras de programação devem ser manipuladas bit a bit e então armazenadas nos endereços nos quais a memória do *CRD* está mapeada. Da mesma forma que valores que deverão ser manipulados como parâmetros pela instrução devem ser armazenados nos endereços do banco de registradores do co-processador. Uma leitura no endereço base da instrução dispara a execução da instrução e retorna um valor, em uma variável definida no programa fonte, de forma análoga à chamada de uma função

em C/C++.

O código apresentado na figura 5.3 é a saída do pré-processamento do bloco mostrado na figura 5.2. Na linha 1, CRD_REG contém o endereço base do banco de registradores do *CRD*, desta forma está sendo realizado uma escrita da variável "A" (definida no código C/C++) no primeiro registrador do banco. Na linha 3 do código, a variável MULT contém o endereço de gravação dos dados de reconfiguração do *datapath* para a realização da instrução *mult*.

Na linha 9 é realizada a primeira chamada à execução da instrução MULT, sendo o valor de retorno armazenado na variável "A", declarada no código C/C++. O Valor de INST define o endereço base de leitura da memória de programação do co-processador. Os endereços bases são deslocados de 16 palavras de 32 bits (máximo de 8 modificações no datapath, uma palavra de 64 bits por modificação), logo INST poderá ter 16 valores inteiros positivos diferentes, onde cada um representa o número da instrução armazenada na memória de programação do *CRD* (de zero a quinze).

Na linha 11 a variável "B" é armazenada no endereço base do banco de registradores do *CRD* a fim de que este valor seja utilizado pela instrução MULT e possa ser reaproveitada para a multiplicação em *hardware* desta variável. Finalmente a linha 13 realiza a execução da instrução MULT armazenando o resultado na variável "B".

```
1   *CDR_REG = A;
2
3   *MULT1++ = 0x00100017;
4   *MULT1++ = 0x00000000;              // Primeiro ciclo
5
6   *MULT1++ = 0x00100023;
7   *MULT1   = 0x00000100;              // Segundo ciclo
8
9   A = *(CDR_PRG+INST);
10
11  *CDR_REG = B;
12
13  B = *(CRD_PRG+INST);
```

Figura 5.3: Código na linguagem C, gerado pelo pré-processamento da *CRDL*

Pequenas estruturas, laços e desvios condicionais também são possíveis de serem implementados no *CRD* usando-se a linguagem *CRDL*. Um exemplo de como obter um melhor desempenho, com a implementação de laços, é mostrado pelo código apresentado na figura 5.4.

```
1   for(i=0;i<N;i++)
2   {
3      B[i] = B[i] * B[i+1];
4   }
5
6
7
8
9
10
11
12
13
14
15
16
17
```

(a)

```
#CRDL
regin.0 <- *B++;
regin.1 <- *B;
inst loop1
{
    for (i=0;i<N;i++)
    {
       reg.1 <- regin.0[i];
       clock;
       reg.0 <- reg.1 * regin.1[i];
       clock;
       regin.0 <- reg.0
       clock;
    }
}
temp = loop1;
#CRDL
```

(b)

Figura 5.4: Código *CRDL* para implementação de um laço. (a) código C original; (b) código *CRDL*

De acordo com a figura 5.4, o *datapath* criado através da *CRDL* se modifica 3 vezes para executar cada uma das iterações do laço, cada modificação é delimitada pela palavra chave *clock*. No primeiro ciclo da instrução o valor contido no endereço de memória apontado pelo registrador de entrada 0 (regin.0) é carregado em um registrador interno (reg.1). No segundo ciclo, é executado uma leitura do conteúdo apontado pelo registrador de entrada 1 (regin.1) que é multiplicado pelo valor lido no ciclo anterior com o valor armazenado no registrador interno *reg.1*. O terceiro ciclo realiza uma escrita do valor obtido na multiplicação na memória do sistema (regin.0). Pode-se dizer, desta forma, que

a nova instrução, que executará todo o laço terá um CPI[2] de $3 * n$, onde n é o número de iterações do laço, (*contando somente a execução da nova instrução*).

A variável *temp* linha 16 da figura 5.4 recebe o conteúdo de retorno do *CRD*, que neste caso não é o valor calculado pela instrução, uma vez que os valores calculados foram escritos, pelos *CRD*, diretamente na memória do sistema. Entretanto, é preciso que alguma variável receba o valor de retorno da instrução, uma vez que ela é invocada de forma análoga à uma chamada de função C/C++.

5.4 Caminho Crítico da Implementação

Conforme mencionado anteriormente, o pré-processador *CRDL* faz uma análise do caminho crítico de cada configuração do *datapath* elaborado pelo desenvolvedor. Para que essa análise seja possível de ser realizada deve ser fornecido ao pré-processador os tempos de atraso de cada recurso disponível no bloco básico (figura 4.7). Estes atrasos podem ser obtidos a partir do relatório de síntese. Com este recurso, torna-se mais fácil ao desenvolvedor a tarefa de programar o *datapath* do *CRD* para que mais do que uma operação, disponível no bloco básico, possa ser executada em seqüência em um mesmo ciclo de clock. Este tipo de execução é realizada programando-se para que uma unidade do bloco básico utilize como entrada a saída de outra unidade tornando este cascateamento limitado pelo tamanho do período de clock utilizado, bem como pelos atrasos das unidades e disponibilidade de recursos no bloco básico e no módulo de programação (figura 4.6)

Para a verificação do período necessário para a execução de um ciclo que uma instrução que deve ser executada em *hardware*, é utilizado um método bem simples. Após a montagem da palavra de controle (Figura 4.11) que determina um ciclo da instrução, pesos são atribuídos a cada um dos bits responsáveis pela interligação do módulo de programação e que são representados por esta palavra. Os pesos utilizados para a computação do período do caminho critico são armazenados em um arquivo que contém os atrasos de todas unidades (assim como tempo necessário para leitura e escrita em memória e freqüência de operação do *CRD*). A soma dos pesos (atrasos das unidades arranjadas seqüencialmente), não deverá possuir período maior que aquele utilizado pelo co-processador.

[2] *clocks* por instrução

5.5 Latência de Leitura e Escrita

Um *datapath* equivalente à linha 1 apresentado na figura 5.2 pode ser visto na figura 5.5. O novo *datapath* equivalente a operação em *software* necessita dois ciclos de *clock* para ser executado, o primeiro denotado por Δ_1 e o segundo denotado por Δ_2. No primeiro ciclo uma leitura do operando é realizada juntamente com sua multiplicação, no segundo ciclo o valor calculado é armazenado no endereço do operando.

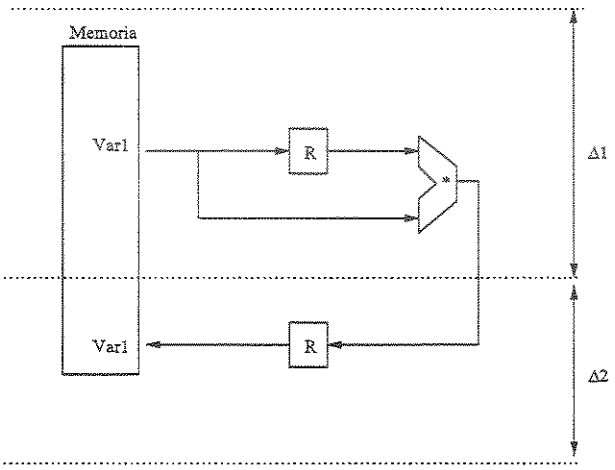


Figura 5.5: *Datapath* para realizar a operação de multiplicação de uma variável

Embora haja substancial redução no tempo de execução da nova instrução deve-se mencionar que o tempo que foi despendido para a gravação do comportamento do *datapath* ainda não foi levado em conta.

Neste exemplo, para se gravar o comportamento do novo *datapath* foram necessárias 4 escritas em memória, duas para cada arranjo das unidades contidas no *CRD*. Para que a instrução seja executada é necessário, ainda, que os endereços de memória dos operandos a serem utilizados pela instrução sejam passados ao *CRD* (escritos no banco de regsitradores REG_mem).

O tempo necessário para a escrita das palavras de programação, juntamente com os endereços *base* é denominado *Overhead de programação* (Figura 5.6).

Para o cálculo do *CPI* de uma nova instrução, os tempos gastos na programação, execução e tomada dos resultados, devem ser computados. Um fator importante que não pode ser deixado de lado é o número total de chamadas que o processador de propósito

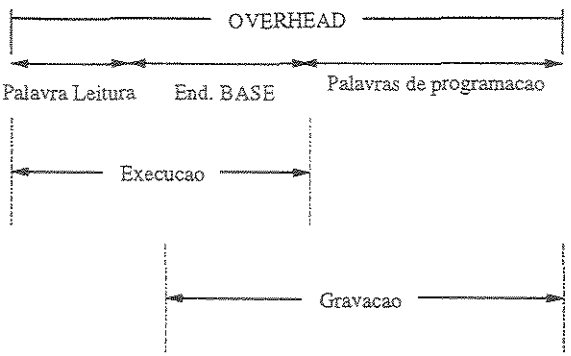


Figura 5.6: Overhead de leitura e gravação de instruções no *CRD*

geral fará para a nova instrução. Um maior número de chamadas representa uma melhor função de amortização do *overhead* de gravação às chamadas, diminuindo o tempo médio da instrução.

O gráfico apresentado na figura 5.7 representa N chamadas do bloco de instruções "A = A * A", em *hardware* e em *software*. Pelo fato do *overhead* de programação ser computado apenas uma vez, a tendência é que as curvas se distanciem cada vez mais de acordo com o crescimento de N, isto é, quanto maior o valor de N, o *CPI* da instrução que está sendo executada em *hardware* tende a ficar menor, melhorando com isto, o desempenho do sistema.

5.5.1 Overhead em Operações com Vetores e Matrizes

A análise dos CPIs estende-se um pouco mais a dados em posições contíguas de memória, como vetores e matrizes. A linguagem *CRDL* permite estruturas simples de laços que, quando bem utilizadas, fazem o *overhead* de programação ou de leitura diminuírem consideravelmente, aumentando desta forma o desempenho da nova instrução a ser executada em *hardware* pelo CRD.

O trecho de código apresentado, através da linguagem *C*, na primeira parte da figura 5.8 pode ser reescrito através da linguagem *CRDL* de forma que possa ser possível sua total execução em *hardware*, conforme é apresentado na segunda parte da figura.

O código do laço apresentado na primeira parte da figura 5.8 quando implementado através da solução por software no *NIOS*, não possuiu custo linear ao número de elementos do vetor, isto se deve ao controle, incremento e comparação com o valor final do laço.

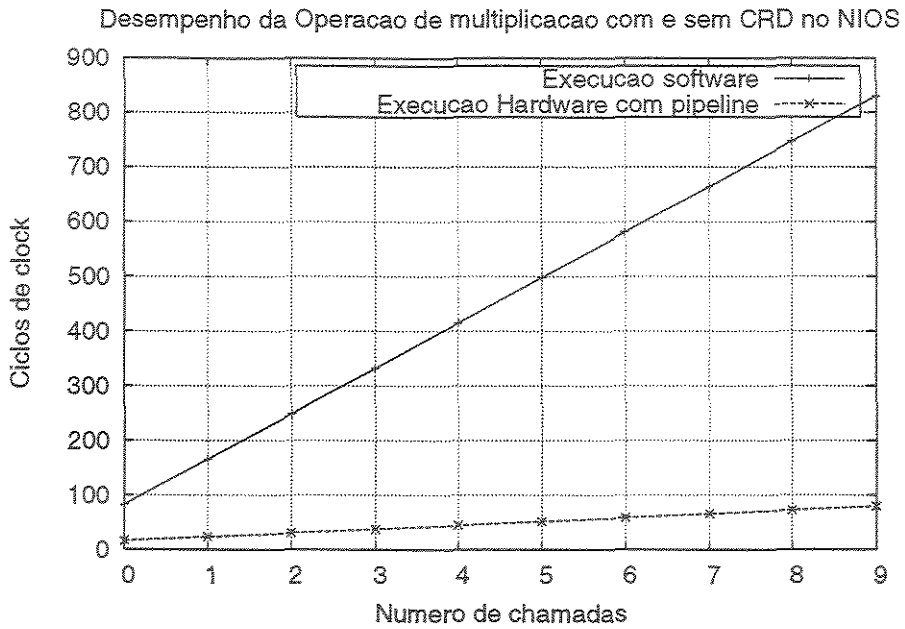


Figura 5.7: Ciclos necessários para a execução do bloco de código "A = A * A" em *hardware* e *software*

O mesmo loop quando implementado através da linguagem *CRDL* apresenta o custo linearizado em função do número de elementos do vetor, atingindo ganhos de desempenho ainda superiores àqueles encontrados com a chamada de n execuções da instrução.

```
1   for(i=0;i<N;i++)
2   {
3       BASE_X[i] = BASE_X[i]*BASE_X[i+1];
4   }
```

```
1   for(i=0;i<N;i++)
2   {
3      reg.0[i] <- regin.0[i]*regin.1[i];
4   }
```

Figura 5.8: Trecho de código responsável por estrutura simples de laço na liguagem C e sua transcrição para a linguagem CRDL

Capítulo 6

Programação de Instruções no *CRD*

A estrutura do *CRD* nos permite flexibilidade na configuração de um novo *datapath*, dedicado à execução de um trecho de programa codificado em uma linguagem de alto nível. Trechos de programas que utilizam estruturas simples, pequenos blocos de código e laços internos, são possíveis de serem executados diretamente em hardware utilizando-se o *CRD*, cabendo ao programador as tarefas de especificar as palavras de controle que irão configurar cada ciclo de execução do novo *datapath* para cada trecho de programa e ainda, inserir no código da aplicação os comandos para disparar a execução de cada um destes trechos. Estas tarefas podem ser realizadas com o auxílio da *CDRL*, visto no capítulo anterior.

6.1 Particionamento Hardware/Software

eralmente a primeira tarefa a ser realizada, quando da utilização de sistemas de processadores de propósito geral e um hardware específico, é determinar a porção da aplicação que será executada no processador de propósito geral (em *software*) e a porção que será executada pelo *hardware* específico (diretamente em *hardware*). Esta tarefa é conhecido como particionamento *Hardware/Software* (HW/SW). Uma forma de se realizar o particionamento HW/SW é implementar toda a aplicação em software e realizar um *profiling*[1] da execução do sistema, utilizando-se ferramentas como *gprof* [32], *ATOM* [77] ou *Spix* [17], determinando assim, os trechos da aplicação que são responsáveis pela maior

[1] levantamento das características de execução de determinado programa

parte do tempo de execução. Em seguida, analisa-se as possibilidades de implementação destes trechos diretamente em hardware. Uma vez realizado os passos acima resta sintetizar o *hardware* capaz de executar os trechos da aplicação selecionados e modificar o código original da aplicação para não mais conter o código que executa estes trechos, mas sim, substituí-lo pelo código que transfere a execução para o *hardware* específico. No caso do *hardware* específico ser reprogramável também deve ser adicionado ao código da aplicação as instruções responsáveis pela programação do *hardware*.

De uma forma geral, a execução de programas tende a despender na maior parte do tempo de execução em uma pequena fração do código, fato esse conhecido como "regra do 90-10", onde 90% do tempo de execução é devido a execução de 10% do código. Em geral, aplicacões destinadas a sistemas embarcados tendem a seguir a regra do 90-10 com uma maior freqüência, que as aplicações destinadas a estações de trabalho.

Em [89], Dinesh et al. apresenta um detalhado estudo sobre o impacto e domínio de laços no tempo de execução de um programa. Dinesh, em seu trabalho, analisou três tipos de benchmarks: **SPECINT** [76] (mcf, gzip2, vpr, vortex, crafy, parser), **MediaBench** [73] (ADPCM, G721, MPEG, JPEG, Pegwit), **NETBench** [74] (dh, drr, tl, route e url), além de algumas aplicações utilizadas em criptografia (AES, 3DES, RC4, RC6, idea, blowfish, seal e sha1).

Os dados apresentados pelo autor (tabelas 6.1, 6.2 e 6.3) reportam a porcentagem do tempo de execução acumulativa dos primeiros 10 laços de cada programa, assim como o número de *core loops* do respectivo programa analisado. Define-se como *core loop* aquele laço que contribui com mais de 5% do tempo total de execução.

Pode-se verificar no trabalho que a contribuição dos primeiros 2 a 4 laços de aplicações embarcadas (MediaBench e NetBench) englobam, em média, 90% do tempo de execução. Desta forma, é possível obter-se um ganho significativo no desempenho de uma aplicação, através da implementação em hardware de poucos laços que são responsáveis pela maior parte do tempo de execução, sem um aumento considerável na área em silício. A área de silício dedicada ao hardware específico pode ainda ser reduzida utilizando-se um hardware reprogramável.

PROGRAMA	Laço										*Core loop*
Bzip2	18	37	46	55	61	67	73	77	82	85	6
Crafty	25	33	40	46	52	57	62	67	71	75	7
Eon	3	5	7	9	11	13	15	16	17	18	0
Gap	17	26	34	41	47	51	56	60	63	67	5
Gzip	29	43	54	65	73	81	86	90	93	96	6
MCF	24	47	58	68	76	83	89	94	94	95	7
Parser	8	15	21	27	33	38	43	47	50	54	6
Twolf	21	28	35	41	45	49	52	55	58	60	4
Vortex	14	23	29	35	37	40	42	43	45	46	4
Vpr	30	42	52	60	66	72	77	81	85	87	7
Média	19	30	38	45	50	55	59	63	66	68	5

Tabela 6.1: Porcentagem do tempo de execução dos 10 primeiros laços do benchmark SPECINT

PROGRAMA	Laço										*Core loop*
ADPCM	100	100	100	100	100	100	100	100	100	100	1
G721decode	47	69	87	92	95	96	97	98	98	97	3
G721encode	46	68	85	90	93	94	95	96	97	97	3
Jpegdecode	41	67	77	86	89	90	90	90	90	90	4
Jpegencode	29	42	54	65	75	84	89	92	93	95	6
Mpegdecode	76	81	84	87	89	90	92	93	93	94	1
Pegwit	40	71	81	85	89	92	94	97	98	98	3
Média	52	69	80	86	89	92	93	94	95	95	4

Tabela 6.2: Porcentagem do tempo de execução dos 10 primeiros laços do benchmark MediaBench

O potencial da abordagem adotada neste trabalho em melhorar o desempenho de um sistema foi testado, inicialmente, com a implementação no kit EXCALIBUR de dois sistemas que calculam a função dada pela equação 6.1. O primeiro sistema consistiu em um processador *NIOS* executando um programa escrito na linguagem C que implementa a função $f(x)$ e o segundo sistema consistindo no processador *NIOS* e um co-processador dedicado capaz de executar, em *hardware*, o termo x^2. A figura 6.1 mostra a evolução do tempo de execução do cálculo de x^2, para i variando de 1 a 10, para o cálculo realizado pelo processador *NIOS* (software) e pelo co-processador (*hardware*). Pode-se notar que o tempo gasto pelo co-processador para calcular x^2 ficou praticamente constante enquanto a implementação por software obteve um comportamento exponencial.

PROGRAMA	Laço										*Core loop*
AES	78	93	97	100	100	100	100	100	100	100	2
Blowfish	62	87	93	95	95	95	95	95	95	95	3
DES	46	90	94	98	98	98	98	98	98	98	2
IDEA	48	87	95	95	95	95	95	95	95	95	3
RC4	95	100	100	100	100	100	100	100	100	100	1
RC6	87	100	100	100	100	100	100	100	100	100	2
Seal	63	98	100	100	100	100	100	100	100	100	2
SHA1	75	98	100	100	100	100	100	100	100	100	2
CRC	87	99	100	100	100	100	100	100	100	100	2
TL	71	82	90	93	95	97	98	98	99	99	3
URL	77	95	99	100	100	100	100	100	100	100	2
DRR	26	48	65	80	85	87	91	93	95	95	4
Média	68	90	95	97	97	98	98	98	99	99	2

Tabela 6.3: Porcentagem do tempo de execução dos 10 primeiros laços do benchmark Security e NetBench

$$f(x) = \sum_{i=0}^{n} A_i * x^i \tag{6.1}$$

A figura 6.2 mostra a evolução dos tempos de execução do cálculo da função dada pela equação 6.1 quando executado no sistema só com o processador *NIOS* (*software*) e quando executado no sistema composto pelo processador e pelo co-processador (*hardware*). O ganho de desempenho poderia ser maior se uma maior granularidade fosse utilizada para a implementação em hardware, como por exemplo o cálculo de $A_i * x^i$ ou mesmo toda a função. Um limitante em se aumentar a granularidade do que será implementado diretamente em hardware é a área de silício necessária à sua implementação.

6.2 Implementação por Granularidade

O trecho de programa a ser escolhido para ser executado diretamente em hardware (no *CRD*) pode ser uma única linha de código, um bloco de linhas de código e mesmo todo um *loop*. Um fator importante a ser levado em consideração quando se realiza um particionamento *hardware/software* é em relação à granularidade do particionamento que tem influência direta no desempenho do sistema, bem como no custo e consumo. Em geral,

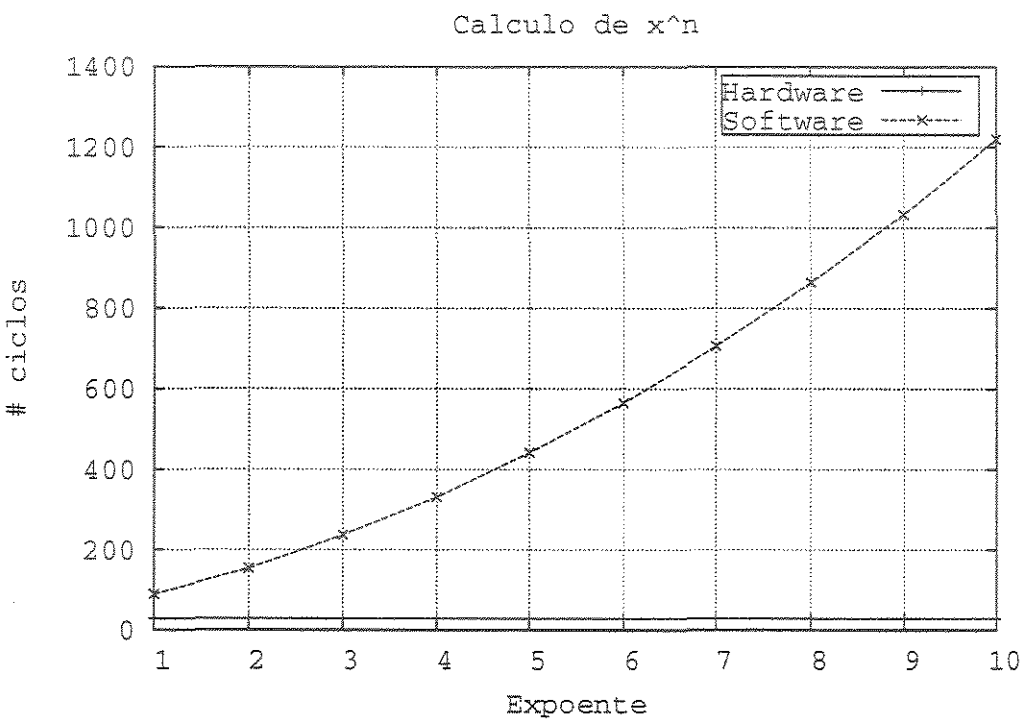


Figura 6.1: Tempo de execução da função potência (x^n) implementada em hardware e software

quanto maior a granularidade em sistemas deste tipo, maior será o ganho de desempenho [92]. No entanto, a utilização de uma maior granularidade acarreta em maior custo e consumo, pois é necessário uma maior área de silício dedicada a execução da tarefa. Independentemente do tamanho da granularidade, vale ressaltar que para que haja um ganho no desempenho, com a execução da nova instrução em *hardware*, é necessário que o trecho de código escolhido utilize instruções do processador de propósito geral pouco eficientes, em relação às outras instruções deste processador.

Devido aos atrasos para invocar a execução do trecho do programa no *CRD* os ganhos no desempenho são mais evidentes na substituição de laços, onde o atraso para programar e disparar a sua execução no *CRD* é diluído pelo número de iterações executadas pelo laço.

Tomemos como exemplo o trecho de programa mostrado na figura 6.3. Suponha que se deseje implementar no *CRD* o comando da linha 5 (a++). Fica evidente que por mais eficiente que seja a execução da nova instrução, criada em *hardware*, dificilmente

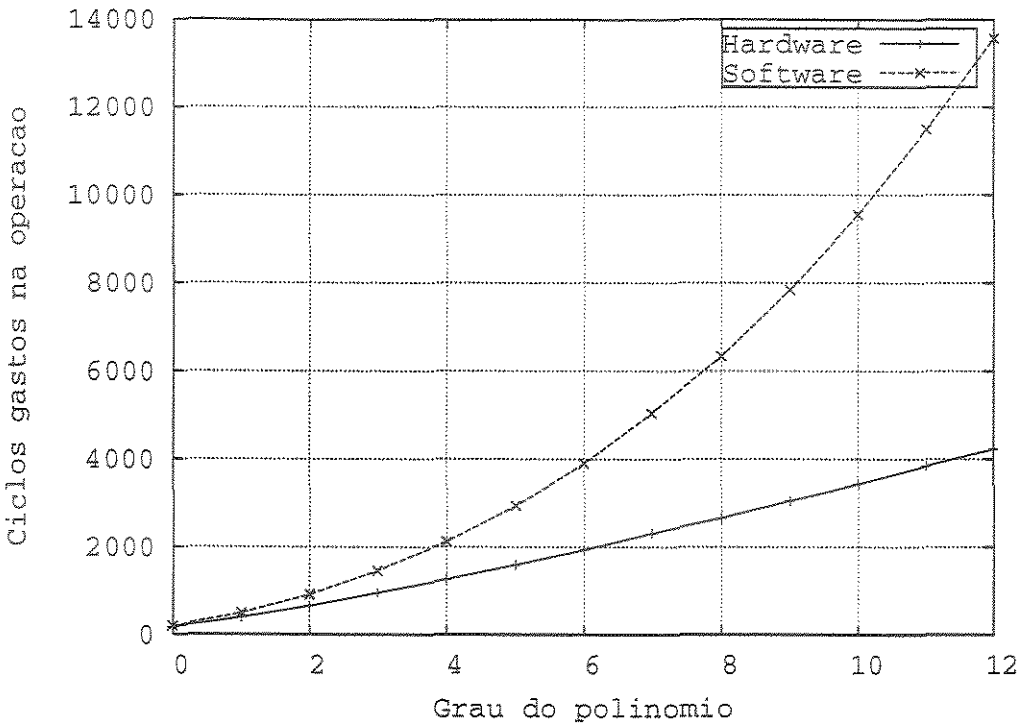


Figura 6.2: Implementação da função $f(x)$ em hardware x software

ela terá desempenho de execução final superior à implementada por software. No processador utilizado para a prototipagem (processador *NIOS*), esta instrução tem um CPI de 1 ciclo de clock quando executada em conjunto com outras instruções no pipeline do processador [18]. Somente para a configuração desta instrução no *CRD* será necessário bem mais do que isso. O mesmo não ocorre na linha de código número 9, onde o conjunto de instruções *NIOS* que irão ser executadas para implementar esta linha possuem um desempenho pior que um equivalente em *hardware*: 92 ciclos em software contra 44 (38 para programação; 4 para passar o controle ao *CRD* e 2 para a execução propriamente dita) ciclos em *hardware*, contabilizando os *overheads* de programação e chamada da instrução e 6 ciclos sem contabilizar os *overheads* de programação.

Um *datapath*, usando os recursos disponíveis no *CRD*[2], capaz de executar a linha de código de número 9 da figura 6.3 é apresentada na figura 6.4. Este *datapath* executa em seu primeiro ciclo de execução, a leitura em memória da variável **b** (a qual teve seu

[2]Apresentado no capítulo 4

```
1   for (i=0;i<N;i++)
2   {
3     if (a>b)
4     {
5        a++;
6     }
7     else
8     {
9        a =(b >> 2) * c;
10    }
11  }
```

Figura 6.3: Código exemplo em C

endereço armazenado em um registrador interno do *CRD* durante a etapa de inicialização), bem como seu deslocamento lógico de 2 posições para a direita, armazenando seu valor de saída em um dos registradores do *CRD*. No próximo ciclo, é realizada uma leitura em memória da variável c (endereço da variável c também foi previamente armazenado no banco de registradores na etapa de inicialização), que é multiplicado pelo resultado guardado no ciclo anterior, tornando assim o valor disponível para retorno ao processador de propósito geral. O ganho de desempenho, medido, para a execução de uma única vez desta instrução é de 2.09 vezes, se considerado o tempo de programação e de 15.3 vezes se só for considerado o tempo de chamada e execução.

A figura 6.5 mostra o novo código, utilizando a *CRDL* para o trecho de programa mostrado na figura 6.3 com a execução do código da linha 9 no *CRD*.

A nova instrução, denominada "INST1" requer dois ciclos para execução, definidos pela palavra reservada *clock* (linhas 7 e 10), o primeiro ciclo estabelece as entradas no deslocador, e sua saída ligada a um registrador. O segundo ciclo realiza a operação de multiplicação entre o registrador e a variável c, localizada em memória. O registrador "reg.0" é o registrador de saída do *CRD*, assim sendo, o valor calculado estará pronto para ser transferido para o *NIOS*.

Quanto maior o número de iterações maior será o *speed-up* alcançado pelo sistema, uma vez que os 38 ciclos utilizados para a programação do *datapath* da figura 6.4 no *CRD*, e a carga dos operandos a serem utilizados para os registradores internos são gastos uma

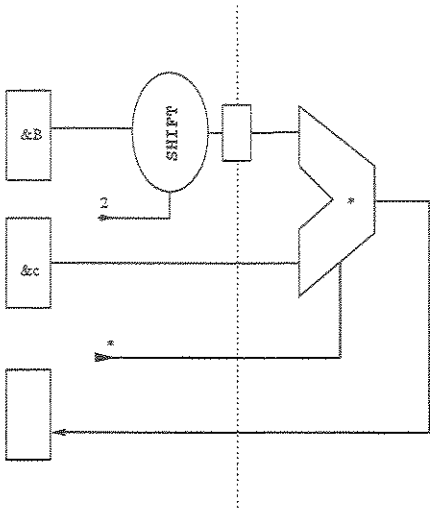


Figura 6.4: *Datapath* para execução da linha 9 do código apresentado anteriormente

única vez. Porém, ainda assim, estará limitado a um valor um pouco maior que 15 vezes, determinado pela equação 6.2.

$$\lim_{n\to\infty} \frac{92 * n}{38 + 6 * n} = \lim_{n\to\infty} \frac{92}{(38/n) + 6} = 15.33 \tag{6.2}$$

6.2.1 Implementação de Blocos de Instruções no *CRD*

Como visto anteriormente a substituição de uma única linha de código pode não propiciar um ganho elevado devido ao *overhead* de programação ser relativamente alto.

Este problema pode ser minimizado programando o *CRD* para executar em *hardware* um bloco de instruções, diluindo assim o *overhead* de programação. A memória de programação do *CRD* tem capacidade, na versão atual, para 16 instruções de até 8 ciclos cada. A memória de programação pode ser usada assim para definir *datapaths* para a execução de blocos ou instruções que aparecem em diversas partes do código do programa original, reduzindo desta forma o *overhead* (devido a programação), uma vez que a programação será realizada uma única vez na primeira ocorrência da instrução. O *overhead* a partir da segunda execução da instrução resume-se ao tempo para invocar a sua execução no *CRD*.

```
1   #CRDL
2   inst INST1
3   {
4     shift.in  <- b;
5     shift.shf <- 2;
6     reg.0 <- shift.out;
7     clock;
8     reg.0 <- reg.0 and c;
9     clock;
10  }
11  #CRDL
12
13  for (i=0;i<N;i++)
14  {
15    if (a>b)
16    {
17       a++
18    }
19    else
20    {
21       #CRDL
22          INST1;       /* chamada da instrução */
23       #CRDL
24    }
```

Figura 6.5: Transcrição de código para a *CRDL*

Considerando ainda o código mostrado na figura 6.3, pode-se por exemplo projetar um novo *datapath* para substituir o código compreendido entre as linhas 3 a 9 (Figura 6.6), esperando desta forma um aumento de desempenho superior ao caso de substituição de uma única linha de código.

A execução do trecho de programa mostrado na figura 6.3 no processador *NIOS* gasta 17 ciclos se **a** for maior que **b** e 102 ciclos se **a** for menor que **b**. Assim considerando-se uma distribuição homogênea na execução dos dois caminhos temos em média um tempo de execução igual a 59 ciclos. Para a execução do mesmo trecho de programa no *CRD*, usando o *datapath* apresentado na figura 6.6 tem-se 14 ciclos para a inicialização dos dados nos registradores do co-processador (Endereço e valor das variáveis), 79 ciclos destinados a configuração do novo *datapath* (escrita das palavras de controle na memória do *CRD*),

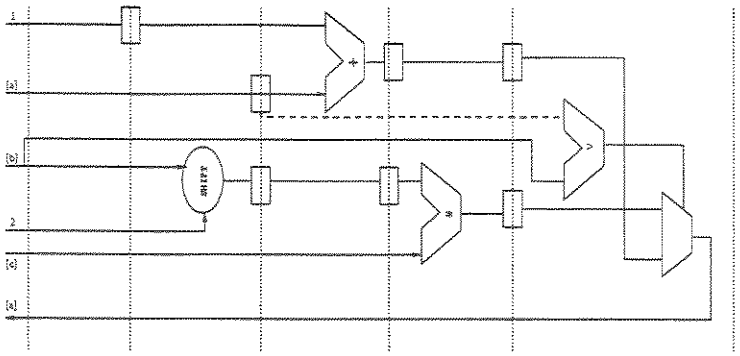


Figura 6.6: *Datapath* utilizada para substituição do bloco compreendido entre as linha 3 e 9, do código exemplo, mostrado na figura 6.3

3 ciclos para a chamada da nova instrução e 5 ciclos para a carga do resultado em um registrador do *NIOS*, totalizando 101 ciclos, produzindo um speedup de:

$$speedup = \frac{(17+102)/2}{(14+79+3)+5} = \mathbf{0.589} \tag{6.3}$$

Nota-se, no entanto, um decréscimo no desempenho quando comparado com a solução em software. Se considerarmos, porém, que este trecho de programa faz parte de um laço que será executado n vezes (Figura 6.8), um ganho de desempenho significativo pode ser observado, pois o *overhead* de programação assim como o de inicialização são diluídos entre as n chamadas, conforme pode ser observado na equação 6.4 e no gráfico da figura 6.7. O código mostrado na figura 6.8 apresenta a codificação do *datapath* usando-se *CRDL*.

$$speedup = \lim_{n\to\infty} \frac{n*(17+102)/2}{(14+79+3)+(5*n)} =$$

$$speedup = \lim_{n\to\infty} \frac{(17+102)/2}{96/n+5} = \mathbf{11.9} \tag{6.4}$$

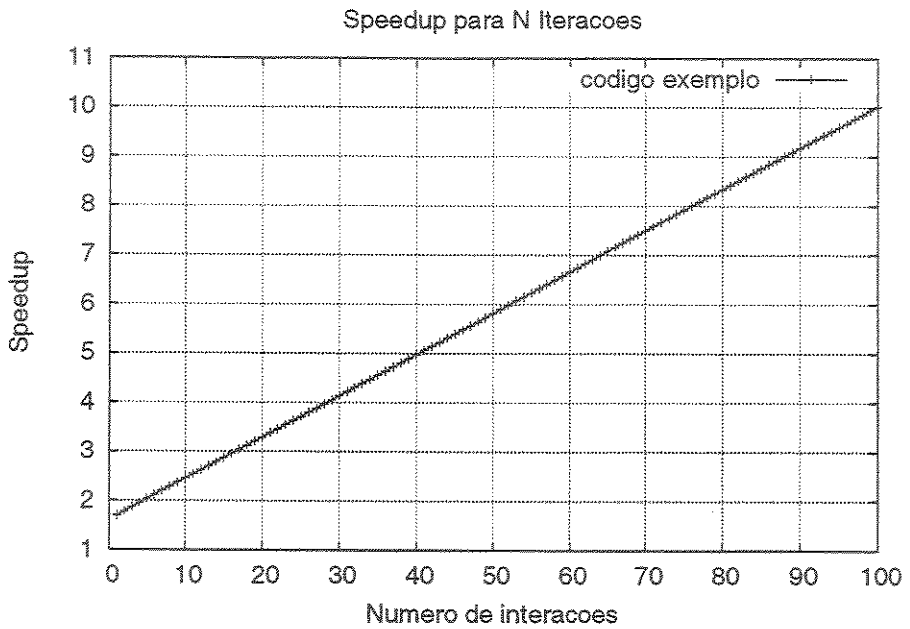


Figura 6.7: *Speed-up* obtido para n execuções consecutivas das instruções mostradas no trecho de programa da figura 6.3 utilizando-se o *CRD* programado com o *datapath* mostrada na figura 6.6

```
1   #CRDL
2   inst loop1
3   {
4     reg.1 <- regin.0;    /* carregado com 1 na etapa  */
5     clock                /* de inicializacao          */
6     shift.in <- b;
7     shift.shf <- 2;
8     reg.0 <- shift.out;
9     clock;
10    reg.1 <- a + reg.1;
11    shift.in  <- a;      /* utilizacao da unidade */
12    shift.shf <- 0;      /* como registrador      */
13    clock;
14    reg.0 <- reg.0 * c;
15    shift.in  <- shift.out;
16    shift.sh  <- 0;
17    clock;
18    if (shift.out > b)
19    {
20       reg.0 <- reg.1;
```

```
19      }
20
21      clock;
22    }
23    #CRDL
24
25    for (i=0;i<N;i++)
26    {
27       #CRDL
28       BLOCO1;
29       #CRDL
30    }
31
```

Figura 6.8: N chamadas a instrução *BLOCO1*

6.3 Implementação de Laços no CRD

Em geral, em um programa que contenha laços aninhados, o laço mais interno será o responsável direto pela maior parte do tempo de execução.

O *CRD*, usando seus contadores internos, é capaz de realizar em *hardware* operações de controle de laços com número de iterações conhecido antes da execução do mesmo (for), e manipulação de estruturas simples (Ex. vetores). Com os recursos disponíveis no *CRD* é possível o uso de uma variável de indução em *hardware*, tornando possível a manipulação de vetores ou espaços contíguos de memória.

Voltando ao exemplo utilizado até aqui, o objetivo agora é implementar a execução de todo o laço em *hardware*, esperando desta forma melhorar ainda mais o desempenho da instrução.

O ganho de desempenho neste caso torna-se superior ao anterior, pois o processador de propósito geral não necessita realizar as operações de indução (incremento do índice do vetor e comparação com o final da condição) na variável de controle, que agora são realizadas diretamente no *CRD*. A figura 6.9 mostra o ganho de desempenho quando todo o laço da figura 6.3 é implementado através de seu equivalente em *hardware*.

Na atual implementação do *CRD* não é possível a implementação de laços aninhados,

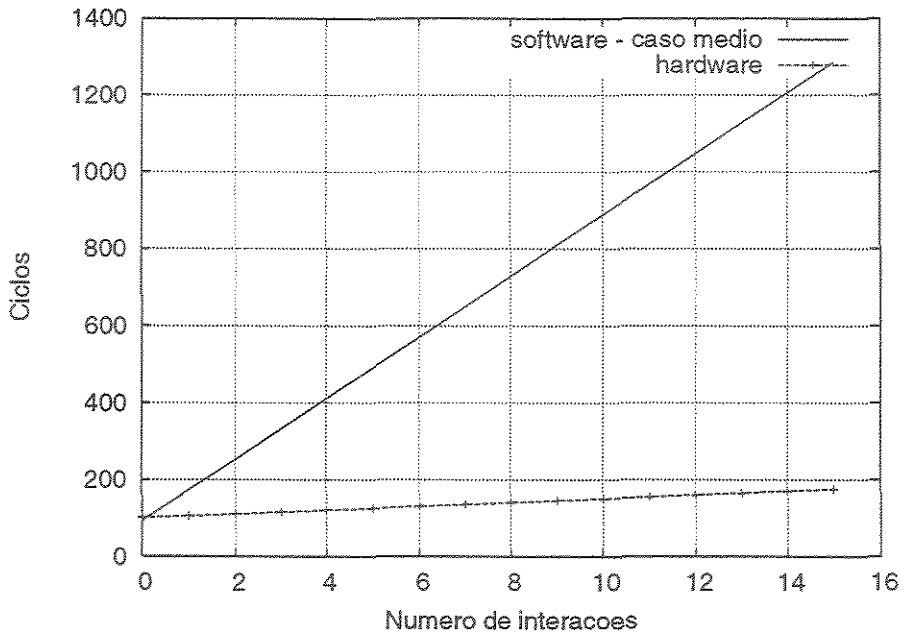


Figura 6.9: Curva comparativa de desempenho, entre a execução do código, apresentado na figura 6.3, em *hardware e software*

porém implementando-se, em *hardware*, somente o laço mais interno é possível obter-se uma considerável melhora no desempenho para diversos programas. A seguir é mostrado um exemplo, extraído do programa *MAT1x3* (Anexo B.12), de como pode ser implementada a execução de um laço interno usando-se o sistema *NIOS & CRD.*

No programa *MAT1X3* é realizada a multiplicação de uma matrix 3x3 por uma matriz 3x1, ou seja:

$$Y = H * X \tag{6.5}$$

Onde:

$$\begin{pmatrix} y1 \\ y2 \\ y3 \end{pmatrix} = \begin{pmatrix} h11 & h12 & h13 \\ h21 & h22 & h23 \\ h31 & h32 & h33 \end{pmatrix} * \begin{pmatrix} x1 \\ x2 \\ x3 \end{pmatrix} = \begin{pmatrix} h11 * x1 + h12 * x2 + h13 * x3 \\ h21 * x1 + h22 * x2 + h23 * x3 \\ h31 * x1 + h32 * x2 + h33 * x3 \end{pmatrix} \tag{6.6}$$

O trecho do código C que realiza essa multiplicação no programa *MAT1X3* é apresentado no Anexo B.

Como no *CRD* não é possível implementar uma instrução que implemente laços aninhados foi escolhido para ser executado no co-processador o laço mais interno que calcula

o valor de $y_i = (h_{i1} * x_1 + h_{i2} * x_2 + h_{i3} * x_3)$.

A figura 6.10 mostra o trecho do código executado no *NIOS* responsável pela execução da multiplicação entre as matrizes H e X, onde um laço executado no *NIOS* faz sucessivas chamadas à instrução *MAT* (linha 11), implementada no *CRD*. As linhas de código 6 a 9, passam para o *CRD* o endereço de memória (do sistema) onde está localizada a linha de H a ser usada na computação, o endereço da matrix X e o endereço onde deve ser armazenado o resultado y_i calculado.

```
1   ...
2   for (i = 0 ; i < 3; i++)
3     {
4       /* do matrix multiply */
5       p_crd = &CRD[0];
6       *p_crd++ = p_h;
7       *p_crd++ = p_x;
8       *p_crd   = p_y++;
9
10      nl = *MAT;
11      p_h += 3;
12    }
13  ...
```

Figura 6.10: Substituição do laço mais interno do programa "matrix1x3" por uma instrução em *hardware*

Ganhos na ordem de 8 a 12 vezes (para a programação dinâmica e estática respectivamente) foram obtidos de acordo com a implementação apresentada na figura 6.10, onde há um compartilhamento de computação entre o *NIOS* e o *CRD*.

Assim como o programa *MAT1X3* apresenta bom ganho de desempenho ao se substituir seu laço mais interno por uma chamada de instrução em *hardware*, outros programas pertencentes ao mesmo benchmark, como *LMS*, *MATRIX1*, *FIR2DIM* se beneficiam desta técnica, como poderá ser verificado no capítulo 7, que apresenta os resultados obtidos para a execução de partes de seus códigos no *CRD*.

6.4 Trabalhando com as Limitações do *CRD*

Conforme já mencionado, a implementação atual do *CRD* possui algumas limitações decorrentes de decisões de projeto que, em geral, foram estabelecidas no sentido de reduzir sua complexidade e área ocupada.

Entre as principais limitações tem-se o tamanho da memória de programação que permite um máximo de 16 instruções programadas simultaneamente no *CRD*. Apesar desta restrição não impedir que um número maior de instruções sejam utilizadas na execução de uma dada aplicação (uma instrução pode ser reprogramada a qualquer momento), ela tem impacto no desempenho devido ao *overhead* envolvido na programação de uma nova instrução no *CRD*. Outra limitação é que só existe um contador que pode ser usado como variável de indução de laços e sua contagem é sempre crescente e iniciada a partir de zero, não permitindo a implementação, no *CRD*, de laços aninhados e de laços com variável de indução decrescente, por exemplo.

Para um melhor aproveitamento da estrutura oferecida pelo *CRD*, pode ser necessário uma reestruturação dos laços do programa original. Algumas técnicas sugeridas neste trabalho, bem como as utilizadas em otimização de compiladores [96, 88], são apresentadas a seguir e podem ser utilizadas para tonar possível ou otimizar implementações de instruções no *CRD*, devido à suas limitações.

1. Evolução da variável de indução

   Para laços cujo valor inicial da variável de controle é diferente de zero é possível realizar modificações nos valores iniciais e finais da variável de controle de forma a tornar possível sua implementação no *CRD*.

   Por exemplo, a figura 6.11.(a) mostra um laço cuja variável de indução não inicia com o valor zero, não sendo possível a sua implementação direta no *CRD*. Examinando-se este laço, observa-se que ele atualiza as posições do vetor B, compreendidas entre os índices J e N. Este laço pode ser reescrito, conforme mostrado na figura 6.11.(b) onde as posições B[J] a B[N] foram redefinidas como sendo C[0] a C[N-J], possibilitando a sua implementação no *CRD*.

2. Fusão de Laços

```
1   for (k=J;k<N;k++)
2   {
3      B[k] = line+(k>>2);
4   }
5
6
7            (a)
```

```
int *C;
C = B+J;
for (k=0;k<(N-J);k++)
{
   C[k] = line+(k>>2);
}
        (b)
```

Figura 6.11: Código exemplo da manipulação de laços através da evolução da variável de indução, para proporcionar a implementação do loop no *CRD*

A fusão de laços[96] é uma técnica empregada quando existem dois laços, embora que distintos, executando uma tarefa similar. Procura-se eliminar a variável de indução de um deles de forma que ambos possam ser chamados por apenas uma instrução executada em *hardware*, ao contrário de duas instruções caso a técnica não fosse aplicada, diminuindo o espaço na memória interna do *CRD* e aumentando o desempenho final da nova instrução, pois apenas um *overhead* de programação é necessário.

3. Distribuição de Laços

A técnica de distribuição de laços [96, 88] propõe a separação de um laço em outros, de forma que se torne possível a implementação do mesmo em *hardware*. A figura 6.12.(a) apresenta um exemplo onde esta técnica é utilizada. Deve-se primeiramente aplicar a técnica de *evolução da variável de indução* para que a contagem do laço tenha seu valor inicial em zero. Porém, ao se realizar esta transformação, é encontrado um problema de dependência de dados na linha (6). Uma distribuição em dois laços (figura 6.12.(b))soluciona este problema.

4. Controle do loop fora dos contadores internos do CRD

Esta técnica permite que sejam efetuadas computações com decrementos e incrementos superiores a uma unidade no *CRD*.

5. Reutilização da Configuração do *Datapath*

Em muitos casos, diferentes instruções ou blocos de instruções representam grande

```
1   for (i=N-1;i>0;i--)
2   {
3       A[i] = B[i] + 1;
4       C[i] = A[i] / 2;
5       D[i] = 2 * C[i+1];
6   }
7
8
9
10              (a)
```

```
for (i=0;i<N-1;i++)
{
    A_1[i] = B_1[i] + 1;
    C_1[i] = A_1[i] / 2;
}
for (i=0;i<N-1;i++)
{
    D_1[i] = 2 * C_1[i+1];
}
              (b)
```

Figura 6.12: (a) Exemplo de código onde a variável de indução é decrementada, impraticável para substituição no *CRD*. (b) Aplicação da técnica de distribuição de laços para a implementação do laço candidato em *hardware*

similaridade entre suas codificações. A técnica aqui mencionada é reaproveitar *datapaths* criadas para execução no *CRD* com diferentes instâncias de entrada, ou como parte (bloco similar) de uma nova *datapath* que venha a ser idealizada. Alguns projetos de pesquisas, como o projeto Chameleon [14], desenvolvido pelo LSC - Unicamp, vêm desenvolvendo ferramentas automáticas para a determinação e agrupamento destes blocos similares, chamados de padrões.

6. Reutilização da Configuração do *Datapath* para diferentes funções

   A técnica apresentada anteriormente procura realizar a implementação em *hardware* de instruções que tenham padrões semelhantes, padrões refinados em cima de seqüências de operações, independente dos argumentos de entrada. Isto invalida tecnicamente, árvores de expressões semelhantes, com diferença entre operadores. Este método consiste em realizar a montagem de um *datapath* (ou instrução em *hardware*) a partir da análise entre semelhanças das árvores de expressões das computações candidatas a implementação.

6.5 Explorando Paralelismo no CRD

A arquitetura do *CRD* permite que algumas operações disponíveis no bloco básico (figura 4.7), sejam executadas em paralelo, desde que não haja dependência de dados entre estas

operações. Como exemplo, suponha que se deseja implementar no *CRD* uma instrução que calcule o valor de Z dado pela expressão mostrada na equação 6.7 abaixo:

$$Z = a_1 * b_1 + a_2 * b_2 \tag{6.7}$$

Uma solução direta (sem o uso de paralelismo) pode ser implementada para ser executada em 7 ciclos da seguinte forma:

i) Ler operando a_1;

ii) Ler operando b_1;

iii) realizar a operação $a_1 * b_1$ e armazenar em um registrador interno ao *CRD*;

iv) Ler operando a_2;

v) Ler operando b_2;

vi) realizar a operação $a_2 * b_2$ e armazenar em um registrador interno ao *CRD*;

vii) somar os resultado calculados nos passos (*iii*) com (*vi*) e retornar o valor.

Porém, se for explorado melhor os recursos disponíveis no *CRD* pode-se ter uma solução implementada usando-se um número menor de ciclos, conforme é apresentado:

i) Ler operando a_1;

ii) Ler operando b_1 e realizar a operação $a_1 * b_1$, armazenando o resultado em um registrador interno ao *CRD*;

iii) Ler operando a_2;

iv) Ler operando b_2 e realizar a operação $a_2 * b_2$, armazenando o resultado em um registrador interno ao *CRD*;

v) realizar a operação de soma entre os valores armazenados em *ii)* e *iv)* e retornar o valor.

O *CRD* pode ser programado para executar em um mesmo ciclo uma operação da ULA, um deslocamento lógico e uma leitura de operando da memória ou do banco de registradores.

Capítulo 7

Resultados

Neste capítulo são apresentados os resultados obtidos com o uso do *CRD* em conjunto com o processador de propósito geral *Nios* de 32 bits da *Altera*.

Para avaliar o desempenho do sistema *NIOS & CRD* foi utilizado o benchmark *DSPStone*, cujos programas originais foram executados no *NIOS* e uma versão modificada com o uso da *CDRL* foram executadas no sistema *NIOS & CRD*. Os resultados obtidos também foram comparados com valores da execução dos mesmos códigos (originais) em sistemas usando diversos DSPs existentes no mercado e disponíveis na literatura [25], pois atualmente, a utilização de DSPs em conjunto com processadores de propósito geral tem sido uma estratégia adotada em projeto de sistemas dedicados que requerem um maior desempenho no processamento de sinais, como aplicações multimídia [83].

Como o objetivo dos experimentos realizados e apresentados a seguir é mostrar o potencial do uso do sistema *NIOS & CRD* no ganho de desempenho, as implementações das instruções no *CRD* foram realizadas usando-se uma transcrição direta, quando possível, das operações descritas no código C original. Desta forma, evitou-se contabilizar aumento de desempenho devido a mudanças algoritmicas na implementação das instruções no *CRD*.

7.1 DSPStone

Os *benchmarks* mais utilizados para avaliar o desempenho de sistemas em processamento de sinais digitais, aplicações embarcadas e multimídia podem ser classificados em dois

tipos:

- Benchmark de aplicação: compostos por um conjunto de programas típicos da área de aplicação (por exemplo : JPG coder e decoder, MPEG coder e decoder, ADPCM coder e decoder, cálculo de CRC, EPIC etc.).

- Benchmark tipo *kernel* Composto por programas que implementam somente os núcleos (trechos mais executados do programa) de programas típicos da área de aplicação (por exemplo: FIR, Dot product, FFT etc.)

Neste trabalho optou-se em utilizar um benchmark tipo *kernel*, o *DSPStone*, com o qual procurou-se evidenciar os ganhos possíveis de serem obtidos em rotinas, críticas quanto ao tempo de execução, comuns a uma grande gama de aplicações.

O *DSPStone* é um benchmark composto pelos programas mostrados na tabela 7.1, cujos nomes refletem de forma direta a computação realizada pelo programa. Por exemplo, o programa *DOT_Product* realiza o produto escalar entre dois vetores.

Programa	Seção
Dot_Product	7.2
Complex_multiply	7.3
Real_Update	7.4
Biquad_one_Section	7.5
FIR	7.6
FIR2Dim	7.7
MAT1x3	7.8
Complex_Update	7.9
N_Complex_Updates	7.10
Convolution	7.11
LMS	7.12
Matrix1	7.13
N_Real_Updates	7.14

Tabela 7.1: Programas que compõem o benchmark DSPStone

Freqüência de operação

Apesar da síntese do processador *NIOS*[1] e do *CRD* tornar possível sua execução a uma freqüência superior a 50Mhz, o kit de desenvolvimento *Excalibur* possui um oscilador fixo de 33Mhz, assim a freqüência de 33MHz foi utilizada para os testes apresentados a seguir. As medidas de tempo de execução dos mesmos programas em diversos DSPs, obtidos na literatura, foram realizadas com o uso de diferentes freqüências. Assim, para efeito de comparação, foi utilizado o tempo de execução e o número de ciclos gastos, também disponíveis na literatura [26].

Chaves de compilação

Para as execuções dos programas no *NIOS*, foram realizadas duas tomadas de dados (número de ciclos para a execução do programa) uma para o programa compilado pelo *gcc* [32] sem chaves extras de otimização e outra para compilação com o uso da chave de otimização **o2** (Ver apêndice A para diretivas de compilação).

7.2 Dot_Product

O kernel DOT_PRODUCT (Apêndice B.1) executa o produto vetorial entre dois vetores de inteiros de 2 elementos cada. A tabela 7.2.(a) apresenta o número de ciclos gastos para executar este programa em diversos processadores DSP e no processador de propósito geral *NIOS*. Para o processador *NIOS* são apresentados dois valores, um para o programa compilado sem otimização e outro compilando com otimização.

A tabela 7.2.(b) apresenta o número de ciclos gastos quando o *kernel* DOT_PRODUCT é executado no sistema *NIOS* & *CRD* com o código das linhas 69 e 70 (Apêndice B.1) implementado no *CRD*. Na tabela 7.2.(b) os dados referentes à linha rotulada "Dinâmico" referem-se àqueles coletados quando é utilizado a programação dinâmica, isto é, a instrução é programada no *CRD* toda vez que ela é executada. Os dados referentes à linha rotulada "Estático" referem-se aos dados para a programação estática, isto é, a instrução é programada uma única vez no *CRD* e a cada chamada só é executado o código para

[1] A segunda versão do soft-core do processador - *NIOS II*

a passagem de parâmetros e ínicio da execução. Nesta tabela também são apresentados os números de ciclos necessários à: programação da instrução no *CRD*; inicialização (passagem dos parâmetros) e execução da instrução.

Processador	#ciclos	tempo(ns)
Ad21	13	787.8
Tms32c50	19	475
Dsp56001	22	666.6
Dsp56156	58	966.6
NEC77016	16	484.8
Nios	203	6151.5
Nios(O2)	192	5818.1

(a)

	Overhead programação	Overhead inicialização	Ciclos de execução	Total	tempo(ns)
Dinâmico	47	7	11	65	1969.7
Estático	-	7	11	18	545.4

(b)

Tabela 7.2: Número de ciclos gastos para executar o *kernel* DOT_PRODUCT. (a) em diversos processadores DSP e no processador NIOS, com e sem otimização O2; (b) no sistema *NIOS & CRD* com programação dinâmica e programação estática.

Os ganhos de desempenho com o uso do sistema *NIOS & CRD* em relação ao processador *NIOS* estão apresentados na tabela 7.3, para a programação do *CRD* de forma dinâmica e estática. Para a forma estática obteve-se um *speed-up* de 1060% quando comparado ao tempo de execução do código otimizado e de 1127% quando comparado à execução do código não otimizado.

Programação do *CRD*	Código otimizado	Código não otimizado
Dinâmico	2.95	3.12
Estático	10.6	11.27

Tabela 7.3: *Speed-up* do sistema *NIOS & CRD* em relação ao processador *NIOS* para o *kernel* DOT_PRODUCT

A figura 7.1 mostra os tempos de execução para os dados obtidos na tabela 7.2.(a). Pode-se notar que o tempo de execução do sistema *NIOS & CRD* é aproximadamente igual ao tempo de execução dos DSPs que executam em uma menor quantidade de ciclos de relógio. O tempo do *NIOS & CRD* poderia ser reduzido caso fosse utilizado uma frequencia maior.

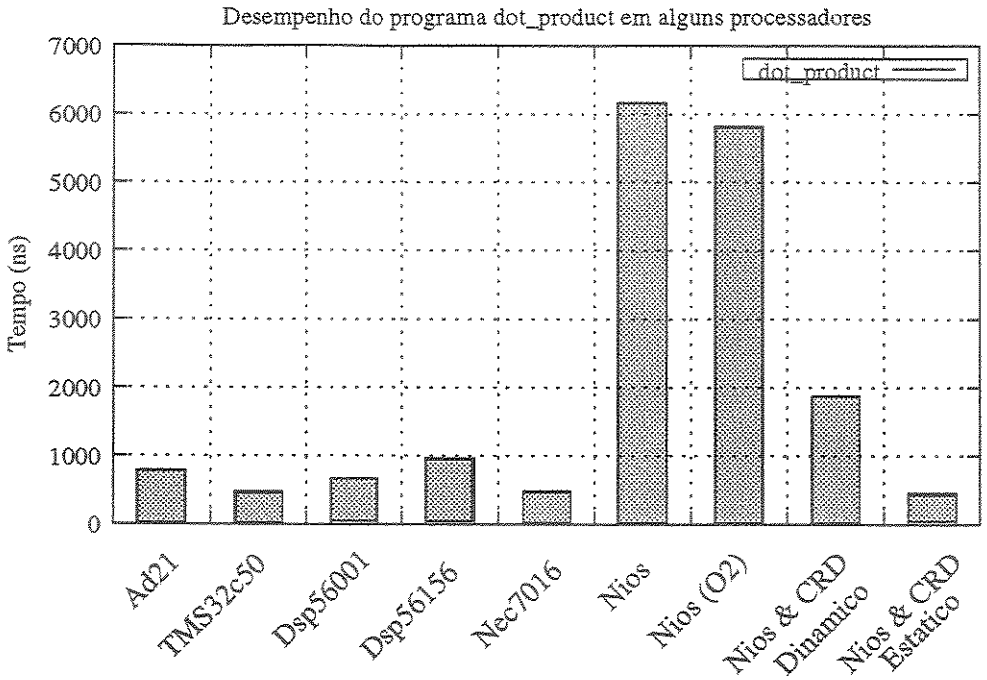


Figura 7.1: Tempo de execução em nanosegundos do *kernel* DOT_PRODUCT

7.3 Complex_multiply

O *kernel* COMPLEX_MULTIPLY executa a multiplicação entre dois números complexos. Na tabela 7.4.(a) são apresentados, assim como no caso anterior, o número de ciclos gastos para executar este *kernel* em diversos processadores DSP e no processador de propósito geral *NIOS*. São apresentados na tabela 7.4.(b) o número de ciclos devidos à configuração, chamada e execução da instrução no co-processador reconfigurável.

O *speed-up* obtido através da substituição do trecho de código compreendido entre as linhas 49 e 50 do anexo B.3, por uma instrução equivalente em *hardware* em relação ao processador *NIOS* é apresentada na tabela 7.5 e a figura 7.2 mostra os tempos de execução em diversos processadores.

Processador	ciclos	tempo(ns)
Ad21	16	969.7
Tms32c50	17	425
Dsp56001	36	1090
Dsp56156	74	1233
NEC77016	18	545.4
Nios	352	10666.6
Nios (O2)	328	9939.4

(a)

	Overhead programação	Overhead inicialização	Ciclos de execução	Total	tempo(ns)
Dinâmico	168	15	22	205	6212.1
Estático	-	15	22	37	1121.2

(b)

Tabela 7.4: Número de ciclos gastos para a execução do *kernel* COMPLEX_MULTIPLY: (a) em diversos processadores e no processador NIOS com e sem chave de otimização; (b) no sistema *NIOS & CRD* com programação dinâmica e estática

Programação do *CRD*	Código otimizado	Código não otimizado
Dinâmico	1.6	1.71
Estático	8.86	9.51

Tabela 7.5: *Speed-up* do sistema *NIOS & CRD* em relação ao processador *NIOS* para o *kernel* COMPLEX_MULTIPLY

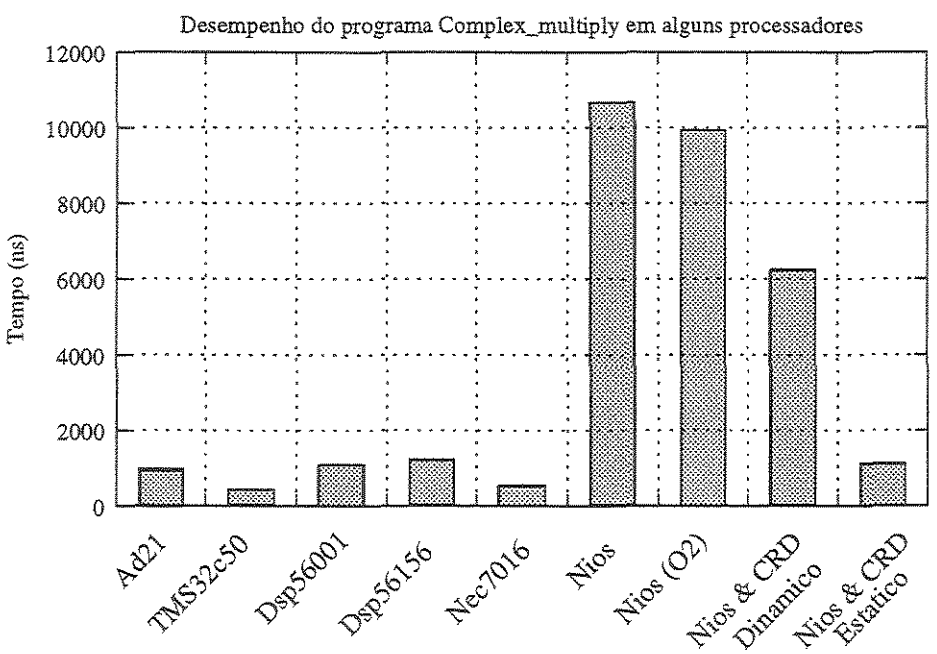


Figura 7.2: Tempo de execução em nanosegundos para o *kernel* COMPLEX_MULTIPLY

7.4 Real_Update

O *kernel* REAL_UPDATE realiza uma operação tipo MAC entre números inteiros. Uma operação tipo MAC é a execução de uma multiplicação seguida de uma operação de soma entre o resultado obtido e um terceiro número (em geral, o resultado da operação MAC anterior). O *profile* da linha que realiza esta operação (linha 58 do anexo B.4), em alguns processadores, é apresentado na tabela 7.6.(a), assim como os *overheads* provocados pela configuração e chamada da instrução pelo processador *NIOS* no *CRD* (Tabela 7.6.(b)).

Processador	ciclos	tempo(ns)
Ad21	6	363.6
Tms32c50	7	175
Dsp56001	16	484.4
Dsp56156	28	466.6
NEC77016	13	393.9
Nios	88	2666.6
Nios(O2)	89	2696.9

(a)

	Overhead programação	Overhead inicialização	Ciclos de execução	Total	tempo(ns)
Dinâmico	47	11	9	67	2030.3
Estático	-	11	9	20	606.6

(b)

Tabela 7.6: Número de ciclos gastos para executar o *kernel* real_update: (a) em diversos processadores; (b) no sistema *NIOS & CRD* com programação dinâmica e programação estática.

A figura 7.3 mostra os tempos de execução deste *kernel* em diversos processadores. Percebe-se que mesmo utilizando a programação estática, o conjunto *NIOS & CRD* não consegue equiparar-se aos DSPs, que em geral, possuem instruções para realizar operações tipo MAC em um ciclo de relógio, ao contrário do *CRD* que por decisão de projeto necessita de no mínimo 2 ciclos para realizar a operação, pelo fato do multiplicador estar inserido dentro da ULA (Ver capítulo 3).

No entanto, se compararmos o ganho de desempenho apresentado entre o processador de propósito geral *NIOS* e o sistema *NIOS & CRD*, obtém-se ganhos na ordem de 4 vezes com a implementação desta instrução em *hardware*, conforme apresentado na tabela 7.7.

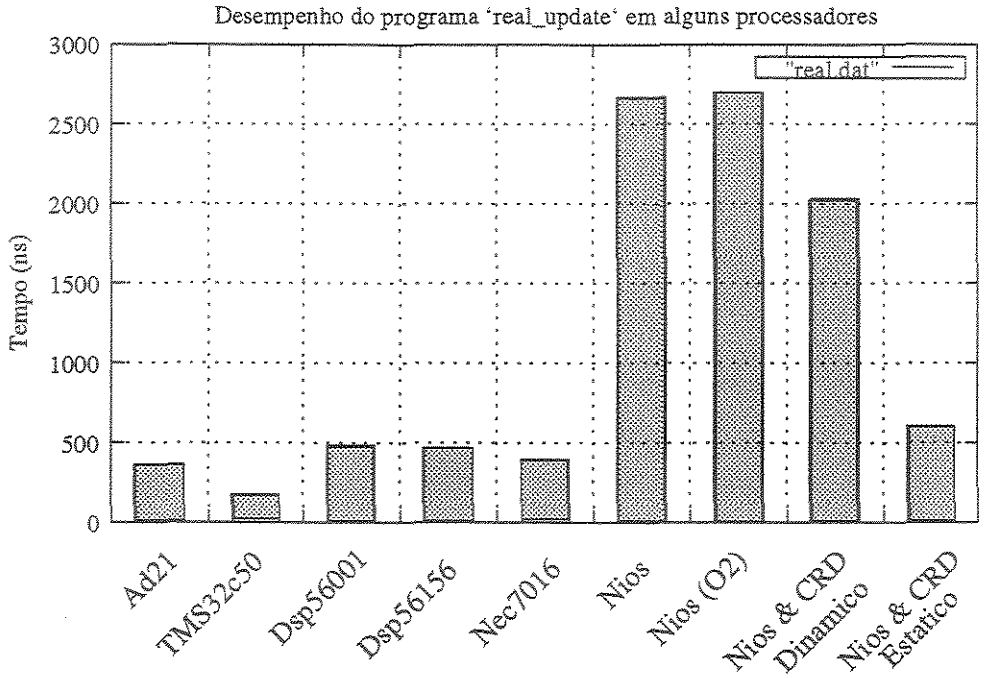


Figura 7.3: Tempo de execução do *kernel* REAL_UPDATE em nanosegundos

Programação do *CRD*	Código otimizado	Código não otimizado
Dinâmico	1.31	1.32
Estático	4.45	4.45

Tabela 7.7: *Speed-up* do sistema *NIOS & CRD* em relação ao processador *NIOS* para o *kernel* REAL_UPDATE

7.5 Biquad_one_section

Para a implementação do *kernel* BIQUAD_ONE_SECTION foram criadas 2 novas instruções para realizar a computação de w_n e y_n, apresentadas na equação 7.1 e 7.2.

$$w_n = x_n - a1 * w_{n-1} - a_2 * w_{n-2} \tag{7.1}$$

$$y_n = b_0 * w_n + b_1 * w_{n-1} + b_2 * wn - 2 \tag{7.2}$$

A tabela 7.8.(a) apresenta o número de ciclos gastos para a execução das linhas 67 a 78 do Anexo B.4 em diversos processadores, já a tabela 7.8.(b) apresenta o número de

ciclos necessários para se programar, inicializar e computar as novas instruções criadas no *CRD*. Pode-se notar uma distinção de ciclos (20/27) para o *overhead* de inicialização, isto se deve ao fato que a "INST1" requer um menor número de variáveis de incialização, em comparação com "INST2".

Processador	ciclos	tempo(ns)
Ad21	29	1757.5
Tms32c50	28	700
Dsp56001	72	2181.8
Dsp56156	60	1000
NEC77016	34	1030.3
Nios	483	14666.6
Nios(O2)	439	13303

(a)

	Overhead programação	Overhead inicialização	Ciclos de execução	Total	tempo(ns)
Dinâmico	47	20/27	8	276	8363.6
Estático	-	20/27	8	182	5515.1

(b)

Tabela 7.8: Ciclos necessários para a execução da instrução *BIQUAD_ONE_SECTION* em diversos processadores. (b) número de ciclos gastos na execucão da instrucão candidata em *hardware* utilizando programação estática e dinâmica

A figura 7.4 apresenta o tempo de execução do *kernel*, para diversos processadores. Pode-se notar que apesar do sistema *NIOS & CRD* conseguir um melhor desempenho quando comparado a abordagem de software, ainda é inferior ao obtido com o uso de DSPs. O principal motivo deste moderado desempenho foi o critério adotado durante a criação das novas instruções. Procurando utilizar uma metodologia de implementação que trouxesse uma economia da memória interna do *CRD* o ganho de desempenho, em contrapartida, foi afetado.

Programação do *CRD*	Código otimizado	Código não otimizado
Dinâmico	1.59	1.75
Estático	2.41	2.65

Tabela 7.9: *Speed-up* do sistema *NIOS & CRD* em relação ao processador *NIOS* para o *kernel* BIQUAD_ONE_SECTION

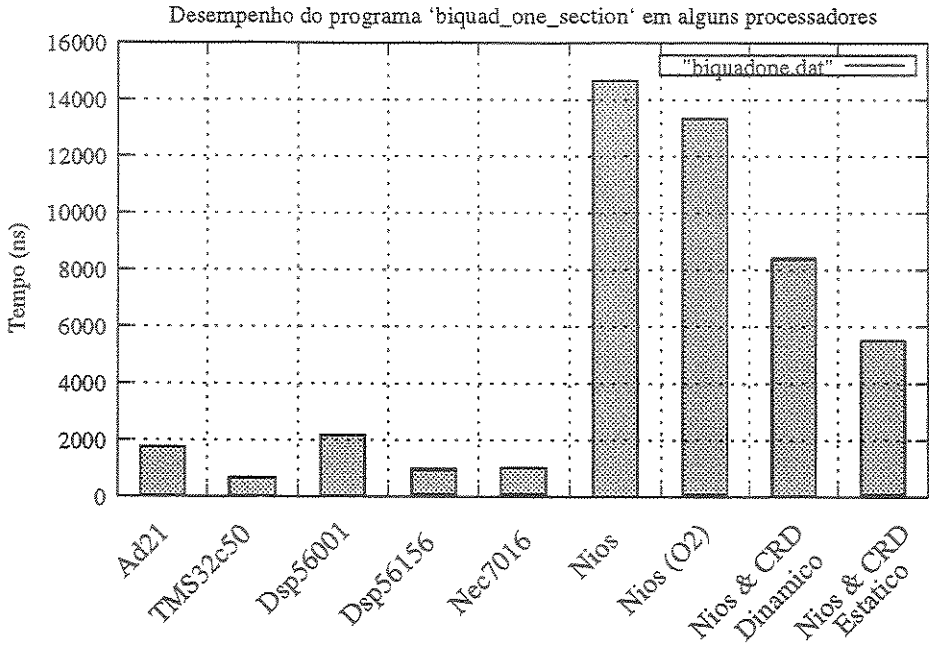


Figura 7.4: Tempo de execução do *kernel* BIQUAD_ONE_SECTION em nanosegundos

7.6 Fir

O *kernel* FIR é um filtro multiplicativo (Equação 7.3) que utiliza operações de multiplicação entre elementos de dois vetores e a cada operação o valor do elemento do primeiro vetor deve ser atualizado pelo anterior.

$$\sum_{i=0}^{N-1} x_i * y_{N-i} \tag{7.3}$$

Foi realizado, para a codificação do *kernel* FIR (Anexo B.5) em *CRDL* uma modificação em sua variável de indução (conforme mencionado no capítulo anterior) para melhor atender às requisições de *hardware* do *CRD*.

A tabela 7.10.(a) apresenta os dados obtidos para a execucão do *kernel Fir* em alguns processadores DSP e o processador *NIOS*. A tabela 7.10.(b) apresenta os ciclos necessários para realizar a gravação, inicialização e chamada da instrução que executa as linhas 66 a 75 do anexo B.9 na abordagem por *hardware*.

Obteve-se ganhos superiores a 14 vezes, para um laço com um número fixo de 16 iterações. Quando o corpo e as instruções subjacentes (Linha 74 e 75 do Anexo B.9) foram

Processador	ciclos	tempo(ns)
Ad21	175	10606
Tms32c50	124	3100
Dsp56001	324	9818.1
Dsp56156	232	3866.6
NEC77016	115	3484.8
Nios	1924	58303
Nios(O2)	1599	48454

(a)

	Overhead programação	Overhead inicialização	Ciclos de execução	Total	tempo(ns)
Dinâmico	95	21	108	224	6787.87
Estático	-	21	108	129	3909.1

(b)

Tabela 7.10: Ciclos necessários para a execução da instrução *fir*: (a) em diversos processadores,; (b) no *CRD*, evidenciando os atrasos decorrentes à programação, inicialização e chamada da instrução

implementadas em *hardware*, conforme pode ser visto na tabela 7.11. Estudos sobre o comportamento do *speed-up*, para variações no número de iterações do laço, mostram ganhos de desempenho ainda maiores, chegando a mais de 4 vezes em relação ao DSP *56156* e de quase 20 vezes quando comparado à execução no processador *NIOS*, para a execução do laço com 40 iterações, conforme é apresentado na figura 7.6

Programação do *CRD*	Código otimizado	Código não otimizado
Dinâmico	7.13	8.58
Estático	12.39	14.91

Tabela 7.11: *Speed-up* do sistema *NIOS & CRD* em relação ao processador *NIOS* para o *kernel* "FIR"

A figura 7.5 apresenta os tempos de execução do *kernel* FIR nos diversos processadores listados na tabela 7.10.(a) e no conjunto *NIOS & CRD*. Para este caso, vale notar que mesmo quando usada a abordagem de programação dinâmica, o tempo de execução do *kernel* FIR é menor do que o de alguns DSPs (AD21 e DSP56001). Para a programação estática o tempo de execução no *NIOS & CRD* é comparável aos dos melhores DSPs (DSP56156, TMS32C50 e NEC7016). Cabe aqui lembrar que o tempo de execução do conjunto *NIOS & CRD* poderia ser em torno de 50% menor, já que por limitações da placa de prototipagem ele está operando a 33Mhz e poderia estar operando a 50Mhz.

Na tabela 7.12 é apresentada as equações obtidas, através de processos de interpolação de dados (aproximação polinomial), para os processadores *NIOS* e *CRD*, com o intuito

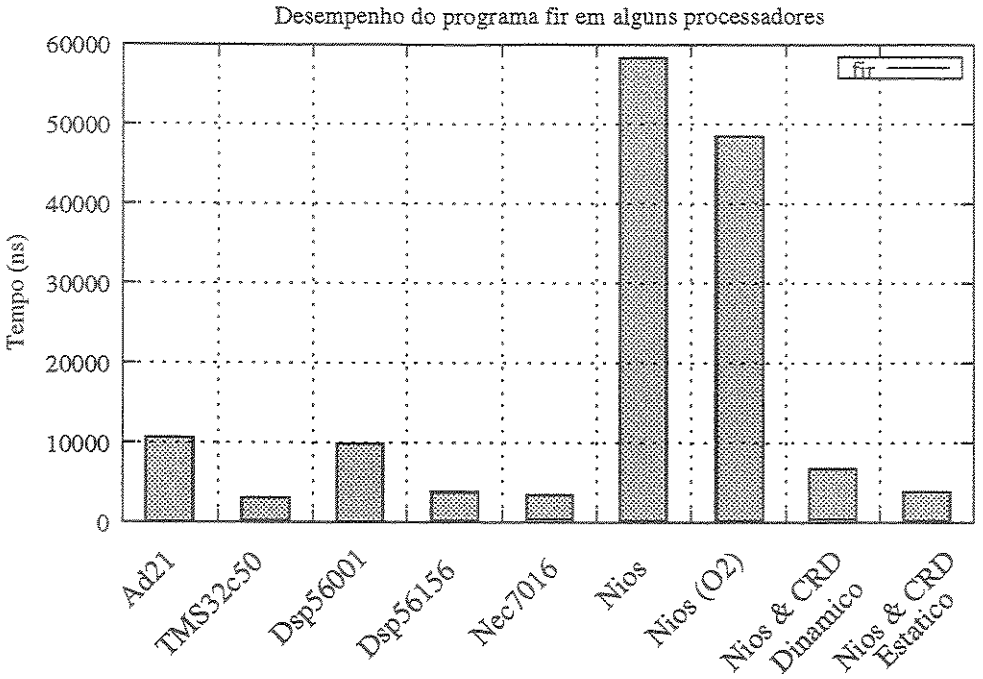


Figura 7.5: Tempo de execução em nanosegundos do *kernel* FIR

de obter comparações com os processadores DSP utilizados no trabalho, levando em consideração a variação do número de iterações do laço. Pode-se notar que quanto maior o número de laços (n) o conjunto *NIOS & CRD* apresenta um número menor de ciclos por instrução para a computação.

Processador	# ciclos
Ad21	$11 * n - 1$
Tms32c50	$7 * n + 12$
Dsp56001	$20 * n + 4$
Dsp56156	$14 * n + 8$
Nios	$121 * n - 12$
Nios & CRD	$6 * n + 12$

Tabela 7.12: Equação do número de ciclos decorridos de uma execução de n iterações para diversos processadores na execução do *kernel* FIR

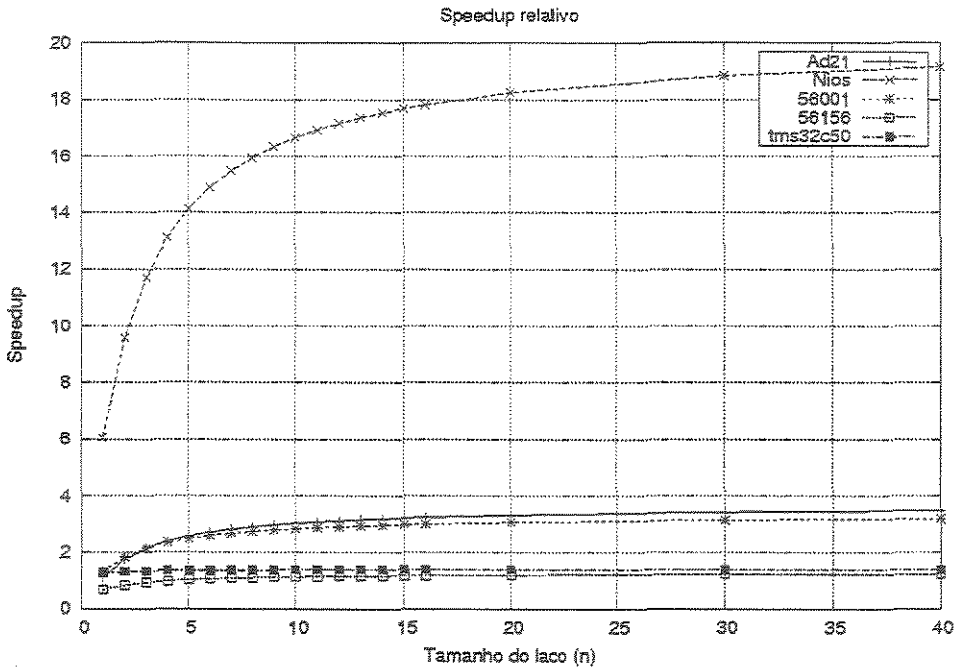


Figura 7.6: *Speed-up* máximo relativo em função do número de iterações do laço

7.7 Fir2dim

O *kernel* FIR2DIM é um *FIR* Bidimensional. Encontra-se aqui, diferentemente do que se tinha até então, a chamada de laços internos, linhas 117 a 140 do Anexo B.10. O número de ciclos necessários para realizar esta computação em diversos processadores pode ser vista na tabela 7.13.(a). Neste caso, chama a atenção o ganho de desempenho obtido (para o caso do processador *NIOS*) com o uso da chave de otimização *o2*.

Para um aumento de desempenho significativo neste *kernel*, procurou-se realizar a substituição dos laços internos, compreendido pelas linhas 129-130, 132-133 e 135-136, pelos seus equivalentes em *hardware*.

Existe a chamada de três laços internos, que podem ser implementados através de uma mesma instrução no *CRD*, alterando somente os valores de inicialização. A tabela 7.13.(b) apresenta os dados quando o *overhead* de programação é único, uma vez que os três laços são executados pela mesma instrução *CRD*. É apresentado, entre parênteses, os ciclos necessários para a inicialização de cada instrução. A primeira inicilização possui um maior valor (155) devido as linhas 122 a 127 do Anexo B.10, que são parte integrante

da inicialização.

O tempo de execução apresentado é aquele necessário à toda a computação, isto é, são computados no tempo total de 1510 ciclos (para programação dinâmica) e 1447 ciclos (programação estática), o tempo de execução das três chamadas das instruções executadas no *CRD* e os ciclos necessários para a computação dos laços externoss (linha 117 e 120 do Anexo B.10).

Processador	ciclos	tempo(μs)
Ad21	1898	115.03
Tms32c50	1938	48.45
Dsp56001	3588	108.77
Dsp56156	9060	151
NEC77016	1546	46.84
Nios	16831	510.03
Nios(O2)	14823	449.18

(a)

	Overhead programação	Overhead inicialização	Ciclos de execução	Total	tempo (μs)
Dinâmico	63	216 (155+32+29)	1231	1510	45.75
Estático	-	216(155+32+29) (155+32+29)	1231	1447	43.84

(b)

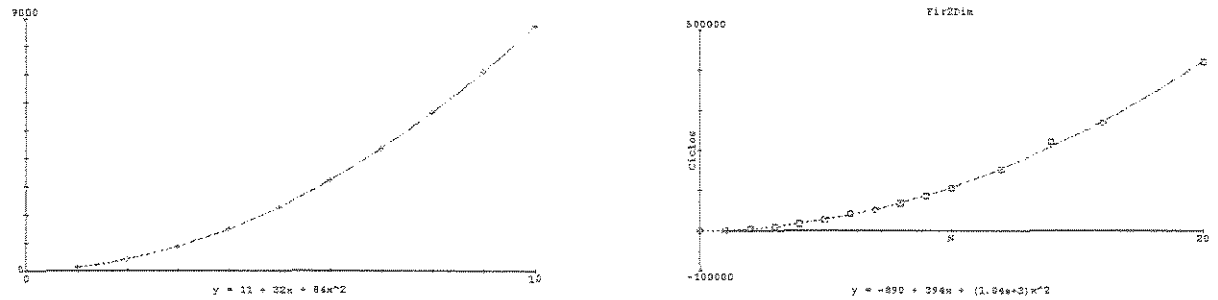
Tabela 7.13: Ciclos necessários para: (a) execução da instrução FIR2DIM em diversos processadores; (b) programação, inicialização e execução da computação no sistema *NIOS & CRD*

O *Speed-up* obtido, através da substituição dos 3 laços mais internos do *kernel* FIR2DIM, por seu equivalente em *hardware*, pode ser visto na tabela 7.14.

Programação do *CRD*	Código otimizado	Código não otimizado
Dinâmico	9.81	11.14
Estático	10.24	11.63

Tabela 7.14: *Speed-up* do sistema *NIOS & CRD* em relação ao processador *NIOS* para o *kernel* "Fir2dim"

Como os tempos de execução envolvidos neste caso dependem do número de iterações realizadas pelos laços externos e pelos laços internos, foi determinado por meio de interpolação, as equações que definem o número de ciclos utilizados para a execução destes laços no *CRD* e no *NIOS*. São apresentados nas figuras 7.7.(a) e 7.7.(b) respectivamente. A tabela 7.15 mostra estas mesmas equações e suas equivalentes para a execução do *kernel* FIR2DIM em diversos processadores DSP.



(a) (b)

Figura 7.7: Curva resultante, obtidas para determinação equação polinomial que descreve o comportamento da computação em hardware (a) e em software (b)

A figura 7.8 mostra os tempos de execução do *kernel* FIR2DIM, onde observa-se que o desempenho do *CRD* e equivalente aos melhores DSPs.

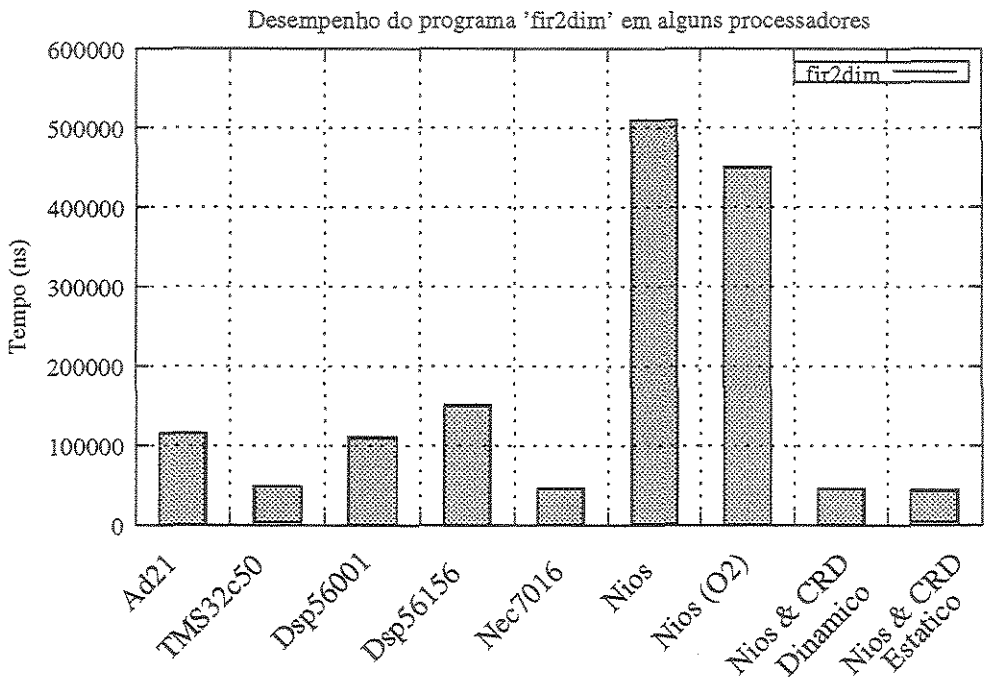


Figura 7.8: Tempo de execução em nanosegundos do *kernel* FIR2DIM

Usando o número de iterações do laço mais externo do *kernel* FIR2DIM de 1 a 20 iterações, pode-se esboçar o gráfico comportamento do *speed-up* obtido com a utilização do *CRD* em relação aos outros processadores, conforme apresentado na figura 7.9

Processador	Equação em função de n
Ad21	$116n^2 + 9n + 6$
Tms32c50	$115n^2 + 22n + 10$
Dsp56001	$216n^2 + 30n + 12$
Dsp56156	$562n^2 + 38n + 16$
Nios	$1040n^2 + 394n + 890$
Nios&CRD	$84n^2 + 32n + 11$

Tabela 7.15: Equação do número de ciclos decorridos de uma execução de N iterações para diversos processadores na execução do *kernel* FIR2DIM

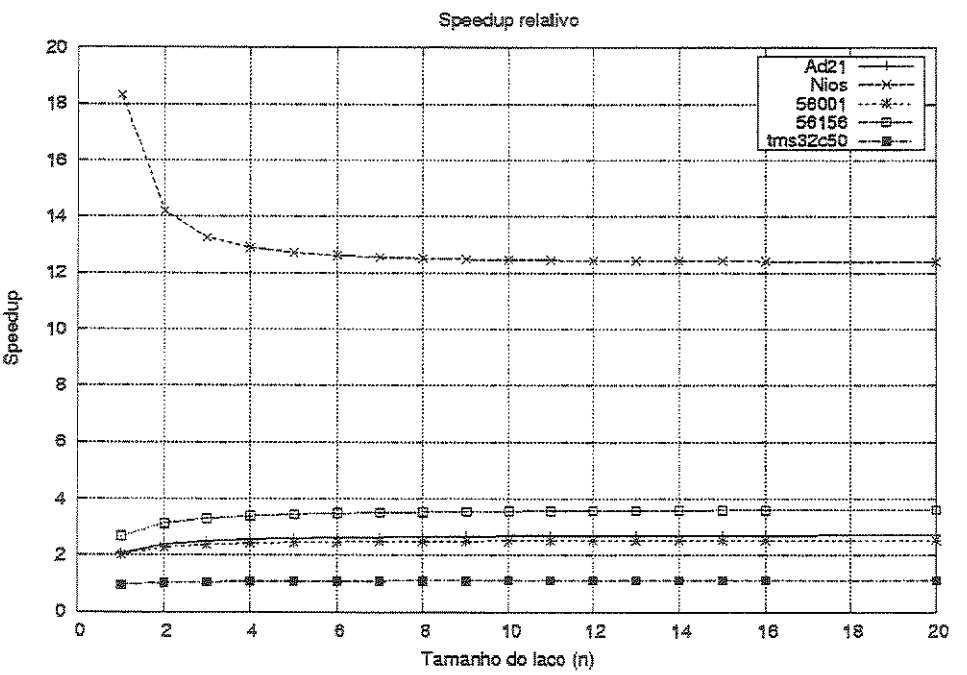


Figura 7.9: Speedup máximo relativo em função do número de iterações do laço

7.8 MAT1x3

O *kernel* MAT1x3 já fora discutido no capítulo anterior. Nesta seção serão apresentados os resultados obtidos conforme a implementação sugerida anteriormente. A tabela 7.16.(a) apresenta o número de ciclos necessários para a computação do *kernel* em alguns processadores DSP e no processador *NIOS*. Na tabela 7.16.(b) é mostrado os números de ciclos devido ao *overhead* de programação e inicialização, assim como o tempo de execução da nova instrução, que substitui o bloco de código compreendido entre as linhas 63 e 64 do Anexo B.12. É computado no número de ciclos de execução da instrução o incremento de 3 unidades (incremento de linha)[2], realizada exclusivamente pelo processador *NIOS*.

Processador	ciclos	tempo(ns)
Ad21	63	3818.2
Tms32c50	109	2725
Dsp56001	186	5636.3
Dsp56156	274	4566.6
NEC77016	84	2545.4
Nios	1480	44848.5
Nios(O2)	864	26181.8

(a)

	Overhead programação	Overhead inicialização	Ciclos de execução	Total	(ns)
Dinâmico	63	17	101	181	5484.8
Estático	-	17	101	118	3575.7

(b)

Tabela 7.16: (a) Número de ciclos necessários para a execução do *kernel mat1x3* em diversos DSPs. (b) Tempo dispendido na execucão da instrucão candidata em *hardware*, no sistema *NIOS & CRD* com programação dinamica e estática

A tabela 7.17 mostra o ganho de desempenho que se obteve com a substituição das linhas 63 e 64 por seu equivalente em *hardware*. Apesar da computação de incremento de linha ser realizada em software, pelo processador *NIOS*, experimentou-se ganhos de até 12 vezes.

Programação do *CRD*	Código otimizado	Código não otimizado
Dinâmico	4.77	8.17
Estático	7.32	12.54

Tabela 7.17: *Speed-up* do sistema *NIOS & CRD* em relação ao processador *NIOS* para o *kernel* MAT1X3

[2]Ver modificação de código proposta no capítulo anterior

Na figura 7.10 é evindenciada a diferença entre os tempos da alternativa de programação em software por um processador de propósito geral e da alternativa *hardware*/software que equiparou-se ao processamento em alguns DSPs.

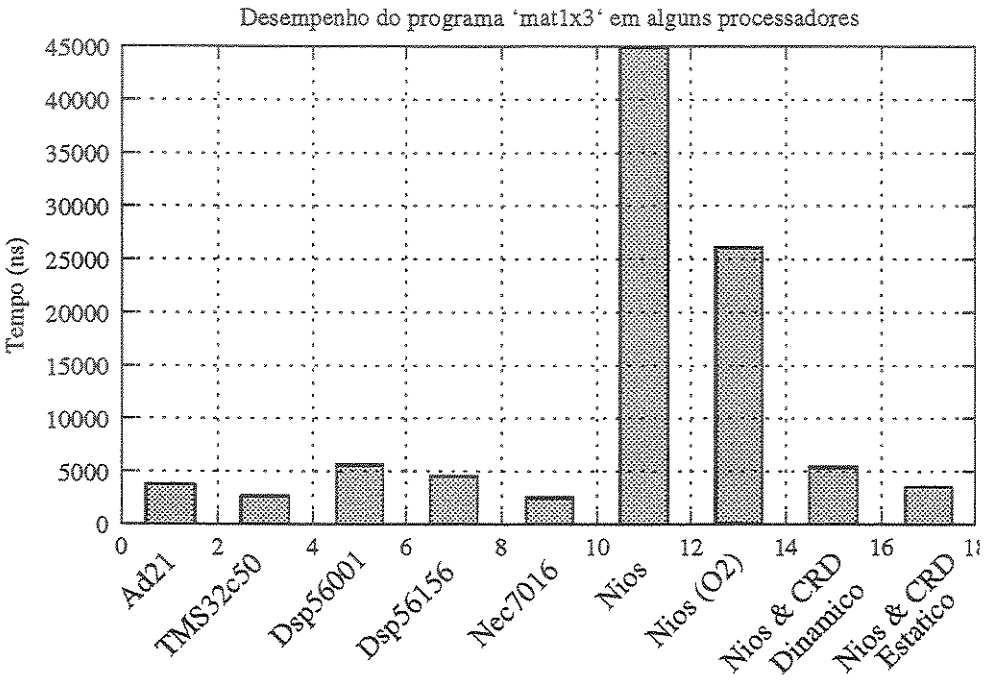


Figura 7.10: Tempo de execução do *kernel* MAT1X3 em nanosegundos

7.9 Complex_update

Conforme vem sendo apresentado até aqui, é apresentada na tabela 7.18.(a) os tempos de execução para o *kernel* COMPLEX_UPDATE em diversos processadores. As linhas de código para os quais foi realizado o *profile* para a tomada de tempos podem são representadas pelas linhas 64 e 72 do Anexo B.6.

Os tempos destinados a configuração, inicialização e execução da instrução apresentada no capítulo anterior (Seção 6.6.1) são apresentados na tabela 7.18.(b). Por se tratar de duas instruções o *overhead* de programação não é único, isto vale também para o

overhead de inicialização, que é duplo. É apresentado na coluna de *overhead* de inicialização, o tempo gasto para a inicialização da primeira chamada (31 ciclos) e da segunda chamada (61 ciclos) perfazendo 92 ciclos. Para a execução foi obtido um total de 11 ciclos para a primeira instrução e 16 ciclos para a segunda, destinadas a substituição do bloco apresentado no Anexo B.6.

Processador	ciclos	tempo(ns)
Ad21	27	1636.3
Tms32c50	38	950
Dsp56001	68	2060.6
Dsp56156	110	1833.3
NEC77016	44	1333.3
Nios	387	11727.3
Nios(O2)	345	10454.5

(a)

	Overhead programação	Overhead inicialização	Ciclos de execução	Total	tempo (ns)
Dinâmico	95	(31)+(61)	(11)+(16)	214	6484.8
Estático	-	(31)+(61)	(11)+(16)	119	3606.1

(b)

Tabela 7.18: Ciclos necessários para: (a) a execução do *kernel* COMPLEX_UPDATE em diversos DSPs; (b) Atrasos inerentes a programação e configuração, assim como tempo dispendido para execução do código em *hardware*

Utilizando a codificação apresentada na seção 6.6.1, experimentou-se ganhos na ordem de 2 a 3 vezes na execução da instrução implementada em *hardware*, conforme é apresentado na tabela 7.19.

Programação do *CRD*	Código otimizado	Código não otimizado
Dinâmico	1.61	1.81
Estático	2.89	3.25

Tabela 7.19: Speedup do sistema *NIOS & CRD* em relação ao processador *NIOS* para o *kernel* COMPLEX_UPDATE

A figura 7.11 apresenta os tempos de execução do *kernel* em diversos processadores. Pode-se notar o melhor desempenho, neste caso, dos DSPs embora o *CRD* apresente ganhos em relação ao *NIOS*.

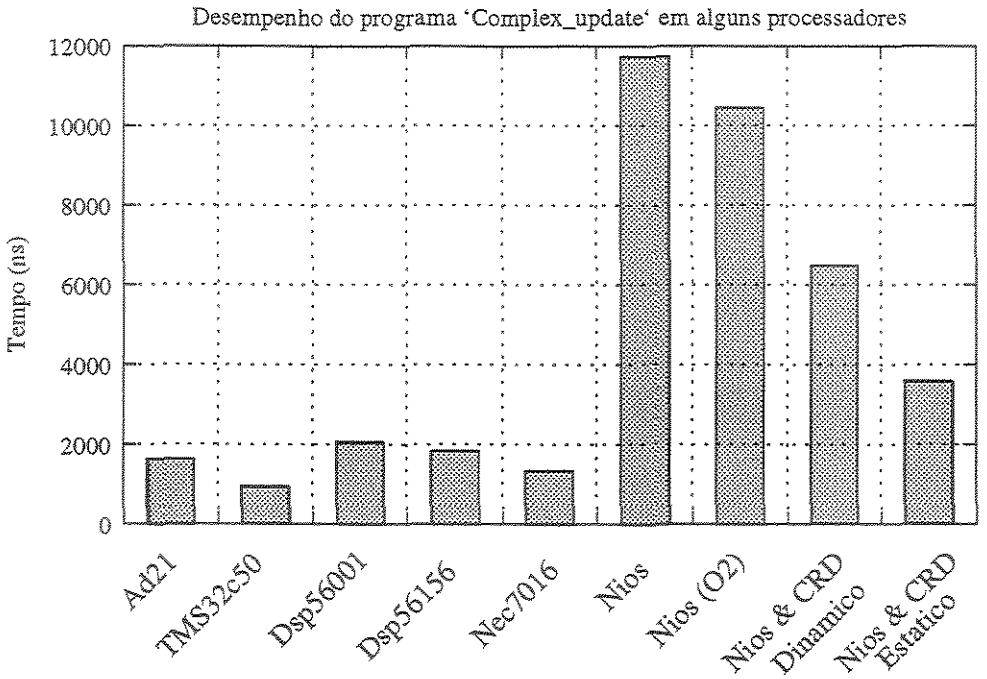


Figura 7.11: Tempo de execução do *kernel* COMPLEX_UPDATE

7.10 N_Complex_Updates

O *kernel* N_COMPLEX_UPDATE realiza n chamadas a computação envolvida no *kernel* COMPLEX_UPDATE (Anexo B.6).

Os ciclos necessários para a execução deste *kernel* são apresentados na tabela 7.20.(a), que reporta valores para um laço com 16 iterações.

Existem diversas possibilidades para a transformação do bloco de código candidato do *kernel* N_COMPLEX_UPDATE, compreendido entre as linhas 74 a 81 do Anexo B.7, em *hardware*. A mais trivial, no entanto, é adicionar as instruções encontradas em COMPLEX_UPDATE no interior de um laço de N iterações. Contudo, conforme apresentado na seção anterior, os ganhos obtidos com a implementação do *kernel* COMPLEX_UPDATE foram baixos, tornando repetidas chamadas desta, uma alternativa longe da mais eficaz do particionamento.

Propõe-se, como um meio de obter ganhos maiores que os apresentados em 7.19, a criação de duas novas instruções, uma delas responsável pelo retorno do vetor da parte real do número, outra da parte imaginária. Os tempos obtidos para programação, configuração

e execução destas instruções são apresentados na tabela 7.20.(b).

Processador	ciclos	tempo(ns)
Ad21	643	38969.7
Tms32c50	600	15000
Dsp56001	1062	32181.8
Dsp56156	2696	44933.3
NEC77016	738	22363.6
Nios	6407	194151.5
Nios(O2)	5614	170121.2

(a)

	Overhead programação	Overhead inicialização	Ciclos de execução	Total	tempo (ns)
Dinâmico	79	24	165	537	16272.7
Estático	-	24	165	409	11484.8

(b)

Tabela 7.20: Ciclos necessários para: (a) a execução do *kernel n_complex_update* em diversos processadores; (b) configuração, inicialização e execução de uma instrução referente ao bloco candidato, em *hardware*

Através da criação de duas instruções em *hardware*, para realizar a tarefa compreendida entre as linhas 74 a 81 do Anexo B.7, alcançou-se desempenho próximo a 20 vezes, quando utilizado configuração estática, conforme é apresentado na tabela 7.21. A figura 7.12 apresenta o tempo despendido na computação da tarefa em diversos processadores, onde pode-se notar a diferença entre o tempo obtido na abordagem tradicional (Software) e na abordagem proposta neste trabalho.

Programação do *CRD*	Código otimizado	Código não otimizado
Dinâmico	10.45	11.93
Estático	19.22	21.94

Tabela 7.21: *Speed-up* do sistema *NIOS & CRD* em relação ao processador *NIOS* para o *kernel* N_COMPLEX_UPDATE

Através da interpolação dos dados obtidos (Figura 7.13) por uma aproximação polinomial, obteve-se uma fórmula aproximada para o número de ciclos gastos tanto para o processador *NIOS* quanto para a abordagem *NIOS & CRD*. A tabela 7.22 apresenta as equações obtidas por interpolação para o número de ciclos gastos na execução deste *kernel* em diversos processadores em função do número de iterações realizadas, que mostram que para grandes valores de n o desempenho do *CRD* tende a ser melhor que os demais processadores.

A figura 7.14 mostra o ganho de desempenho máximo relativo entre o *NIOS & CRD* e os outros processadores. A linha referente ao processador *NIOS* evidencia que o speed-up máximo alcançado é delimitado superiormente por 20. Reforçando os resultados apresentamos alguns cálculos realizados, para a abordagem de reconfiguração dinâmica (onde o *overhead* tende a se dissolver em grandes iterações), quando comparado ao processador *NIOS*[3], onde o speed-up relativo máximo é dado pela equação 7.4

$$speedup = \lim_{n\to\infty} \frac{7 + 400 * n}{216 + 20 * n} = \lim_{n\to\infty} \frac{\overbrace{7/n + 400}}{\underbrace{216/n + 20}} = 20 \tag{7.4}$$

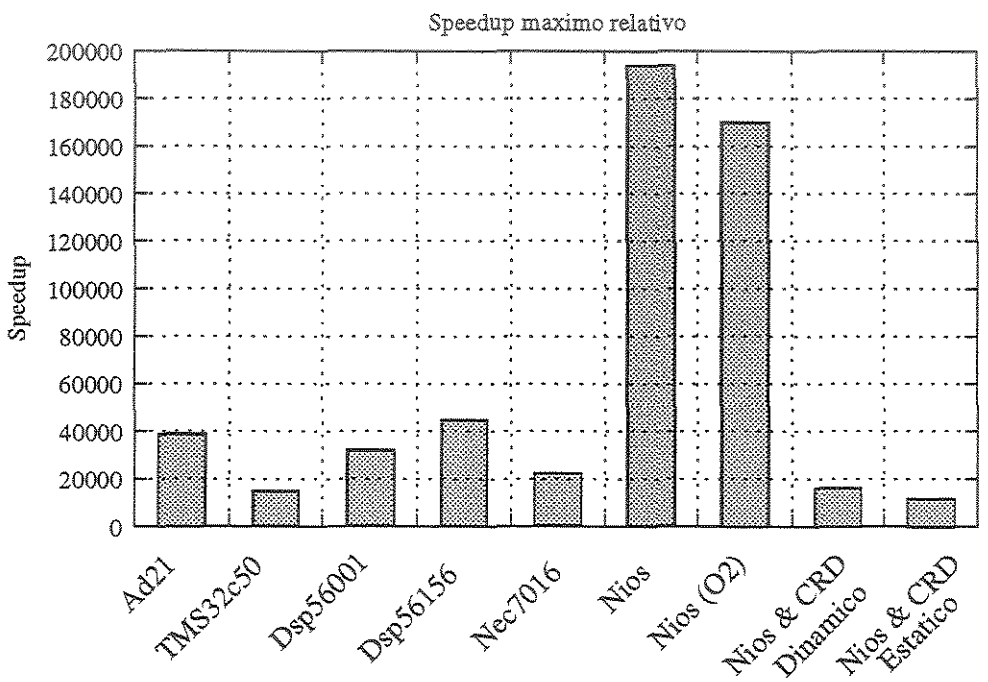


Figura 7.12: Tempo obtido pela execução do *kernel* N_COMPLEX_UPDATE

[3] através das equações interpoladas

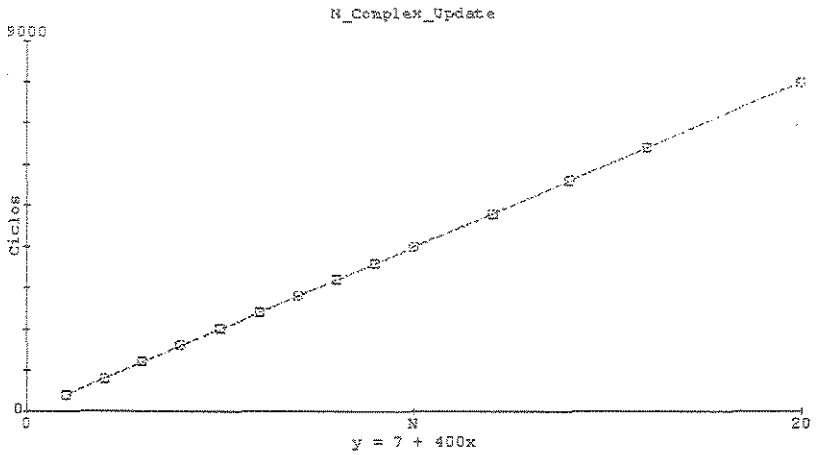


Figura 7.13: Curva resultante da interpolação dos dados obtidos para o processador NIOS

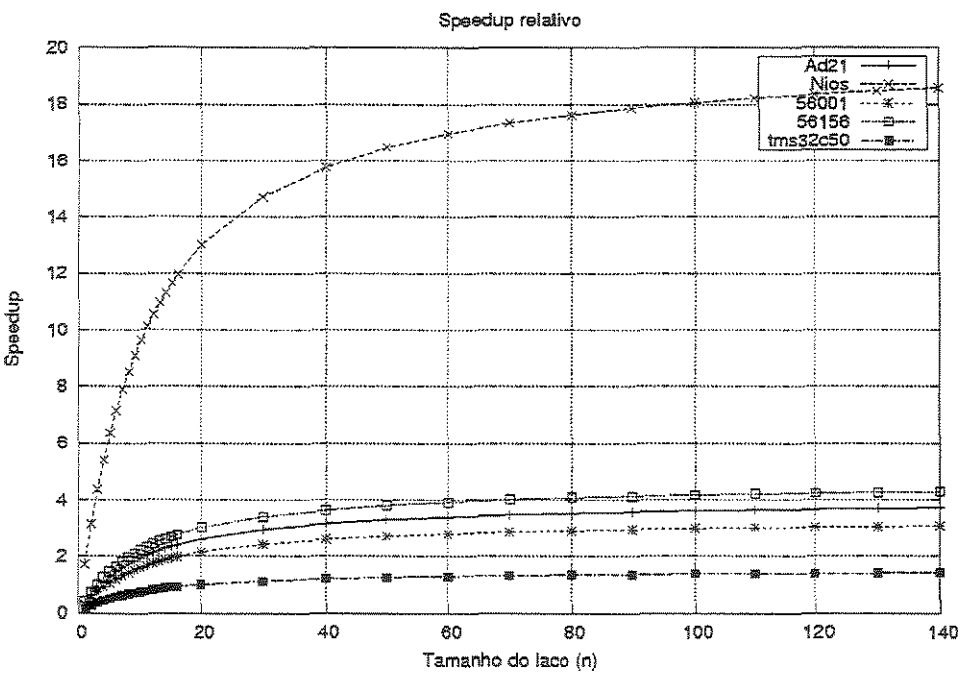


Figura 7.14: *Speed-up* máximo relativo em função do número de iterações do laço

Processador	Equacão
Ad21	$40 * n + 3$
Tms32c50	$37 * n + 8$
Dsp56001	$66 * n + 6$
Dsp56156	$168 * n + 8$
Nios	$400 * n + 7$
Nios & CRD (Dinâmico)	$20 * n + 216$

Tabela 7.22: Equação do número de ciclos decorridos de uma execução de N iterações para diversos processadores

7.11 Convolution

É apresentado nesta parte do trabalho os resultados obtidos com a implementação do *kernel* CONVOLUTION. A tabela 7.23.(a) apresenta o número de ciclos necessários para a execução das linhas (59) a (62) (Anexo B.8) para alguns processadores.

Na tabela 7.23.(b) encontram-se os números de ciclos necessários à configuração e execução da instrução no co-processador *CRD*.

Processador	ciclos	tempo(ms)
Ad21	100	6060
Tms32c50	90	2250
Dsp56001	232	7030.3
Dsp56156	400	6666.6
NEC77016	69	2090.9
Nios	1560	47272.7
Nios(O2)	1431	43363.6

(a)

	Overhead programação	Overhead inicialização	Ciclos de execução	Total	tempo (ms)
Dinâmico	95	26	102	223	6757.5
Estático	-	26	102	128	3878.8

(b)

Tabela 7.23: Número de ciclos necessários: (a) para a execução do *kernel* convolution em diversos processadores; (b) para configurar, inicializar e executar a instrução equivalente ao bloco de instruções candidatas em *hardware*

Seguindo a implementação sugerida no capítulo anterior, promoveu-se um ganho de desempenho de até 12 vezes em relação ao processador *NIOS*, para um número de 16 iterações, conforme é mostrado na tabela 7.24. A figura 7.15 mostra os tempos de execução do *kernel* CONVOLUTION em diversos DSP, no *NIOS* e no *NIOS & CRD*.

Note que mesmo com o uso de configuração dinâmica o *NIOS & CRD* possui tempos de execução comparados aos DSPs.

Programação do *CRD*	Código otimizado	Código não otimizado
Dinâmico	6.41	6.99
Estático	11.18	12.18

Tabela 7.24: *Speed-up* do sistema *NIOS & CRD* em relação ao processador *NIOS* para o *kernel* CONVOLUTION

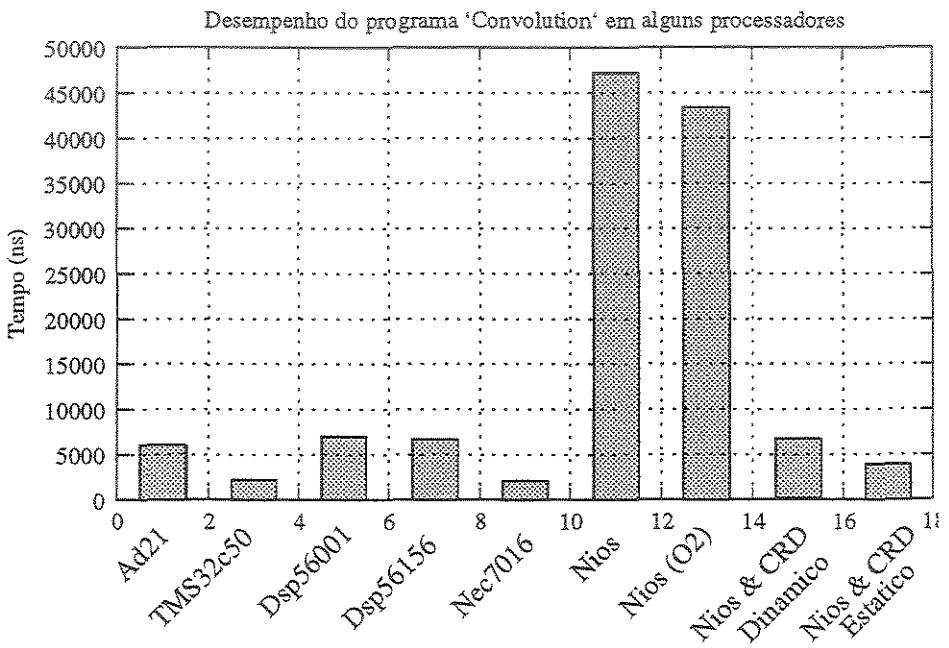


Figura 7.15: Tempo de execução do *kernel* CONVOLUTION

Através da variação do número de iterações do laço, pode-se estimar, utilizando-se a interpolação dos dados, equações polinomiais para o processador *NIOS* e para o conjunto *NIOS & CRD*, conforme pode ser visto na tabela 7.25. Estas equações tornam-se úteis para estimativas relativas de desempenho para a execução de laços com um grande número de iterações.

Processador	Ciclos
Ad21	$6 * n + 4$
Tms32c50	$5 * n + 10$
Dsp56001	$14 * n + 8$
Dsp56156	$24 * n + 16$
Nios	$97 * n + 8$
Nios & CRD	$6 * n + 31$

Tabela 7.25: Equações que descrevem o número de ciclos decorridos de uma execução de N iterações para diversos processadores na execução do *kernel* CONVOLUTION

7.12 Lms

Para a implementação do algoritmo "true LMS" (Fig. 7.16), foram criadas duas novas instruções, uma delas será responsável pelo filtro FIR e pela atualização das variáveis de estado, a outra pela atualização do coeficiente.

As primeira instrução é responsável pela substituição do laço compreendido entre as linhas (130) e (131) do Anexo B.11. A segunda substitui o segundo laço codificado pelas linhas (145) e (146) do mesmo Anexo. Nota-se, no entanto, que o bloco de código compreendido entre as linhas (135) e (143) ficam fora das computações realizadas por estas duas novas instruções.

```
1   Pegar valor de entrada
2   Salvar valor de entrada
3   Realizar filtro FIR
4   Deslocar vetor X
5   Através de d(n), calcular e(n)
6   Atualizar coeficientes
7   Saída de y(n)
```

Figura 7.16: Etapas de execução do algoritmo true LMS

O número de ciclos de execução do algoritmo apresentado na figura 7.16 para diversos processadores é reportada na tabela 7.26.(a). A tabela 7.26.(b) mostra o número de ciclos gastos para configuração (79 ciclos para a primeira e 63 para a segunda instrução), inicialização de valores (30 ciclos para a primeira e 19 ciclos para a segunda instrução) e execução (86 ciclos para a primeira e 72 para a segunda instrução) das instruções adicio-

nadas ao *CRD*.

Processador	ciclos	tempo(ns)
Ad21	248	15030.3
Tms32c50	232	5800
Dsp56001	718	21757.5
Dsp56156	918	15300
NEC77016	249	7545.45
Nios	3766	114121.21
Nios(O2)	3186	96545.45

(a)

	Overhead programação	Overhead inicialização	Ciclos de execução	Total	tempo(ns)
Dinâmico	79	30	$(6+5*n)$	195	16363.6
	+	+	+	+	
	63	19	$(6+4*n)$	152	
Estático	-	30	$(6+5*n)$	116	11818.1
		+	+	+	
		19	$(6+4*n)$	89	

(b)

Tabela 7.26: Número de ciclos necessários: (a) para a computacão do *kernel LMS* em diversos processadores; (b) para *overheads* envolvidos na criação, configuração e chamada/Execução d duas instruções implementadas no *CRD*

Achou-se desnecessário criar uma terceira instrução para a substituição do bloco de código que realiza a computação do *erro* no algoritmo citado. Esta tarefa torna-se dedicada ao processador *NIOS*, fazendo com que haja um processamento misto entre ele e o co-processador *CRD*. O bloco de instruções destinados a realizar esta codificação pode ser visualizado na figura 7.17.

O tempo total de execução do novo bloco misto é apresentado na tabela 7.27, que mostra os resultados tanto para a programação dinâmica (540 ciclos) quanto para a estática (398 ciclos). Os resultados são apresentados sob forma de variáveis onde:

1. $Inst1$ e $Inst2$ significam o tempo gasto para programação, inicialização e execução das instruções 1 e 2.

2. $TSoftware$ significa o tempo gasto para o processador computar as linhas 135 e 143 do anexo B.11 (Apresentado na linha 10 e 11 da figura 7.17)

Programação do *CRD*	Descrição dos *overheads*	Total	Equação
Dinâmico	$Inst1 + Inst2 + TSoftware$	540	$9N + 396$
Estático	$Inst1 + Inst2 + TSoftware$	390	$9N + 254$

Tabela 7.27: Descrição dos ciclos de clock envolvidos na computação da instrução em *hardware*

```
0   ..
1   y = 0 ;
2
3     *p_crd++ = p_x2;            // p_x2 = &X2[1];
4     *p_crd++ = p_x--;           // p_x  = &X[1];
5     *p_crd++ = p_h--;           // p_h  = &H[1];
6     *p_crd   = &y;
7
8     nl = *FIR;
9
10    y += H[0] * (X[0] = x)  ;       // Software
11    error = (d - y) * delta ;       // Software - Total de 193 ciclos
12                                    //            de clock
13
14    p_crd = &CRD_REG[0];
15    *p_crd++ = p_x;
16    *p_crd++ = error;
17    *p_crd   = p_h;
18
19    nl = *UPDT;
20  ...
```

Figura 7.17: Modificações necessárias para a execução do novo bloco de código LMS

Apesar da alternativa de codificação mista entre o processador *NIOS* e o co-processador *CRD*, experimentou-se ganhos de desempenho na ordem de 8 vezes quando comparado a programação estática, conforme é apresentado na tabela 7.28 e ainda desempenho superior a alguns processadores de sinais digitais mesmo para um pequeno número de iterações nos laços implementados, conforme mostrado na figura 7.18

Programação do *CRD*	Código Otimizado	Código não otimizado
Programação Dinâmica	5.9	6.97
Programação Estática	8.16	9.65

Tabela 7.28: *Speed-up* do sistema *NIOS & CRD* em relação ao processador *NIOS* para o *kernel* "LMS"

Após realizar medições do tempo de execução para diferentes valores de n, pôde-se interpolar os dados obtidos encontrando-se uma equação polinomial aproximada que representa o comportamento da computação para o processador *NIOS*, conforme apresentado

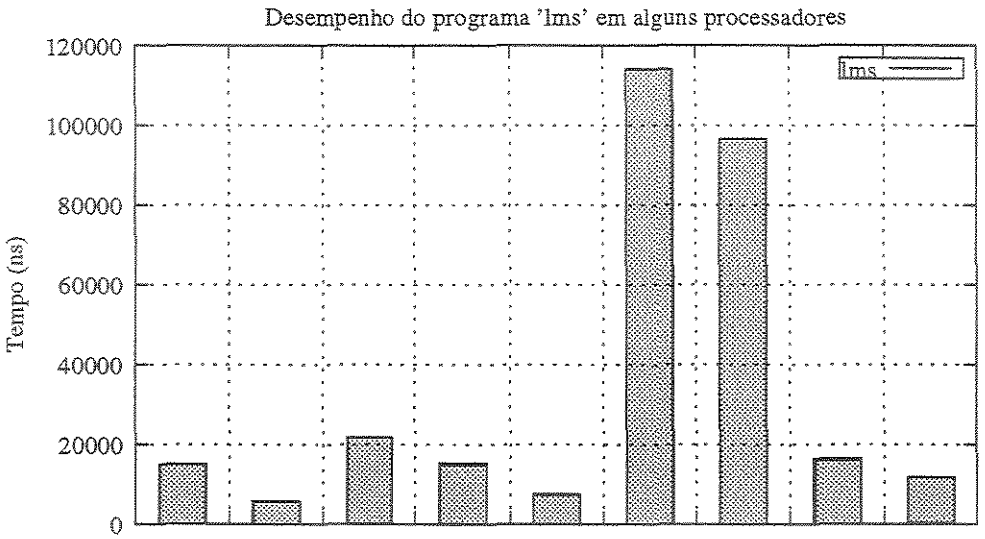


Figura 7.18: Tempo de execução do *kernel* LMS

na tabela 7.29. Desta forma foi possível estabeler ganhos de desempenho relativos para diferentes valores de n do *kernel* "LMS", em configuração dinâmica e estática, conforme é apresentado nas figuras 7.19 e 7.20 respectivamente.

Processador	Ciclos
Ad21	$14 * n + 24$
Tms32c50	$13 * n + 24$
Dsp56001	$42 * n + 46$
Dsp56156	$56 * n + 22$
Nios	$230 * n + 86$

Tabela 7.29: Equação do número de ciclos decorridos de uma execução de N iterações para diversos processadores na execução do *kernel* LMS

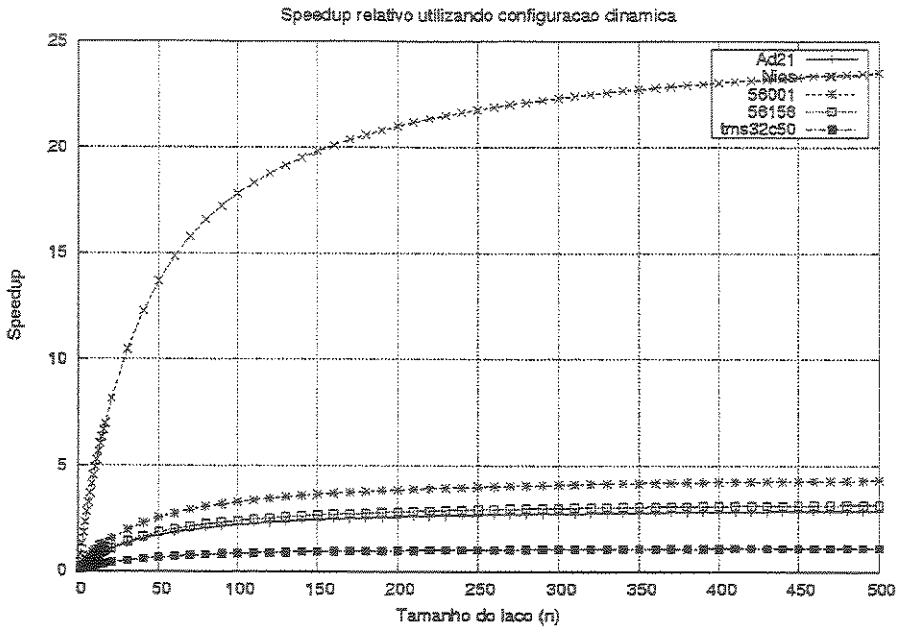


Figura 7.19: *Speed-up* máximo relativo em função do número de iterações do laço, considerando configuração dinâmica

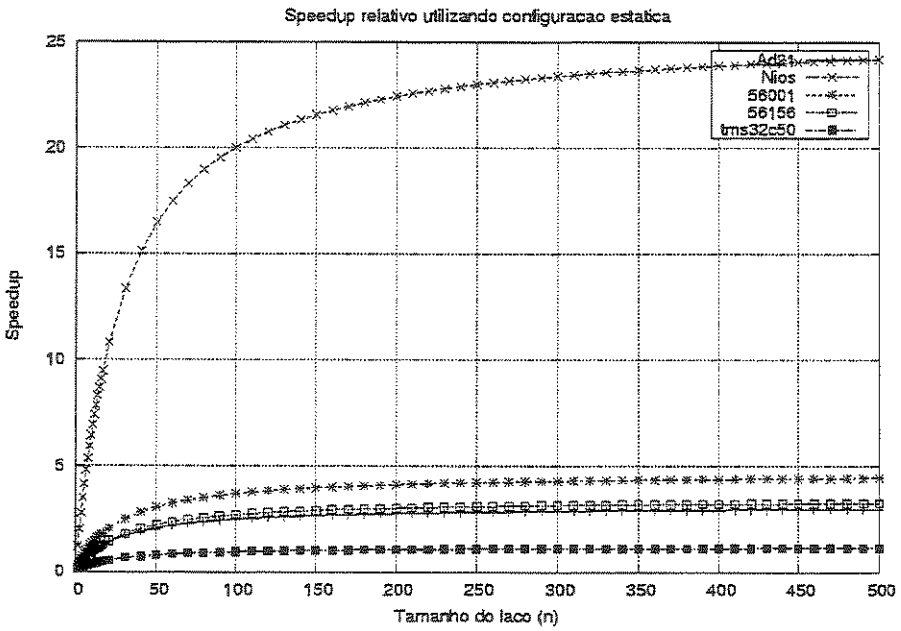


Figura 7.20: *Speed-up* máximo relativo em função do número de iterações do laço, considerando configuração estática

7.13 Matrix1

O *kernel* "Matrix1" realiza uma multiplicação entre duas matrizes A e B de tamanho arbitrário[4]. O bloco de código compreendido entre as linhas 106 a 122 do Anexo B.5, foram submetidos a um *profile* em diversos processadores e o resultado para matrizes de dimensões 10x10 são apresentados na tabela 7.30.(a).

Apesar do bloco de código conter 3 laços, um interno a outro, por motivos já mencionados do decorrer do trabalho foi implementado em *hardware*, apenas o mais interno dos laços. Os ciclos necessários para a configuração e execução da nova instrução pode ser visualizada na tabela 7.30.(b), onde diferentemente do que vinhamos apresentando até aqui, o *overhead* de inicialização da instrução foi omitido por motivações conceituais. *O que deve ser considerado overhead de inicialização e/ou execução ?*

Os *overhead* de inicialização da instrução que deve substituir o bloco compreendido entre as linhas 116 e 117 do Anexo B.6, pode ser contabilizados também como *overhead* de execução do processador *NIOS*, afinal, é necessário o resultado das computações da linha 110 e 112 para que seja possível a inicialização nos registradores do *CRD*. Assim como no caso anterior (Seção 7.12) temos uma computação mista entre *NIOS* e *CRD*, porém, neste caso a tomade de tempo entre tarefas torna-se bem mais complicada, pois uma depende da outra.

Processador	ciclos	tempo(μs)
Ad21	6796	411.87
Tms32c50	7238	180.95
Dsp56001	19494	590.72
Dsp56156	55134	918.9
NEC77016	7864	238.3
Nios	101,187	3066.27
Nios(O2)	94198	2854.48

(a)

	Overhead programação	Ciclos de execução	Total	tempo (μs)
Dinâmico	63	102	7248	219.63
Estático	-	102	7185	217.72

(b)

Tabela 7.30: Número de ciclos necessários para: (a) a execução do *kernel matrix1* em diversso processadores; (b) programação e execução das linhas 116 e 117 do Anexo B.6, em *hardware*

[4]A única restrição é que a dimensão da matriz tem que ser maior que 1

A tabela 7.31 contém os *speed-ups* obtidos na substituição do laço tipo "for", mais interno da computação, pela chamada de uma instrução executada em *hardware*.

Programação do *CRD*	Código otimizado	Código não otimizado
Dinâmico	10.50	11.28
Estático	10.58	11.36

Tabela 7.31: *Speed-up* do sistema *NIOS & CRD* dinâmico e estático em relação ao processador *NIOS* para o *kernel* "Matrix1" utilizando-se matrizes de dimensões 10x10

A figura 7.21 descreve o comportamento da execução das linhas 104 a 124 do Anexo B.6 no processador *NIOS* para computações entre matrizes de dimensão nxn. Procurou-se, através da interpolação dos dados obtidos, encontrar uma aproximação polinomial (Tabela 7.32) para realizar estudos de desempenho com o processador para valores diferentes de n. O mesmo processo foi realizado com a abordagem *NIOS & CRD* conforme pode ser observado na figura 7.22.

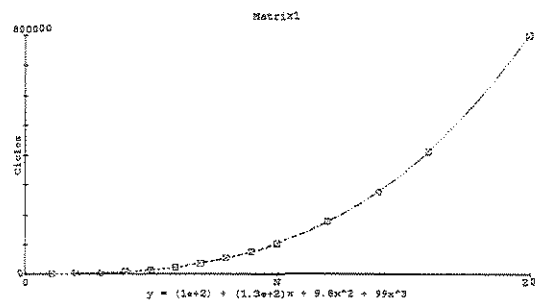


Figura 7.21: Curva resultante na interpolação dos dados obtidos na execução do *kernel* "matrix1" no processador *NIOS*

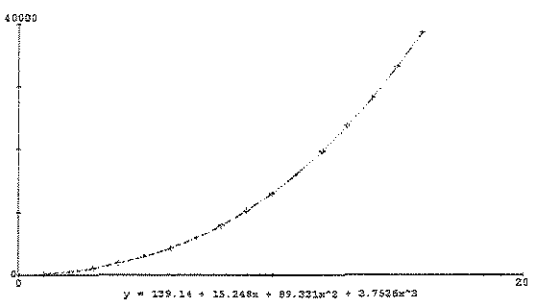


Figura 7.22: Curva resultante na interpolação dos dados obtidos na execução do *kernel* "matrix1" no sistema *NIOS & CRD*

A figura 7.23 mostra o tempo de execução do bloco compreendido entre as linhas (104) a (124) do Anexo B.6 em diversos processadores. Pode-se comprovar que a alternativa *NIOS & CRD* em alguns casos tem desempenho superior a alguns DSPs, mesmo para a tomada de tempo em uma matriz de pequenas dimensões ($x = y = z = 10$). Caso matrizes de maiores dimensões estiverem envolvidas na computação, melhores resultados poderão ser experimentados conforme é apresentado na figura 7.24, onde há ganhos de aproximadamente 4 vezes em cima do processador *DSP56001*.

Processador	Equação em função de n
Ad21	$6 + z(9 + x(7 + 6y))$
Tms32c50	$8 + z(13 + x(21 + 5y))$
Dsp56001	$14 + z(28 + x(32 + 16y))$
Dsp56156	$14 + z(22 + x(44 + 54y))$
Nios	$100 + z(130 + x(9.8 + 99y))$
Nios&Crd	$139,14 + z(15,24 + x(89,33 + 3,75y))$

Tabela 7.32: Equação do número de ciclos necessários para a execução do *kernel* MATRIX1 para diversos processadores

7.14 N_Real_Updates

O *kernel* N_real_updates realiza a operação "real_update" (Seção 7.4) N vezes. A tabela 7.33.(a) apresenta o número de ciclos gastos durante a computação do laço com 16 iterações, compreendido entre as linhas (61) e (62) do Anexo B.13. A tabela 7.33.(b) informa o número de ciclos gastos com a configuração, inicialização e execução da nova instrução em *hardware*.

Processador	ciclos	tempo(ns)
Ad21	131	7939.39
Tms32c50	184	4600
Dsp56001	262	7939.39
Dsp56156	682	11366.67
NEC77016	146	4424
Nios	1607	48696.9
Nios(O2)	1526	46242.4

(a)

	Overhead programação	Overhead inicialização	Ciclos de execução	Total	tempo (ns)
Dinâmico	63	11	70	144	4363.63
Estático	-	11	70	81	2454.54

(b)

Tabela 7.33: Número de ciclos necessários para: (a) a execução das linhas de profiling do Anexo B.13 em diversos processadores; (b) a inicialização, programação e execução da instrução equivalente em *hardware*

Se a codificação do *kernel* "real_update" já obteve ganhos significativos quando implementado através do *CRD*, é de se esperar que a codificação do *kernel* "N_real_update" tenha ganhos de desempenho ainda mais altos, devido as n chamadas da instrução que deverão ser realizadas por *hardware*. A tabela 7.34 apresenta o ganho de desempenho obtido em relação ao processador *NIOS* quando a solução *NIOS & CRD* é adotada.

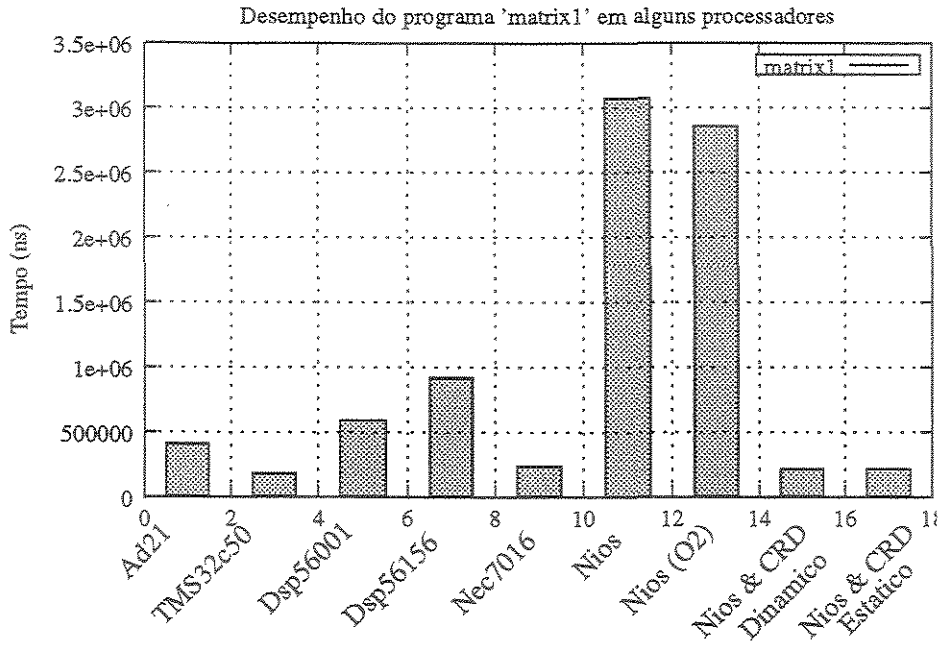


Figura 7.23: Tempo de execução do *kernel* Matrix1 para diferentes sistemas

Programação do *CRD*	Código otimizado	Código não otimizado
Dinâmico	10.59	11.16
Estático	18.84	19.84

Tabela 7.34: *Speed-up* do sistema *NIOS & CRD* em relação ao processador *NIOS* para o *kernel* N_REAL_UPDATE

Pode-se perceber, através da figura 7.25 que a abordagem sugerida, tem desempenho melhor que todas as outras computações, mesmo para um pequeno número de iterações. Realizando uma aproximação polinomial com dados obtidos na variação do número de iterações do laço, pode-se encontrar uma equação aproximada que descreve o comportamento da curva de desempenho para diversos valores de n, conforme é apresentado na tabela 7.35, nos permitindo realizar testes de ganho máximo de desempenho, conforme o apresentado na figura 7.26, onde pode-se visualizar ganhos superiores a 20 vezes, para o processador *NIOS*, quando o n aproxima-se de 250 e de aproximadamente 18 vezes quando comparado ao processador de sinais digitais *56156*.

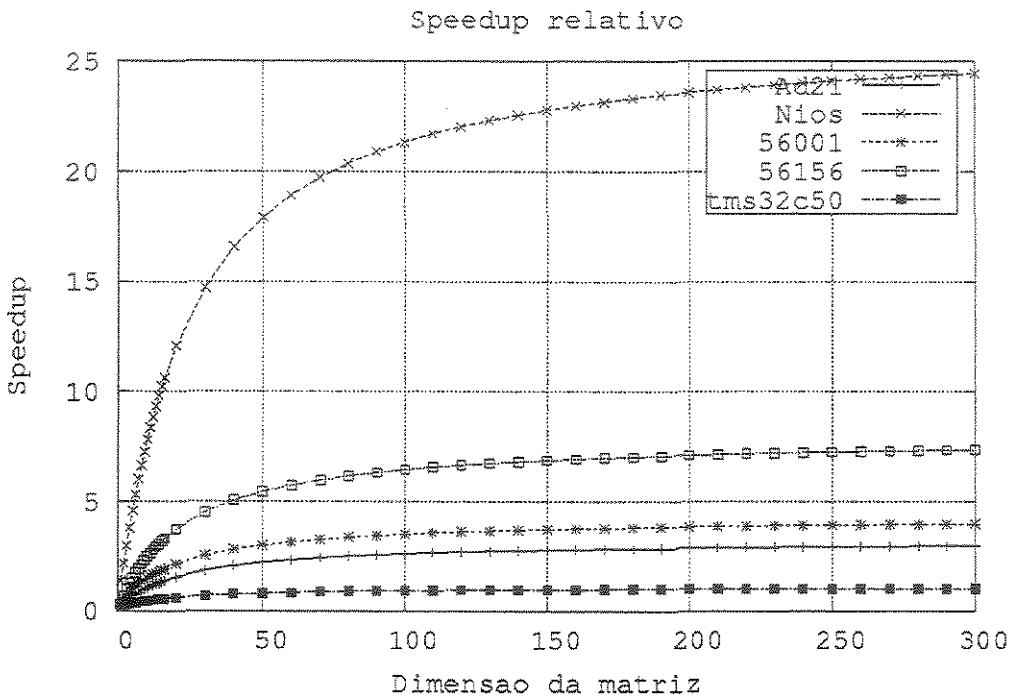


Figura 7.24: *Speed-up* máximo relativo em função do número de iterações do laço

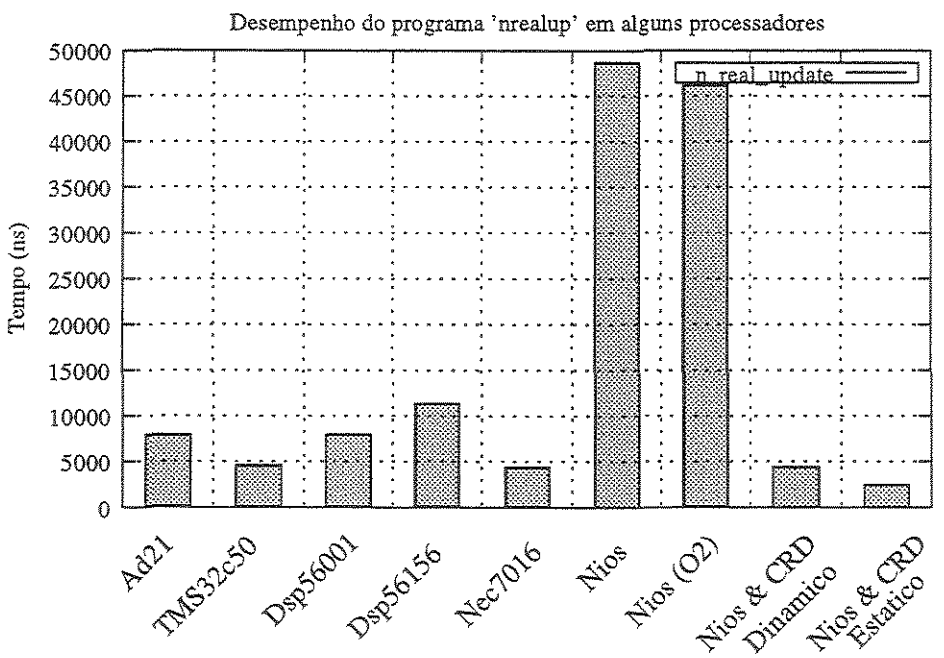


Figura 7.25: Tempo de execução do *kernel* N_Real_updates

Processador	Ciclos
Ad21	$8 * n + 3$
Tms32c50	$11 * n + 8$
Dsp56001	$16 * n + 6$
Dsp56156	$42 * n + 10$
Nios	$100 * n + 7$
Nios & CRD	$4 * n + 64$

Tabela 7.35: Equação do número de ciclos decorridos da execução do *kernel* N_COMPLEX_UPDATE para diversos processadores

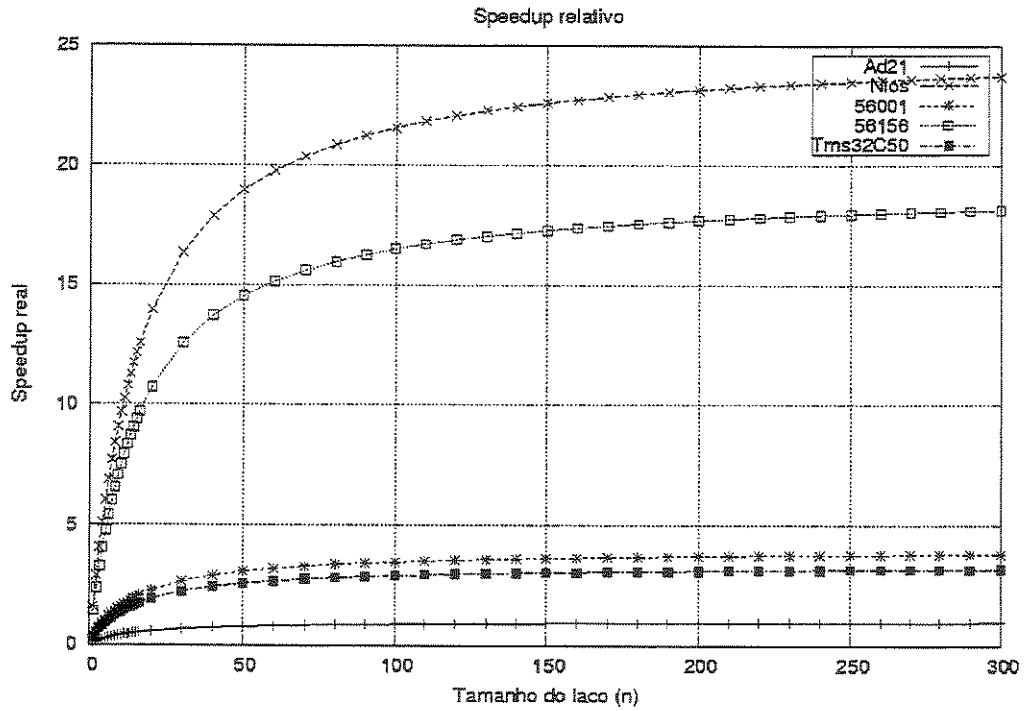


Figura 7.26: *Speed-up* máximo relativo em função do número de iterações do laço

Capítulo 8

Conclusões e Trabalhos Futuros

8.1 Conclusões

Neste trabalho foi apresentado um co-processador que, associado a um processador RISC, é capaz de proporcionar ganhos significativos no desempenho de aplicações dedicadas.

De forma a reduzir a área ocupada pelo co-processador o mesmo foi dotado de uma matriz de interconexões, onde um cojunto de unidades básicas podem ser rearranjadas em tempo de execução, para implementar diferentes *datapaths* capazes de executar diretamente, em hardware, diversos comandos e estruturas típicas de linguagens de alto nível, como os comandos de repetição do tipo *"for"*.

O co-processador é mapeado no espaço de memória do processador *host*, facilitando a sua utilização com diversos processadores de propósito geral existentes no mercado. O uso do co-processador não impede que outras formas para o aumento de desempenho também possam ser utilizadas, como a utilização e combinação de diversos processadores de propósito geral, co-processador reconfigurável e DSP em um mesmo SoC[1].

Os resultados aqui apresentados foram obtidos para uma implementação direta dos trechos de programas escolhidos. Nestas implementações procurou-se ser fiel à codificação original, assim resultados ainda melhores podem ser obtidos com um estudo mais detalhado do trecho de programa a ser implementado em hardware e optando-se por uma implementação mais otimizada como mostrado, por exemplo, para o *kernel* COMPLEX_MULTIPLY.

[1] System on a chip

Durante a fase de testes, verificou-se que o co-processador aliado a ao processador (*NIOS*), obteve ganhos satisfatórios na maioria dos programas do benchmark *DSPStone*, conforme pode ser visto na tabela 8.1 e apresentados sob outra forma, na figura 8.1. Os maiores *speed-ups* são encontrados em programas que fazem uso de laços, principalmente laços internos, esses, na nova abordagem, substituídos por instruções em hardware.

Programa	*Speed-up*	
	Dinâmico	Estático
DOT_PRODUCT	2.95	10.6
COMPLEX_MULTIPLY	1.6	8.86
REAL_UPDATE	1.31	4.45
BIQUAD_ONE_SECTION	1.59	2.41
FIR	7.13	12.39
FIR2DIM	9.81	10.24
MAT1X3	4.77	7.32
COMPLEX_UPDATE	1.61	2.89
N_COMPLEX_UPDATE	10.45	19.22
CONVOLUTION	6.41	11.18
LMS	5.9	8.16
MATRIX1	10.5	10.58
N_REAL_UPDATES	13.26	22.44

Tabela 8.1: *Speed-ups* obtidos na utilização do sistema *NIOS & CRD* no *benchmark* DSPStone

Os resultados obtidos com o uso do sistema *NIOS & CRD*, para o *benchmark* DSPStone, foram comparados com valores de desempenho disponíveis na literatura para diversos DSPs (Digital Signal Processors). Mesmo quando o sistema *NIOS & CRD* está operando a 66% de sua freqüência nominal, obteve-se desempenho superior em diversos casos, como mostrado na tabela 8.2.

Na computação de determinadas aplicações faz-se necessário a utilização de DSPs, ASP (Application Specific Processors) ou ASICs quando um melhor desempenho do sistema é requisitado. No entanto, a especialização de um processador para determinada tarefa traz o inconveniente da falta de flexibilidade ou ser inadequada a outras. Com o intuito de buscar um aumento de desempenho para o maior leque de aplicações possíveis, um sistema reconfigurável pode vir a ser uma abordagem eficiente para várias aplicações.

Outro fator importante, no desenvolvimento de sistemas embarcados é quanto ao

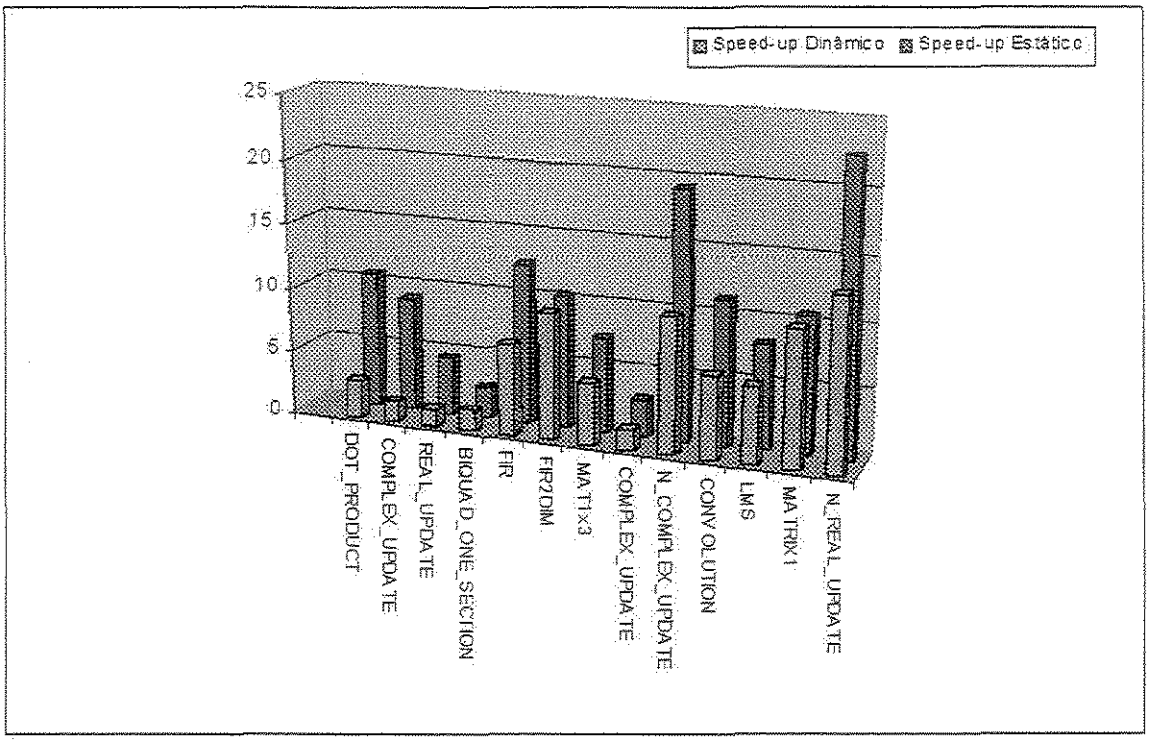


Figura 8.1: Gráfico dos *speed-ups* obtidos na conversão dos *kernels* do *benchamark* DSPS-tone em hardware, utilizando a abordagem *NIOS & CRD*

tempo de desenvolvimento. Em um mercado onde a contínua evolução tecnológica se faz presente, existe a real necessidade do projeto de sistemas embarcados em janelas de tempo cada vez mais estreitas, atrelado a isto, novos produtos têm uma vida útil cada vez mais curta, de modo que o retorno financeiro de seu projeto deve ser obtido em poucos meses [27]. Apresentando o *CRD* sob forma de um IP, pode-se rapidamente prototipar sistemas *SoC*, dimuindo o tempo de projeto e custo de implementação, obtendo-se aumento de desempenho em diversas aplicações, como exemplo, aplicações multimídia que vem ganhando cada vez mais espaço em sistemas de propósito geral e em sistemas embarcados.

8.2 Trabalhos futuros

Uma área que poderia ser muito bem explorada utilizando o *CRD* é a conversão automática de laços internos para aplicação no co-processador reconfigurável.

	AD21	TMS32c50	Dsp56001	Dsp56156	Nec77016
Freqüência	16.5Mhz	40Mhz	33Mhz	60Mhz	33.33Mhz
Programa					
Dot_Product	1.44/0.4	0.87/0.24	1.22/0.34	1.77/0.49	0.88/0.24
Complex_multiply	0.86/0.15	0.38/0.07	0.97/0.17	1.10/0.19	0.48/0.087
Real_Update	0.6/0.18	0.29/0.08	0.8/0.23	0.77/0.23	0.65/0.19
Biquad_one_Section	0.32/0.21	0.127/0.083	0.39/0.26	0.18/0.12	0.186/0.12
Matrix1	1.89/1.87	0.83/0.82	2.71/2.68	4.22/4.18	1.09/1.08
Complex_Update	0.45/0.25	0.26/0.14	0.57/0.31	0.51/0.28	0.36/0.21
N_Complex_Updates	3.14/2.39	1.21/0.92	2.59/1.97	3.62/2.76	1.80/1.37
Convolution	1.56/0.89	0.58/0.33	1.81/1.04	1.71/0.98	0.53/0.31
FIR	2.71/1.56	0.79/0.45	2.51/1.44	0.98/0.57	0.89/0.51
FIR2Dim	2.62/2.51	1.10/1.05	2.47/2.37	3.44/3.3	1.068/1.02
LMS	1.27/0.91	0.49/0.35	1.84/1.32	1.25/0.93	0.63/0.46
MAT1x3	1.06/0.69	0.76/0.49	1.57/1.02	1.27/0.83	0.71/0.46
N_Real_Updates	3.23/1.81	1.87/1.05	3.23/1.81	4.63/2.6	1.8/1.01

Tabela 8.2: *Speed-up* **real** (Estático/Dinâmico) em relação a diversos tipos de DSPs para valores de testes fixados no *benchmark* DSPstone

O compilador seria responsável por, em tempo de compilação, realizar o profile do código e a troca por palavras de programação ao código menos eficaz, ou então a todos laços mais internos encontrados. Atualmente cabe ao programador substituir as instruções desejadas por seu equivalente em hardware.

Um trabalho que pode ser explorado usando o *CRD* é o desenvolvimento de *datapath* com tamanho de ciclo variável. Durante a etapa de síntese do *CRD* sabe-se o tempo necessário para que cada unidade do *CRD* apresente uma computação válida, já em outra etapa, na contrução de determinado *datapath*, a linguagem *CDRL*, através da palavra de programação estima o período necessário para que a computação seja realizada, informando ao programador caso o *datapath* desenvolvido tenha problemas quanto ao caminho crítico. Uma forma de se aumentar ainda mais o desempenho do *CRD* é fornecer ao co-processador a freqüência dada pelo menor período atingido na etapa de síntese, e conforme o *datapath* for criado, a *CDRL* (por meio da palavra de programação) informe a quantidade de ciclos que o novo *datapath* precisará, diminuindo desta forma o tempo necessário para que seja fornecido o resultado previsto no final do *datapath*.

Com o intuito de se obter resultados ainda melhores, pode-se concentrar esforços

na modificação do *datapath* do processador de propósito geral (*host*) a fim de diminuir a latência na chamada e execução da instrução no co-processador. Novas versões do processador NIOS já permitem a criação de novas instruções a partir de sua configuração na ferramenta de trabalho (*Quartus II*).

8.2.1 Dificuldades encontradas

Para a execução de um pequeno pedaço de código devidamente particionado no sistema *NIOS & CRD* é necessária uma grande harmonia entre todas as entidades envolvidas, o processador de propósito geral e seus componentes, o código a ser compilado, a linguagem intermediária, a síntese e *design* do co-processador reconfigurável, o terminal de comunicação com o sistema, entre outros. Apenas um destes operando de maneira errônea, acarretará erro ao sistema como um todo. Como a modificação geralmente se dava em todas as partes do processo, ao final de uma execução, um erro tornava obrigatória a vista em quase todas as partes do sistema, tarefa na maioria das vezes muito cansativa.

O fato da linguagem intermediária, responsável pela codificação do *datapath* do código a ser implementado em hardware, não ter sido completamente terminada, trouxe um aumento no tempo na execução dos testes. Os elementos de 64 bits, responsáveis pela configuração dos novos *datapaths* foram estabelecidos bit a bit, ocasionando esporadicamente alguns equívocos.

Mudanças no projeto inicial

Com o intuito de diminuir o número de bits da palavra de configuração, resolveu-se colocar o multiplicador no interior da *ULA*, que mostrou-se um equívoco no decorrer do projeto. Da forma que foi idealizado inicialmente, o *CRD* torna-se incapaz de realizar uma instrução tipo *MAC* de forma eficiente. Da forma que foi concebido, é possível apenas, a realização de uma operação por vez na *ULA* do *CRD*. Com algumas modificações torna-se possível a realização da instrução pela *ULA* e pelo multiplicador simultaneamente.

Outro fator que tornou-se bastante limitado no decorrer dos testes foi o incremento dado pelo contador interno do *CRD*. Pequenas modificações no projeto do contador do *CRD* poderiam ter sido levadas em conta para um melhor arranjo do particionamento,

tais como passo de contagem crescente, decrescente e diferente de uma unidade, para um melhor aproveitamento com operações em linhas de matrizes.

8.3 Contribuições

Neste trabalho foi proposto uma arquitetura reconfigurável disposta em um co-processador mapeado em memória, que foi desenvolvido como evolução de alguns trabalhos publicados anteriormente, juntamente com um pré-processador de uma linguagem responsável pela configuração e execução das instruções que serão executadas no co-processador.

Como fator de destaque no trabalho, pode-se citar a proposta de rearranjo dos blocos funcionais, através de uma matriz de multiplexadores, diferenciando-se dos trabalhos publicados até então, assim como a unidade de controle de laços e instruções que determina o final de uma instrução multi-ciclo ou o final de laço.

A facilidade de conexão com um processador genérico RISC, através de um co-processador, apresentado sob forma de um IP, mapeado em memória e de pequenas dimensões também podem ser evidenciados.

Referências Bibliográficas

[1] A. Lynn Abbott, Peter M. Athanas, Luna Chen, and Robert L. Elliot. Finding lines and building pyramids with splash 2, April, 1994. In Duncan A. Buell and Kenneth L. Pocek, editors, Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines.

[2] 2003. Advanced Micro Devices - AMD29050 datasheet.

[3] N. Shirazi; P. Athanas and A. Abbott. Implementation of a 2-d fast fourier tansform on an fpga-based custom computing machine, 1995.

[4] P. Athanas and A. Abbott. Real-time image processing on a custom computing platform, February 1995. pages 16 24.

[5] Peter M. Athanas. Processor reconfiguration through instruction-set metamorphosis, 1993. volume 26; pages 11-18.

[6] Ray Bittner; Peter Athanas. Wormhole run-time reconfiguration., 1999.

[7] Zhi Alex Ye; Andreas Moshovos; Scot Hauck; Prithviraj Banerjee. Chimaera : A high-performance architecture with a tightly-coupled reconfigurable functional unit, 2000.

[8] Michael Baxter. Icarus: A dynamically reconfigurable computer architecture., 1999.

[9] R.A et alii Bergamaschi. Automating the design of socs using cores, 2001. IEEE Design Test of Computers, Vol. 18, No. 5, Set/Out. 2001. pp 32-45.

[10] Arnold S. Berger. Embedded systems design: An introduction to processes tools, techniques, 2002. CPM Books.

[11] Duncan A. Buell, Je rey M. Arnold, and Walter J. Kleinfelder. Splash 2: Fpgas in a custom computing machine, 1996. IEEE Computer Society Press.

[12] R. Buyya. High performance cluster computing: Architectures and systems, 1999. Prentice Hall Nj, USA.

[13] Srihari Cadambi, Jeffrey Weener, Seth Copen Goldstein, Herman Schmit, and Don ald E. Thomas. Managing pipeline-reconfigurable fpgas, February, 1998. In Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays.

[14] 2004. CHAMELEON Project. Laboratório de Sistemas de Computação - IC UNICAMP
http://www.lsc.ic.unicamp.br.

[15] R. D. Witting; P. Chow. Onechip : An fpga processor with reconfigurable logic, 1996. Editors J. Arnold and K. L. Pocek. pages 126-135.

[16] Y. Chung and V. Prasanna. Parallel object recognition on an fpga-based configurable computing platform, Oct. 1997.

[17] B. Cmelik. Spixtools introduction and user's manual, 1993. Techinical Report SMLI TR-93-6. Sun Microsystems Laboratory, Mountaim View, CA.

[18] Altera Corporation. Nios programmer's reference manual, 2001.

[19] Darren C. Cronquist, Paul Franklin, Stefan G. Berg, and Carl Ebeling. Specifying and compiling applications for rapid, April 1998. In Kenneth L. Pocek and Je rey M. Arnold, editors, Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines.

[20] A. Dandalis and V. Prasanna. Fast parallel implementation of dft using configurable devices., Sept 1997.

[21] Andre DeHon. Dpga utilization and application. In *Proceedings of the 1996 ACM fourth international symposium on Field-programmable gate arrays*, pages 115–121. ACM Press, 1996.

[22] E. Mirsky; A. DeHon. Matrix : a reconfigurable computing architecture with configurable instruction distribution and deployable resources, 1996.

[23] Al Dewey. The vhsic hardware description language (vhdl) program. In *Proceedings of the 21st conference on Design automation*, pages 556–557. IEEE Press, 1984.

[24] K. Diefendorff and P.K. Dubey. How multimedia workloads will change processor design, 1997. IEEE Com-puter Magazine, pp. 43-45.

[25] Dspstone benchmark.
http://www.ert.rwth-aachen.de/Projekte/Tools/DSPSTONE/dspstone.html.

[26] Dspstone benchmark report.
http://www.ert.rwth-aachen.de/Projekte/Tools/DSPSTONE/report.ps.gz.

[27] Luigi Carro e Flávio Rech Wagner. Sistemas computacionais embarcados, 2003. Minicurso da Reunião anual da SBC - Campinas/SP.

[28] Carl Ebeling, Darren C. Cronquist, and Paul Franklin. Rapid: Reconfigurable pipeline datapath, August 1996. In Reiner W. Hartenstein and M. Glesner, editors, Proceedings of the 6th International Workshop on Field-Programmable Logic and Compilers.

[29] Carl Ebeling, Darren C. Cronquist, Paul Franklin, Jason Secosky, and Stefan G. Berg. Mapping applications to the rapid configurable architecture, April 1997. In Kenneth L. Pocek and Je rey Arnold, editors, Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines.

[30] D. Edenfeld, A Kahng, M. Rodgers, and Y. Zorian. 2003 technology roadmap for semiconductors, 2004. Published by the IEEE Computer Society. 47-56.

[31] Hartej Singh; Ming-Hau Lee; Guangming Lu; Fadi J. Kurdahi; Nader Bagherzadeh; Eliseu M. Chaves Filho, 1999.

[32] 2003. Free Software Foundation
GNU Compiler Collection
http://gcc.gnu.org.

[33] Charlé R. Rupp; Mark Landguth; Tim Garverick; Edson Gomersal; Harry Holt; Jerrey M. Arnold; Maya Gokhale, 1998.

[34] R. Hartenstein. A decade of reconfigurable computing: a visionary retrospective, 2001. Design, Automation and Test in Europe, Munich, Alemanha, Mar. 2001. Proceedings, IEEE Computer Society Press, 2001.

[35] John R. Hauser, 1997. The GARP architecture.

[36] Berry Kane; Joe Heinrich. *MIPS RISC Arcìtecture*, volume 1. Prentice Hall, 1992.

[37] David A. Patterson; John L. Hennessy. *Computer Organization & Design - The Hardware/Software Interface*, volume 1. Morgan Kaufmann, 1998.

[38] Dzung T. Hoang and Daniel P. Lopresti. FPGA implementation of systolic sequence alignment. In Herbert Grünbacher and Reiner W. Hartenstein, editors, *Field-Programmable Gate Arrays: Architectures and Tools for Rapid Prototyping*, pages 183–191. Springer-Verlag, Berlin, 1992.
citeseer.nj.nec.com/hoang92fpga.html.

[39] Bernardo Kastrup; Arjan Bink; Jan Hoogerbrugge. Concise : A compiler-driven cpld-based instruction set accelerator, 1999.

[40] M. J. Wirthlin; B. L. Hutchings. Disc : The dynamic instruction set computer., 1995.

[41] Z. Hwang, K.; Xu. Scalable parallel computing: Technology, architecture, programming ,, 1998. McGraw-Hill.

[42] Intel processors.
http://developer.intel.com/hardwaredesign/processors.htm.

[43] C. Iseli and E. Sanchez. Spyder: a sure (superscalar and reconfigurable) processor., 1995. Journal of Supercomputing,9(3):231-252.

[44] Jack Jean, Xuejun Liang, Brian Drozd, and Karen Tomko. Accelerating an ir automatic target recognition application with fpgas, April, 1999. In Kenneth L. Pocek and

Jef-frey M. Arnold, editors, Proceedings of the Seventh Annual IEEE Symposium on Field- Programmable Custom Computing Machines.

[45] S. Hauck; T. W. Fry; M. M. Hosler; J. P. Kao. The chimaera reconfigurable functional unit, 1997.

[46] C.E. Kozyrakis and D.A. Patterson. A new direction for computer architecture research, Nov. 1998. IEEE Computer Magazine.

[47] S. Kumar, L. Pires, D. Pandalai, and H. Spaanenburg. Bechmarking technology for configurable computing system, 1998. In IEEE Sysmposium on Field-Programmable Custom Computing Machines.

[48] S. Goldstein; H. Schmit; M. Moe; M. Budiu; S. Cadambi; R. Taylor; R. Laufer. Piperench: A coprocessor for streaming mulimedia acceleration, 1999.

[49] Ming-Hau Lee, Hartej Singh, Guangming Lu, Nader Bagherzadeh, Fadi J. Kurdahi, Eliseu M. C. Filho, and Vladimir Castro Alves. Design and implementation of the morphosys reconfigurable computing processor. *J. VLSI Signal Process. Syst.*, 24(2-3):147–164, 2000.

[50] E. Lemoine and D. Mereceron. Run time reconfiguration of fpga for scanning genomic databases, 1995.

[51] David Dahle Leslie. The ucsc kestrel general purpose parallel processor. citeseer.nj.nec.com/370266.html.

[52] H. Lewis, T. G.; El-Rewini. Distributed and parallel computing, 1998. Manning.

[53] Yanbing Li, Tim Callahan, Ervan Darnell, Randolph Harr, Uday Kurkure, and Hon Stockwood. Hardware-software co-design of embedded reconfigurable architectures, 2000. In 37th conference on Design automation, pages 507-512. Annual ACM IEEE Design Automation Conference.

[54] Magarshack. Improving soc design quality through a reproducible design flow, 2002. IEEE Design Test of Computers, Vol. 19, No. 1, Jan-Fev. 2002, pp 76-83.

[55] Alan Marshall, Tony Stans eld, Igor Kostarnov, Jean Vuillemin, and Brad Hutchings. A reconfigurable arithmetic array for multimedia applications, In Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays. February 1999.

[56] Peter Marwedel. Embedded system design. kluwer academic publishers, 2003. ISBN 1-4020-7690-8
http://ls12-www.cs.uni-dortmund.de/ marwedel/kluwe-es-book/.

[57] 2004. Mentor Graphics - Leonardo Spectrum Manual
http://www.mentor.com/leonardospectrum/datasheet.pdf.

[58] Takashi Miyamori and Kunle Olukotun. Remarc (abstract): reconfigurable multimedia array coprocessor. In *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, page 261. ACM Press, 1998.

[59] G. E. Moore. Cramming more components onto integrated circuits, 1965. Electronics Magazine, Vol. 38, Abr. 1965. pp 114-117.

[60] Emeka Mosanya and Eduardo Sanchez. A fpga-based hardware implementation of generalized profile search using online arithmetic, Februery 1999. In Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays.

[61] Freescale documentation library.
http://www.freescale.com/webapp/sps/library/docuñib.jsp.

[62] Steven S. Muchnick. *Advanced compiler design implementation*, volume 1. Morgam Kaufamann, 1997.

[63] S. J. Mullender. Distributed-operating systems, 1996. ACM Computing Surveys, Vol. 28, No. 1, March.

[64] D. Roncin P. Bertin and J. Vuillemin. Programmable active memories, 1989. Technical Report 3, Digital Equipment. Paris Research Laboratory.

[65] G. Pfister. In search of clusters, 1998. Prentice Hall.

[66] A. Dandalis; V. Prasanna and J. Rolim. An adaptive cryptografic engine for ipsec architectures., (Submited, April 2000).

[67] R. S. Pressman. Software engineering: A practitioner's approach, 2001. McGraw Hill, 5th edition, November.

[68] Reconfigurable computing definition. *http://www.acm.uiuc.edu/sigarch/projects/reconf/report$_1$.html.*

[69] Michael Rencher and Brad L. Hutchings. Automated target recognition on splash-2., April 1997. In Kenneth L. Pocek and Je rey Arnold, editors, Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, pages 192..200.

[70] P. I. Mackinlay; P. Y. K. Cheung; W. Luk; R. D. Sandiford. Reiley-2 : A flexible platform for codesign and dynamic reconfigurable computing research. In M. Glesner W. Lik, P. Y. K. Cheung, editor, *Field Programmable Logic and Applications*, volume 1304, pages 91–100. LNCS, 1997.

[71] Ronald Laufer; R. Reed Taylo; Herman Schmit. Pci-piperench : and the swordapi : A system for stream based reconfigurable computing, 1999.

[72] Moe Shahdad. An overview of vhdl language and technology. In *Proceedings of the 23rd ACM/IEEE conference on Design automation*, pages 320–326. IEEE Press, 1986.

[73] C. Lee; M. Potkonjak; W.H. Smith. Mediabench: A tool for evaluating and sythesizing multimedia and communications systems, 1997. International Symposium on Microarchitecture, Micro 30, pages 292-303, Research Triangle Park, NC.

[74] G. Memmik; W.H. Smith and W. Hu. Netbench: A benchmarking suite for network processors, 2001. In Proceedings of International Conference on Computer-Aided Design(ICCAD), Pages 39-42, San Jose, CA.

[75] R. Razdan; M. D. Smith. A high-performance microarchitecture with hardware-programmable functional units, 1994.

[76] Specint benchmark, 2000.
http://www.specbench.org/cpu2000/.

[77] A. Srivastava and A. Eustace. Atom: A system for building customized program analysis tools, 1994. ACM conference on Programming Language Design and Implementation, Pages 196-205, Orlando, FL.

[78] Maya B. Gokhale; Janice M. Stone. Napa c : Compiling for a hybrid risc/fpga architecture, 1998.

[79] Visual instruction set (vis) manual
sun microsystems inc.
http://www.sun.com.

[80] Roy A. Sutton, Vason P. Srini, and Jan M. Rabaey. A multiprocessor dsp system using paddi-2. In *Proceedings of the 35th annual conference on Design automation*, pages 62–65, 1998.

[81] Dominic Sweetman. *See Mips run*, volume 2. Morgan Kaufmann, 1 edition, 1999.

[82] Deependra Talla and Lizy K. John. Performance evaluation and benchmarking of native signal processing, 1999. Laboratory for Computer Architecture, Department of Electrical and Computer Engineering, The University of Texas at Austin.

[83] Deependra Talla and Lizy Kurian John. Execution characteristics of multimedia applications on a pentium ii processor, 2000. Laboratory of Computer Architecture. Department of Electrical and Computer Engineering - The University of Texas at Austin.

[84] Texas instruments inc. tms320c6000 assembly benchmarks at texas instruments: C64x dsp benchmarks, 2003.
http://www.ti.com/sc/docs/products/dsp/.

[85] Tinyrisc processor manual.
http://www.lsilogic.com/.

[86] J. Vuillemin; P. Bertin; D. Roncin; M. Shand; H. Touati and P. Boucard. Programmable active memories: Reconfigurable systems come of age, March, 1996. 4(1):56 69.

[87] 2004. TRIMEDIA Processors
http://www.trimedia.com.

[88] Alfred V. Aho; Ravi Sethi; Jeffrey D. Ullman. **Compilers**, *Principles, Techniques and tool*, volume 1. Addison Wesley, 1985.

[89] Dinesh C. Suresh; Walid A. Najjar; Frank Vahid; Jason R. Villarreal and Greg Stitt. Profiling tools for hardware/software partitioning of embedded applications, 2003. LCTES, Pages 189-198, San Diego, California.

[90] E. Waingold, M. Taylor, V. Sarkar, V. Lee, W. Lee, J. Kim, M. Frank, P. Finch, S. Devabhaktumi, R. Barua, J. Babb, S. Amarsinghe, and A. Agarwal. Baring it all to software: The raw machine. Technical report, 1997.

[91] J. R. Hauser; J. Wawrzynek. Garp : A mips processor with a reconfigurable coprocessor, 1997.

[92] J. R. Hauser; J. Wawrzynek. *Augmenting a Microprocessor with Reconfigurable Hardware.* PhD thesis, UNIVERSITY OF CALIFORNIA, BERKELEY, 2000.

[93] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, and S. Ghosh. Prism-ii compiler and architecture, April, 1993. In Duncan A. Buell and Kenneth L. Pocek, editors, Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines.

[94] Michael J. Wirthlin and Brad L. Hutchings. Sequencing run-time reconfigured hardware with software, February 1996. In Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays.

[95] W. Wolf. Modern vlsi design: A system approach, 1994. Englewood Cliffs, Prentice Hall.

[96] Michael Wolfe. *High Performance compiler for parallel computing*, volume 1. Addison-Wesley, 1996.

[97] Xilinx technical documentation.
http://www.xilinx.com/support/library.htm.

[98] Yamauchi, Shogo Nakaya, and Nobuki Kajihara. Sop: A reconfigurable massively parallel system and its control-data-flow based compiling method, April 1996. In Kenneth L. Pocek and Je rey Arnold, editors, Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines.

[99] E. Yourdon. Análise estruturada moderna, 1990.
Campus. ISBN 8570016158.

Apêndice A

Diretivas de compilação para a tomada dos dados

Os resultados mostrados nas tabelas do capítulo anterior, foram adquiridos através da tomada de tempo, de execução dos programas, em alguns processadores, com as seguintes chaves de compilação.

- Ad21

  Compiling command (C files):

```
g21 $*.c -O2 -a intmem_only -map -Wall -mstatic-spill -save-temps
-D__ADSP2101__ -runhdr ${ADI_DSP}/21xx/lib/2101_hdr.obj -o $*.exe
```

- DSP56156

  Compiling command:

```
g561c -alo -S -O -Wall -D__DSP56156__ -D__PROF56__
  -DSTART_PROFILING='__asm("START_PROFILING:")'
  -DEND_PROFILING='__asm("END_PROFILING:")' $*.c
g561c -alo -asm '-occ -L' $*.asm -o $*.cld
```

- TMS320C50

Compiling and linking command: (assembler reference code)

```
dspa -w -s $*.asm -l -v50 -d__TMS320C50__
dsplnk $*.cmd
```

- DSP56001

  Compiling command:

```
g56k -alo -S -O -D__DSP56000__ -D__PROF56__
  -DSTART_PROFILING='__asm("START_PROFILING:")'
  -DEND_PROFILING='__asm("END_PROFILING:")' $*.c
g56k -alo -asm '-occ -L' $*.asm -o $*.cld
```

- NEC μPD770xx

  Compiler: CC77016 C Compiler Version 1.12 Intermetrics 1993

- Nios

  Compiling command:

```
nios-elf-gcc -I ../inc -I ../../inc -I ../../../inc
-m32 $*.c -o $*.c.o -c
```

Apêndice B

Programas benchmark DSPstone

B.1 Dot_Product

```
0   /*
1    * benchmark program:   dot_product.c
2    *
3    * benchmark suite:     DSP-kernel
4    *
5    * description:         dot product benchmarking
6    *
7    * This program performs a dot product of the form Z=AB,
8    * where A is a [1x2] vector and B is a [2x1] vector.
9    *
10   *          A[1 x 2] * B[2 x 1] = Z
11   *
12   * vector A[1 x 2]= |a1 a2|
13   *
14   * vector B[2 x 1]= | b1 |
15   *                  | b2 |
16   *
17   * dot product Z = a1*b1 + a2* b2
18   *
19   * vector elements are stored as
20   *
21   * A[1 x 2] = { a1, a1 }
22   *
23   * B[2 x 1] = { b1, b2 }
24   *
25   *
26   * reference code:       none
27   *
28   * f. verification:      with printf function
29   *
30   * organization:         Aachen University of Technology - IS2
31   *                       DSP Tools Group
32   *                       phone:  +49(241)807887
```

```
33    *                          fax:    +49(241)8888195
34    *                          e-mail: zivojnov@ert.rwth-aachen.de
35    *
36    * author:                  Juan Martinez Velarde
37    *
38    * history:                 10-05-94 C Code creation (Martinez Velarde)
39    *
40    *                          $Author: schraut $
41    *                          $Date: 1995/01/26 11:10:35 $
42    *                          $Revision: 1.2 $
43    */
44
45    #define STORAGE_CLASS  register
46    #define TYPE           int
47
48    void pin_down(TYPE *Z)
49    {
50      *Z = 0 ;
51    }
52
53    TYPE main()
54    {
55      static  TYPE A[2] = {2,1} ;
56      static  TYPE B[2] = {2,5} ;
57      static  TYPE Z    = 0   ;
58
59      STORAGE_CLASS TYPE *p_a = &A[0] ;
60      STORAGE_CLASS TYPE *p_b = &B[0] ;
61      STORAGE_CLASS TYPE *p_z = &Z ;
62
63      STORAGE_CLASS TYPE f ;
64
65      pin_down(&Z) ;
66
67      START_PROFILING;
68
69      for (f=0;f<2;f++)
70        *p_z  += *p_a++ * *p_b++ ;
71
72      END_PROFILING;
73
74      pin_down(&Z) ;
75
76      return(0)  ;
77    }
```

B.2 Complex_multiply

```
0   /*
1    * benchmark program:   complex_multiply.c
2    *
3    * benchmark suite:     DSP-kernel
4    *
5    * description:         complex_multiply - filter benchmarking
6    *
7    * reference code:      target assembly
8    *
9    * f. verification:     simulator based
10   *
11   *  organization:        Aachen University of Technology - IS2
12   *                       DSP Tools Group
13   *                       phone:  +49(241)807887
14   *                       fax:    +49(241)8888195
15   *                       e-mail: zivojnov@ert.rwth-aachen.de
16   *
17   * author:              Juan Martinez Velarde
18   *
19   * history:             9-5-94 creation (Martinez Velarde)
20   *
21   *                      $Author: schraut $
22   *                      $Revision: 1.2 $
23   *                      $Date: 1995/02/01 11:54:14 $
24   */
25
26  #define STORAGE_CLASS static
27  #define TYPE  int
28
29  void
30  pin_down(TYPE *ar, TYPE *ai, TYPE *br, TYPE *bi, TYPE *cr, TYPE *ci)
31  {
32    *ar = 2 ;
33    *ai = 1 ;
34    *br = 2 ;
35    *bi = 5 ;
36  }
37
38  void
39  main()
40  {
41    STORAGE_CLASS TYPE ar, ai ;
42    STORAGE_CLASS TYPE br, bi ;
43    STORAGE_CLASS TYPE cr, ci ;
44
45    pin_down(&ar, &ai, &br, &bi, &cr, &ci) ;
46
47    START_PROFILING;
48
49    cr  = ar*br - ai*bi ;
```

```
50      ci   = ar*bi + ai*br ;
51
52      END_PROFILING;
53
54      pin_down(&ar, &ai, &br, &bi, &cr, &ci) ;
55
56    }
57
```

B.3 Real_update

```
0   /*
1    * benchmark program:   real_update.c
2    *
3    * benchmark suite:     DSP-kernel
4    *
5    * description:         real_update - filter benchmarking
6    *
7    * This program performs a real update of the form D = C + A*B,
8    * where A, B, C and D are real numbers .
9     *
10   * reference code:      target assembly
11   *
12   * f. verification:     simulator based
13   *
14   *  organization:        Aachen University of Technology - IS2
15   *                       DSP Tools Group
16   *                       phone:  +49(241)807887
17   *                       fax:    +49(241)8888195
18   *                       e-mail: zivojnov@ert.rwth-aachen.de
19   *
20   * author:              Juan Martinez Velarde
21   *
22   * history:             11-05-94 creation (Martinez Velarde)
23   *
24   *                      $Author: schraut $
25   *                      $Date: 1995/01/30 10:06:02 $
26   *                      $Revision: 1.3 $
27   */
28
29  #define STORAGE_CLASS register
30  #define TYPE  int
31
32  void
33  pin_down(TYPE *p)
34  {
35
36    *p = 0 ;
37
38  }
39
```

```
40
41  TYPE
42  main()
43  {
44    static TYPE A = 10 ;
45    static TYPE B = 2 ;
46    static TYPE C = 1 ;
47    static TYPE D = 0 ;
48
49    STORAGE_CLASS TYPE *p_a = &A ;
50    STORAGE_CLASS TYPE *p_b = &B ;
51    STORAGE_CLASS TYPE *p_c = &C ;
52    STORAGE_CLASS TYPE *p_d = &D ;
53
54    pin_down(&D) ;
55
56    START_PROFILING;
57
58    *p_d  = *p_c + *p_a * *p_b ;
59
60    END_PROFILING;
61
62    pin_down(&D) ;
63
64    return(0)  ;
65  }
```

B.4 Biquad_one_section

```
0   /*
1    *  benchmark program   : biquad_one_section.c
2    *
3    *  benchmark suite     : DSP-kernel
4    *
5    *  description         : benchmarking of an one iir biquad
6    *
7    *
8    *      The equations of the filter are:
9    *       w(n) =    x(n) - a1*w(n-1) - a2*w(n-2)
10   *       y(n) = b0*w(n) + b1*w(n-1) + b2*w(n-2)
11   *
12   *                              w(n)
13   *    x (n)------(-)----------->-|->----b0---(+)-------> y(n)
14   *                A              |             A
15   *                |            |1/z|           |
16   *                |              | w(n-1)      |
17   *                |              v             |
18   *                |-<--a1-----<-|->----b1-->-|
19   *                |              |             |
20   *                |            |1/z|           |
21   *                |              | w(n-2)      |
```

```
22    *                |              v              |
23    *                |-<--a2-----<--->----b2-->-|
24    *
25    *     The values w(n-1) and w(n-2) are stored in w1 and w2
26    *
27    *
28    *  reference code     :
29    *
30    *  func. verification : from separate computation
31    *
32    *  organization       : Aachen University of Technology - IS2
33    *                     : DSP Tools Group
34    *                     : phone    : +49(241)807887
35    *                     : fax      : +49(241)8888195
36    *                     : e-mail   : zivojnov@ert.rwth-aachen.de
37    *
38    *  author             : Juan Martinez Velarde
39    *
40    *  history            : creation 19-3-1994
41    *
42    *                     $Author: schraut $
43    *                     $Date: 1995/01/30 07:33:38 $
44    *                     $Revision: 1.2 $
45    */
46
47
48    #define STORAGE_CLASS register
49    #define TYPE int
50
51    TYPE
52    pin_down(TYPE x)
53    {
54      return ((TYPE) 7) ;
55    }
56
57
58    TYPE main()
59    {
60
61      STORAGE_CLASS TYPE y, w ;
62
63      static TYPE x = 7, w1= 7 , w2 = 7 ;
64      static TYPE b0 = 7, b1 = 7 , b2 = 7  ;
65      static TYPE a1 = 7, a2 = 7 ;
66
67      START_PROFILING;
68
69      w  = x - a1 * w1 ;
70      w -= a2 * w2 ;
71      y  = b0 * w ;
72      y += b1 * w1 ;
73      y += b2 * w2 ;
```

```
74
75      w2 = w1 ;
76      w1 = w  ;
77
78      END_PROFILING;
79
80      x  = pin_down(x) ;
81      w1 = pin_down(w1) ;
82      w2 = pin_down(w2) ;
83
84      return((TYPE) y) ;
85    }
```

B.5 Matrix1

```
 0  /*
 1   * benchmark program:   matrix1.c
 2   *
 3   * benchmark suite:     DSP-kernel
 4   *
 5   * description:         generic matrix - multiply benchmarking
 6   *
 7   * This program performs a matrix multiplication of the form C=AB,
 8   * where A and B are two dimensional matrices of arbitrary dimension.
 9   * The only restriction os that the inner dimension of the arrays must
10   * be greater than 1.
11   *
12   *          A[X x Y] * B[Y x Z] = C[X x Z]
13   *
14   *                   |a11     a12     ..      a1y|
15   *                   |a21     a22     ..      a2y|
16   * matrix A[X x Y]= |..      ..      ..      .. |
17   *                   |a(x-1)1 a(x-1)2 ..  a(x-1)y|
18   *                   |ax1     ax2     ..      axy|
19   *
20   *
21   *                   |b11     b12     ..      b1z|
22   *                   |b21     b22     ..      b2z|
23   * matrix B[Y x Z]= |..      ..      ..      .. |
24   *                   |b(y-1)1 b(y-1)2 .. b(y-1)z|
25   *                   |by1     by2     ..      byz|
26   *
27   *                   |c11     c12     ..      c1z|
28   *                   |c21     c22     ..      c2z|
29   * matrix C[X x Z]= |..      ..      ..      .. |
30   *                   |c(x-1)1 c(x-1)2 .. c(x-1)z|
31   *                   |cx1     cx2     ..      cxz|
32   *
33   * matrix elements are stored as
34   *
35   * A[X x Y] = { a11, a12, .. , a1y,
```

```
36    *              a21, a22, .. , a2y, ...,
37    *              ax1, ax2, .. , axy}
38    *
39    * B[Y x Z] = { b11, b21, .., b(y-1)1, by1, b12, b22, .. , b(y-1)z, byz}
40    *
41    * C[X x Z] = { c11, c21, .. , c(x-1)1, cx1, c12, c22, .. ,c(x-1)z, cxz }
42    *
43    *
44    * reference code:
45    *
46    * f. verification:
47    *
48    * organization:          Aachen University of Technology - IS2
49    *                        DSP Tools Group
50    *                        phone:  +49(241)807887
51    *                        fax:    +49(241)8888195
52    *                        e-mail: zivojnov@ert.rwth-aachen.de
53    *
54    * author:               Juan Martinez Velarde
55    *
56    * history:              3-4-94 creation (Martinez Velarde)
57    *                       5-4-94 profiling (Martinez Velarde)
58    *
59    *                       $Author: schraut $
60    *                       $Date: 1995/03/24 08:58:48 $
61    *                       $Revision: 1.4 $
62    */
63
64   #define STORAGE_CLASS  register
65   #define TYPE           int
66
67   #define X 10 /* first dimension of array A */
68   #define Y 10 /* second dimension of array A, first dimension of array B */
69   #define Z 10 /* second dimension of array B */
70
71   TYPE
72   pin_down(TYPE A[], TYPE B[], TYPE C[])
73   {
74     int i ;
75
76     for (i = 0 ; i < X*Y; i++)
77         A[i] = 1 ;
78
79     for (i = 0 ; i < Y*Z ; i++)
80         B[i] = 1 ;
81
82     for (i = 0 ; i < X*Z ; i++)
83         C[i] = 0 ;
84
85     return((TYPE)0) ;
86
87   }
```

```
88
89   TYPE
90   main()
91   {
92     static  TYPE A[X*Y] ;
93     static  TYPE B[Y*Z] ;
94     static  TYPE C[X*Z] ;
95
96     STORAGE_CLASS TYPE *p_a = &A[0] ;
97     STORAGE_CLASS TYPE *p_b = &B[0] ;
98     STORAGE_CLASS TYPE *p_c = &C[0] ;
99
100    STORAGE_CLASS TYPE f,i,k ;
101
102    pin_down(&A[0], &B[0], &C[0]) ;
103
104    START_PROFILING;
105
106    for (k = 0 ; k < Z ; k++)
107      {
108        p_a = &A[0] ;                    /* point to the beginning of array A */
109
110        for (i = 0 ; i < X; i++)
111          {
112            p_b = &B[k*Y] ;              /* take next column */
113
114            *p_c =  0 ;
115
116            for (f = 0 ; f < Y; f++) /* do multiply */
117              *p_c += *p_a++ * *p_b++ ;
118
119            *p_c++  ;
120
121          }
122      }
123
124    END_PROFILING;
125
126    pin_down(&A[0], &B[0], &C[0]) ;
127
128    return(0)  ;
129  }
```

B.6 Complex_Update

```
0   /*
1    * benchmark program:   complex_update.c
2    *
3    * benchmark suite:     DSP-kernel
4    *
5    * description:         complex_update - filter benchmarking
```

```
 6    *
 7    * This program performs a complex update of the form D = C + A*B,
 8    * where A, B, C and D are complex numbers .
 9    *
10    *          A = Ar + j Ai
11    *          B = Br + j Bi
12    *          C = Cr + j Ci
13    *          D = C + A*B =   Dr  +  j Di
14    *                      =>  Dr = Cr + Ar*Br - Ai*Bi
15    *                      =>  Di = Ci + Ar*Bi + Ai*Br
16    *
17    * reference code:       target assembly
18    *
19    * f. verification:      simulator based
20    *
21    *  organization:         Aachen University of Technology - IS2
22    *                        DSP Tools Group
23    *                        phone:  +49(241)807887
24    *                        fax:    +49(241)8888195
25    *                        e-mail: zivojnov@ert.rwth-aachen.de
26    *
27    * author:               Juan Martinez Velarde
28    *
29    * history:              11-5-94 creation (Martinez Velarde)
30    *
31    *                       $Author: schraut $
32    *                       $Date: 1995/01/26 10:27:31 $
33    *                       $Revision: 1.2 $
34    */
35
36   #define STORAGE_CLASS register
37   #define TYPE  int
38
39   void
40   pin_down(TYPE *p)
41   {
42
43     *p++ = 0 ;
44     *p   = 0 ;
45
46   }
47
48
49   TYPE
50   main()
51   {
52     static TYPE A[2] = { 2,1 } ;
53     static TYPE B[2] = { 2,5 } ;
54     static TYPE C[2] = { 3,4 } ;
55     static TYPE D[2] = { 0,0 } ;
56
57     STORAGE_CLASS TYPE *p_a = &A[0] ;
```

```
58      STORAGE_CLASS TYPE *p_b = &B[0] ;
59      STORAGE_CLASS TYPE *p_c = &C[0] ;
60      STORAGE_CLASS TYPE *p_d = &D[0] ;
61
62      pin_down(&D[0]) ;
63
64      START_PROFILING;
65
66      *p_d    = *p_c++ + *p_a++ * *p_b++ ;
67      *p_d++ -=          *p_a   * *p_b-- ;
68
69      *p_d    = *p_c   + *p_a-- * *p_b++ ;
70      *p_d   +=          *p_a   * *p_b ;
71
72      END_PROFILING;
73
74      pin_down(&D[0]) ;
75
76      return(0)  ;
77  }
```

B.7 N_Complex_Updates

```
0   /*
1    * benchmark program:   n_complex_updates.c
2    *
3    * benchmark suite:     DSP-kernel
4    *
5    * description:         n complex updates - filter benchmarking
6    *
7    * This program performs n complex updates of the form
8    *           D(i) = C(i) + A(i)*B(i),
9    * where A(i), B(i), C(i) and D(i) are complex numbers,
10   * and i = 1,...,N
11   *
12   *          A(i) = Ar(i) + j Ai(i)
13   *          B(i) = Br(i) + j Bi(i)
14   *          C(i) = Cr(i) + j Ci(i)
15   *          D(i) = C(i) + A(i)*B(i) =   Dr(i)  +  j Di(i)
16   *                       =>  Dr(i) = Cr(i) + Ar(i)*Br(i) - Ai(i)*Bi(i)
17   *                       =>  Di(i) = Ci(i) + Ar(i)*Bi(i) + Ai(i)*Br(i)
18   *
19   * reference code:      target assembly
20   *
21   * f. verification:     simulator based
22   *
23   *  organization:        Aachen University of Technology - IS2
24   *                       DSP Tools Group
25   *                       phone:  +49(241)807887
26   *                       fax:    +49(241)8888195
27   *                       e-mail: zivojnov@ert.rwth-aachen.de
```

```
28    *
29    * author:                Juan Martinez Velarde
30    *
31    * history:               13-5-94 creation (Martinez Velarde)
32    *
33    *                        $Author: schraut $
34    *                        $Date: 1995/01/26 11:00:36 $
35    *                        $Revision: 1.2 $
36    */
37
38   #define STORAGE_CLASS register
39   #define TYPE  int
40   #define N              16
41
42   void
43   pin_down(TYPE *pa, TYPE *pb, TYPE *pc, TYPE *pd)
44   {
45     STORAGE_CLASS int i ;
46
47     for (i=0 ; i < N ; i++)
48       {
49         *pa++ = 2 ;
50         *pa++ = 1 ;
51         *pb++ = 2 ;
52         *pb++ = 5 ;
53         *pc++ = 3 ;
54         *pc++ = 4 ;
55         *pd++ = 0 ;
56         *pd++ = 0 ;
57       }
58   }
59
60
61   TYPE
62   main()
63   {
64     static TYPE A[2*N], B[2*N], C[2*N], D[2*N] ;
65
66     STORAGE_CLASS TYPE *p_a = &A[0], *p_b = &B[0] ;
67     STORAGE_CLASS TYPE *p_c = &C[0], *p_d = &D[0] ;
68     STORAGE_CLASS TYPE i ;
69
70     pin_down(&A[0], &B[0], &C[0], &D[0]) ;
71
72     START_PROFILING;
73
74     for (i = 0 ; i < N ; i++, p_a++)
75       {
76         *p_d    = *p_c++ + *p_a++ * *p_b++ ;
77         *p_d++ -=          *p_a   * *p_b-- ;
78
79         *p_d    = *p_c++ + *p_a-- * *p_b++ ;
```

```
80          *p_d++ +=           *p_a++ * *p_b++ ;
81        }
82
83      END_PROFILING;
84
85      pin_down(&A[0], &B[0], &C[0], &D[0]) ;
86
87      return(0)   ;
88    }
```

B.8 Convolution

```
0   /*
1    * benchmark program:    convolution.c
2    *
3    * benchmark suite:      DSP-kernel
4    *
5    * description:          convolution - filter benchmarking
6    *
7    * reference code:       target assembly
8    *
9    * f. verification:      none
10   *
11   *  organization:         Aachen University of Technology - IS2
12   *                        DSP Tools Group
13   *                        phone:  +49(241)807887
14   *                        fax:    +49(241)8888195
15   *                        e-mail: zivojnov@ert.rwth-aachen.de
16   *
17   * author:               Vojin Zivojnovic
18   *
19   * history:              14-1-94 creation (Vojin Zivojnovic)
20   *                       18-3-94 asm labels included (Martinez Velarde)
21   *
22   *                       $Author: schraut $
23   *                       $Date: 1995/01/30 07:24:54 $
24   *                       $Revision: 1.2 $
25   */
26
27  #define STORAGE_CLASS register
28  #define TYPE  int
29  #define LENGTH 16
30
31  void pin_down(TYPE * px, TYPE * ph)
32  {
33          STORAGE_CLASS TYPE    i;
34
35          for (i = 0; i < LENGTH; ++i) {
36                  *px++ = 1;
37                  *ph++ = 1;
38          }
```

```
39
40  }
41
42
43  TYPE main()
44  {
45
46          static TYPE     x[LENGTH];
47          static TYPE     h[LENGTH];
48
49          STORAGE_CLASS TYPE y;
50          STORAGE_CLASS TYPE i;
51          STORAGE_CLASS TYPE *px = x;
52          STORAGE_CLASS TYPE *ph = &h[LENGTH - 1];
53
54
55          pin_down(&x[0], &h[0]);
56
57          START_PROFILING;
58
59          y = 0;
60
61          for (i = 0; i < LENGTH; ++i)
62                  y += *px++ * *ph--;
63
64          END_PROFILING;
65
66
67          return ((TYPE) y);
68
69  }
```

B.9 Fir

```
0   /*
1    * benchmark program:   fir.c
2    *
3    * benchmark suite:     DSP-kernel
4    *
5    * description:         fir - filter benchmarking
6    *
7    * reference code:      target assembly
8    *
9    * f. verification:     simulator
10   *
11   *  organization:        Aachen University of Technology - IS2
12   *                       DSP Tools Group
13   *                       phone:  +49(241)807887
14   *                       fax:    +49(241)8888195
15   *                       e-mail: zivojnov@ert.rwth-aachen.de
16   *
```

```
17    * author:                Juan Martinez Velarde
18    *
19    * history:                12-4-94 creation (Martinez Velarde)
20    *                         22-1-95 START and END PROFILING Macros included
21    *                                 pin_down routine -> 3 arguments (Schraut)
22    *
23    *                         $Date: 1995/01/25 17:14:52 $
24    *                         $Author: schraut $
25    *                         $Revision: 1.2 $
26    */
27
28   #define STORAGE_CLASS register
29   #define TYPE  int
30   #define LENGTH 16
31
32   void
33   pin_down(TYPE * px, TYPE * ph, TYPE y)
34   {
35     STORAGE_CLASS TYPE    i;
36
37     for (i = 1; i <= LENGTH; i++)
38       {
39         *px++ = i;
40         *ph++ = i;
41       }
42
43   }
44
45   TYPE
46   main()
47   {
48     static TYPE  x[LENGTH];
49     static TYPE  h[LENGTH];
50
51     static TYPE  x0 = 100;
52
53     STORAGE_CLASS TYPE i ;
54     STORAGE_CLASS TYPE *px, *px2 ;
55     STORAGE_CLASS TYPE *ph ;
56     STORAGE_CLASS TYPE y;
57
58     pin_down(x, h, y);
59
60     ph  = &h[LENGTH-1] ;
61     px  = &x[LENGTH-1]  ;
62     px2 = &x[LENGTH-2]  ;
63
64     START_PROFILING ;
65
66     y = 0;
67
68     for (i = 0; i < LENGTH - 1; i++)
```

```
69        {
70          y += *ph-- * *px ;
71          *px-- = *px2-- ;
72        }
73
74      y += *ph * *px ;
75      *px = x0 ;
76
77      END_PROFILING ;
78
79      pin_down(x, h, y);
80
81      return ((TYPE) y);
82    }
```

B.10 Fir2dim

```
0   /*
1    * benchmark program:   fir2dim.c
2    *
3    * benchmark suite:     DSP-kernel
4    *
5    * description:         fir2dim - filter benchmarking
6    *
7    * The image is an array IMAGEDIM * IMAGEDIM pixels. To provide
8    * conditions for the FIR filtering, the image is surrounded by a
9    * set of zeros such that the image is actually stored as a
10   * ARRAYDIM * ARRAYDIM = (IMAGEDIM + 2) * (IMAGEDIM + 2) array
11   *
12   *          <--ARRAYDIM-->
13   *         |0 0 0 .... 0 0| A
14   *         |0 x x .... x 0| |
15   *         |0 x x .... x 0| ARRAY_
16   *         |0 image area 0| DIM
17   *         |0 x x .... x 0| |
18   *         |0 x x .... x 0| |
19   *         |0 0 0 .... 0 0| V
20   *
21   * The image (with boundary) is stored in row major storage. The
22   * first element is array(1,1) followed by array(1,2). The last
23   * element of the first row is array(1,514) following by the
24   * beginning of the next column array(2,1).
25   *
26   * The two dimensional FIR uses a 3x3 coefficient mask:
27   *
28   *         |c11 c12 c13|
29   *         |c21 c22 c23|
30   *         |c31 c32 c33|
31   *
32   * The output image is of dimension IMAGEDIM * IMAGEDIM.
33   *
```

```
34    * reference code:       target assembly
35    *
36    * f. verification:      simulator based
37    *
38    *  organization:         Aachen University of Technology - IS2
39    *                        DSP Tools Group
40    *                        phone:  +49(241)807887
41    *                        fax:    +49(241)8888195
42    *                        e-mail: zivojnov@ert.rwth-aachen.de
43    *
44    * author:               Juan Martinez Velarde
45    *
46    * history:              15-5-94 creation (Martinez Velarde)
47    *
48    *                       $Author: schraut $
49    *                       $Date: 1995/01/30 07:17:23 $
50    *                       $Revision: 1.2 $
51    */
52
53   #define STORAGE_CLASS register
54   #define TYPE          int
55   #define IMAGEDIM      4
56   #define ARRAYDIM      (IMAGEDIM + 2)
57   #define COEFFICIENTS  3
58
59   void
60   pin_down(TYPE *pimage, TYPE *parray, TYPE *pcoeff, TYPE *poutput)
61   {
62     STORAGE_CLASS TYPE    i,f;
63
64     for (i = 0 ; i < IMAGEDIM ; i++)
65       {
66         for (f = 0 ; f < IMAGEDIM ; f++)
67           *pimage++ = 1 ;
68       }
69
70     pimage = pimage - IMAGEDIM*IMAGEDIM  ;
71
72     for (i = 0; i < COEFFICIENTS*COEFFICIENTS; i++)
73       *pcoeff++ = 1;
74
75     for (i = 0 ; i < ARRAYDIM ; i++)
76       *parray++ = 0 ;
77
78
79     for (f = 0 ; f < IMAGEDIM; f++)
80       {
81         *parray++ = 0 ;
82         for (i = 0 ; i < IMAGEDIM ; i++)
83           *parray++ = *pimage++ ;
84         *parray++ = 0 ;
85       }
```

```
86
87      for (i = 0 ; i < ARRAYDIM ; i++)
88        *parray++ = 0 ;
89
90      for (i = 0 ; i < IMAGEDIM * IMAGEDIM; i++)
91            *poutput++ = 0 ;
92    }
93
94
95    void main()
96    {
97
98      static TYPE  coefficients[COEFFICIENTS*COEFFICIENTS] ;
99      static TYPE  image[IMAGEDIM*IMAGEDIM]   ;
100     static TYPE  array[ARRAYDIM*ARRAYDIM]   ;
101     static TYPE  output[IMAGEDIM*IMAGEDIM] ;
102
103     STORAGE_CLASS TYPE *pimage  = &image[0]        ;
104     STORAGE_CLASS TYPE *parray  = &array[0], *parray2, *parray3 ;
105     STORAGE_CLASS TYPE *pcoeff  = &coefficients[0] ;
106     STORAGE_CLASS TYPE *poutput = &output[0]       ;
107     STORAGE_CLASS TYPE k, f, i;
108
109     pin_down(&image[0], &array[0], &coefficients[0], &output[0]);
110
111     pimage  = &image[0]        ;
112     parray  = &array[0]        ;
113     pcoeff  = &coefficients[0] ;
114     poutput = &output[0]       ;
115
116     START_PROFILING;
117     for (k = 0 ; k < IMAGEDIM ; k++)
118       {
119
120         for (f = 0 ; f < IMAGEDIM ; f++)
121           {
122             pcoeff = &coefficients[0] ;
123             parray = &array[k*ARRAYDIM + f] ;
124             parray2 = parray + ARRAYDIM ;
125             parray3 = parray + ARRAYDIM + ARRAYDIM ;
126
127             *poutput = 0 ;
128
129             for (i = 0 ; i < 3 ; i++)
130               *poutput += *pcoeff++ * *parray++ ;
131
132             for (i = 0 ; i < 3 ; i++)
133               *poutput += *pcoeff++ * *parray2++ ;
134
135             for (i = 0 ; i < 3 ; i++)
136               *poutput += *pcoeff++ * *parray3++ ;
137
```

```
138             poutput++ ;
139           }
140       }
141     END_PROFILING;
142
143     pin_down(&image[0], &array[0], &coefficients[0], &output[0]);
144
145   }
146
```

B.11 Lms

```
0   /*
1    * benchmark program:   lms.c
2    *
3    * benchmark suite:     DSP-kernel
4    *
5    * description:         lms - filter benchmarking
6    *
7    *
8    *                x(n-0)  ---- x(n-1)  ------ x(n-i) -----  x(n-N+1)
9    *    x(n) --->---------| 1/z|-------| 1/z|-.....-| 1/z|----
10   *                  |   -----    |   ------   |   ------   |
11   *                  h0           h1           hi           hN-1
12   *                    \--------  |____   _____|  _________/
13   *                             \______(+)_______/
14   *                                    |
15   *                                    |---------> y(n)
16   *                                   v
17   *          d(n) --------------------->(+)--------> e(n)
18   *                                    -
19   *
20   *
21   * Notation and symbols:
22   * x(n)  - Input sample at time n.
23   * d(n)  - Desired signal at time n.
24   * y(n)  - FIR filter output at time n.
25   * H(n)  - Filter coefficient vector at time n.  H={h0,h1,..,hN-1}
26   * X(n)  - Filter state variable vector at time N.
27   *         X={x(0),x(n-1),...,x(n-N+1)}
28   * delta - Adaptation gain.
29   * N     - Number of coefficient taps in the filter.
30   *
31   * PROCESSING:
32   *
33   *     True LMS Algorithm
34   *     ------------------
35   *     Get input sample
36   *     Save input sample
37   *     Do FIR
38   *     Shift vector X
```

```
39    *     Get d(n), calculate e(n)
40    *     Update coefficients
41    *     Output y(n)
42    *
43    *
44    * System equations:
45    *
46    * e(n)   = d(n) - H(n)  X(n)        (FIR filter and error)
47    * H(n+1)= H(n) + delta X(n) e(n)   (Coefficient update)
48    *
49    * References:
50    *  "Adaptive Digital Filters and Signal Analysis", Maurice G. Bellanger
51    *     Marcel Dekker, Inc. New York and Basel
52    *
53    *  "The DLMS Algorithm Suitable for the Pipelined Realization of Adaptive
54    *      Filters", Proc. IEEE ASSP Workshop, Academia Sinica, Beijing, 1986
55    *
56    *
57    *
58    * reference code:       target assembly
59    *
60    * f. verification:      none
61    *
62    *  organization:         Aachen University of Technology - IS2
63    *                        DSP Tools Group
64    *                        phone:  +49(241)807887
65    *                        fax:    +49(241)8888195
66    *                        e-mail: zivojnov@ert.rwth-aachen.de
67    *
68    * author:               Juan Martinez Velarde
69    *
70    * history:              29-3-94 creation (Martinez Velarde)
71    *                       03-4-94 first revision, optimized (Martinez Velarde)
72    *
73    *                       $Author: schraut $
74    *                       $Date: 1995/01/30 07:48:50 $
75    *                       $Revision: 1.2 $
76    */
77
78    #define STORAGE_CLASS  register
79    #define TYPE           int
80
81
82    #define N 16  /* number of coefficient taps */
83
84
85    void
86    pin_down(TYPE *d, TYPE *x, TYPE *delta, TYPE *p_H, TYPE *p_X)
87    {
88      int f ;
89
90      *d = 7 ;
```

```
91      *x = 8 ;
92      *delta = 1 ;
93
94      for (f = 0 ; f < N ; f++)
95        {
96          *p_H++ = 1 ;
97          *p_X++ = 1 ;
98        }
99
100   }
101
102   TYPE
103   main()
104   {
105     static TYPE H[N] ; /* Filter Coefficient Vector */
106     static TYPE X[N] ; /* Filter State Variable Vector */
107
108     TYPE delta  ;  /* Adaption Gain */
109     TYPE d ;       /* Desired signal */
110     TYPE x ;       /* Input Sample */
111     TYPE y ;       /* FIR LMS Filter Output */
112     STORAGE_CLASS TYPE error ;   /* FIR error */
113
114     STORAGE_CLASS TYPE f ;
115
116     STORAGE_CLASS TYPE *p_H, *p_X, *p_X2 ;
117
118     pin_down(&d,&x,&delta,&H[0],&X[0]) ;
119
120     p_H = &H[N-1] ;
121     p_X = &X[N-1] ;
122     p_X2= &X[N-2] ;
123
124     START_PROFILING;
125
126     y = 0 ;
127
128     /* FIR filtering and State Variable Update */
129
130     for (f = 1 ; f < N  ; f++)
131       y += *p_H-- * (*p_X-- = *p_X2--) ;
132
133     /* last convolution tap, get input sample */
134
135     y += *p_H * (*p_X = x)  ;
136
137     /*
138      *  error as the weighted difference
139      *  between desired and calculated signal
140      *
141      */
142
```

```
143     error = (d - y) * delta ;
144
145     for (f =  0; f < N ; f++)
146       *p_H++ += error * *p_X++ ;   /* update the coefficients */
147
148     END_PROFILING;
149
150     pin_down(&d,&x,&y,&H[0],&X[0]) ;
151
152     return(0)  ;
153
154   }
155
```

B.12 Mat1x3

```
0   /*
1    * benchmark program:   mat1x3.c
2    *
3    * benchmark suite:     DSP-kernel
4    *
5    * description:         1x3 matrix - multiply benchmarking
6    *
7    *               |h11 h12 h13|   |x1|   |y1|   | h11*x1+h12*x2+h31*x3 |
8    *               |h21 h22 h23| * |x2| = |y2| = | h21*x1+h22*x2+h23*x3 |
9    *               |h31 h32 h33|   |x3|   |y3|   | h31*x1+h32*x2+h33*x3 |
10   *
11   * Element are to store in following order:
12   *
13   * matrix h[9]={h11,h12,h13, h21,h22,h23, h31,h32,h33}
14   * vector x[3]={x1,x2,x3}
15   * vector y[3]={y1,y1,y3}
16   *
17   * reference code:       none
18   *
19   * f. verification:
20   *
21   * organization:         Aachen University of Technology - IS2
22   *                       DSP Tools Group
23   *                       phone:  +49(241)807887
24   *                       fax:    +49(241)8888195
25   *                       e-mail: zivojnov@ert.rwth-aachen.de
26   *
27   * author:               Juan Martinez Velarde
28   *
29   * history:              31-3-94 creation (Martinez Velarde)
30   *
31   *                       $Author: schraut $
32   *                       $Date: 1995/01/26 11:17:25 $
33   *                       $Revision: 1.2 $
34   */
```

```
35
36   #define STORAGE_CLASS  register
37   #define TYPE           int
38
39   TYPE
40   main()
41   {
42
43     static TYPE h[9]={1,2,3,1,2,3,3,2,1} ;
44     static TYPE x[3]={1,1,1} ;
45     static TYPE y[3]={0,0,0} ;
46
47     STORAGE_CLASS TYPE *p_x = &x[0] ;
48     STORAGE_CLASS TYPE *p_h = &h[0] ;
49     STORAGE_CLASS TYPE *p_y = &y[0] ;
50
51     STORAGE_CLASS TYPE f,i ;
52
53     START_PROFILING;
54
55     for (i = 0 ; i < 3; i++)
56       {
57         /* p_x points to the beginning of the input vector */
58
59         p_x = &x[0]  ;
60
61          /* do matrix multiply */
62
63         for (f = 0 ; f < 3; f++)
64             *p_y += *p_h++ * *p_x++ ;
65
66         /* next element */
67
68         p_y++ ;
69       }
70
71     END_PROFILING;
72
73     return(0)  ;
74
75   }
```

B.13 N_Real_Updates

```
0    /*
1     * benchmark program:   n_real_updates.c
2     *
3     * benchmark suite:     DSP-kernel
4     *
5     * description:         n_real_updates - filter benchmarking
6     *
```

```
 7    * This program performs n real updates of the form
 8    *           D(i) = C(i) + A(i)*B(i),
 9    * where A(i), B(i), C(i) and D(i) are real numbers,
10    * and i = 1,...,N
11    *
12    * reference code:       target assembly
13    *
14    * f. verification:      simulator based
15    *
16    *  organization:         Aachen University of Technology - IS2
17    *                        DSP Tools Group
18    *                        phone:  +49(241)807887
19    *                        fax:    +49(241)8888195
20    *                        e-mail: zivojnov@ert.rwth-aachen.de
21    *
22    * author:               Juan Martinez Velarde
23    *
24    * history:              25-5-94 creation (Martinez Velarde)
25    *
26    *                       $Author: schraut $
27    *                       $Date: 1995/01/26 09:44:22 $
28    *                       $Revision: 1.2 $
29    */
30
31   #define STORAGE_CLASS register
32   #define TYPE          int
33   #define N             16
34
35   void
36   pin_down(TYPE *pa, TYPE *pb, TYPE *pc, TYPE *pd)
37   {
38     STORAGE_CLASS int i ;
39
40     for (i=0 ; i < N ; i++)
41       {
42         *pa++ = 10 ;
43         *pb++ = 2 ;
44         *pc++ = 10 ;
45         *pd++ = 0 ;
46       }
47   }
48
49   TYPE main()
50   {
51     static TYPE A[N], B[N], C[N], D[N] ;
52
53     STORAGE_CLASS TYPE *p_a = &A[0], *p_b = &B[0] ;
54     STORAGE_CLASS TYPE *p_c = &C[0], *p_d = &D[0] ;
55     STORAGE_CLASS TYPE i ;
56
57     pin_down(&A[0], &B[0], &C[0], &D[0]) ;
58
```

```
59      START_PROFILING;
60
61      for (i = 0 ; i < N ; i++)
62        *p_d++  = *p_c++ + *p_a++ * *p_b++ ;
63
64      END_PROFILING;
65
66      pin_down(&A[0], &B[0], &C[0], &D[0]) ;
67      return(0)   ;
68    }
```