Geração e Vetorização de Instruções de Multiplicação e Acumulação para Processadores DSP SIMD

Natália Viana Fargasch

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Geração e Vetorização de Instruções de Multiplicação e Acumulação para Processadores DSP SIMD

Natália Viana Fargasch[1]

18 de Dezembro de 2000

Banca Examinadora:

- Prof. Dr. Guido Costa Souza de Araújo (Orientador)
- Profa. Dra. Mariza Andrade Silva Bigonha
- Prof. Dr. Luiz Eduardo Buzato
- Prof. Dr. Paulo César Centoducatte (Suplente)

[1]**Financiamento para este projeto concedido pela PRPG-CAPES e pelo CNPq**

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Fargasch, Natália Viana

F224g Geração e vetorização de instruções de multiplicação e acumulação para processadores DSP SIMD / Natália Viana Fargasch -- Campinas, [S.P. :s.n.], 2001.

Orientador : Guido Costa Souza de Araújo

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

1. Compiladores (Programa de computador). 2. Linguagem de programação (Computadores). I. Araújo, Guido Costa Souza de. II. Universidade Estadual de Campinas. Instituto de Computação . III. Título.

Geração e Vetorização de Instruções de Multiplicaçã e Acumulação para Processadores DSP SIMD

Este exemplar corresponde à redação final Dissertação devidamente corrigida e defendi por Natália Viana Fargasch e aprovada p Banca Examinadora.

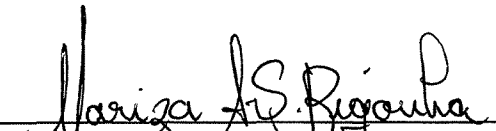
Campinas, 05 de Fevereiro de 2001.

Prof. Dr. Guido Costa Souza de Araújo
(Orientador)

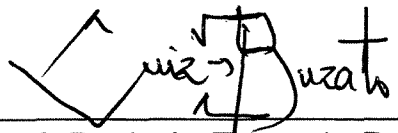
Dissertação apresentada ao Instituto de Co putação, UNICAMP, como requisito parcial pa a obtenção do título de Mestre em Ciência Computação.

TERMO DE APROVAÇÃO

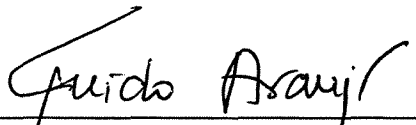
Tese defendida e aprovada em 07 de janeiro de 2001, pela Banca Examinadora composta pelos Professores Doutores:



Prof. Dr. Mariza Andrade Silva Bigonha
UFMG



Prof. Dr. Luiz Eduardo Buzato
IC – UNICAMP



Prof. Dr. Guido Costa Souza de Araújo
IC – UNICAMP

Resumo

Processadores que são projetados para executar aplicações específicas — em oposição a processadores de propósito geral — representam uma porcentagem cada vez maior do total de processadores vendidos anualmente. Esses processadores são utilizados em aparelhos eletrônicos como telefones celulares e câmeras digitais, dispositivos médicos de monitoração, modems, sistemas militares de radar, componentes eletrônicos de automóveis, *set-top boxes*, etc. As aplicações que são executadas por esses processadores tipicamente demandam um alto desempenho, combinado com reduzido tamanho de código e dissipação de energia.

Esta dissertação aborda um dos problemas presentes durante a geração de código para uma classe desses processadores, os *processadores de sinais digitais* (DSPs): como o compilador pode utilizar as instruções especializadas desses processadores a fim de aumentar a densidade e melhorar o desempenho do código gerado. É proposto um procedimento que permite a detecção/geração de instruções de multiplicação e acumulação (muito comuns nas aplicações desses processadores). É ainda apresentado um método que permite explorar a possibilidade de execução de código em paralelo por duas ou mais unidades funcionais quando essas são capazes de operar simultaneamente sobre diferentes dados.

Os métodos aqui apresentados permitem uma exploração bastante agressiva das instruções de multiplicação e acumulação, e se utilizam de algoritmos de análise de fluxo de dados e técnicas de reestruturação de laços. Não é conhecido nenhum trabalho que aborde esse problema da maneira como é apresentada neste.

Abstract

Application specific processors — as opposed to general purpose processors — account for an ever increasing percentage of the processors sold each year. These processors are widely used in electronic devices such as cellular phones and digital cameras, medical monitoring devices, modems, military radar systems, electronic components in vehicles and set-top boxes, to name a few. The applications that usually run on these processors demand high performance, reduced code size and low power consuption.

This thesis addresses one of the issues that arise when generating code for a class of these processors, the *digital signal processors* (DSPs): how the compiler can take advantage of their specialized instructions in order to reduce the size and improve performance of the code generated. A method is proposed that allows for the detection/generation of multiply and accumulate instructions (typically present in these processors' applications). Also presented in this work is a method that makes it possible to explore the possibility of running code in parallel on two or more functional units when these are capable of operating simultaneously on different data.

The methods herein presented allow for an aggressive harnessing of multiply and accumulate instructions; to accomplish this goal they rely on data flow analysis algorithms and on loop restructuring techniques. No other work is known of that addresses this problem the way it is dealt with in this thesis.

"Somehow I can't believe that there are any heights that can't be scaled by a man who knows the secrets of making dreams come true. This special secret, it seems to me, can be summarized in four C s. They are curiosity, confidence, courage, and constancy, and the greatest of all is confidence. When you believe in a thing, believe in it all the way, implicitly and unquestionably."

Walt Disney

Agradecimentos

À PRPG-CAPES e ao CNPq pelo suporte financeiro.

Ao meu professor e orientador Guido, pelo apoio e participação, pela experiência transmitida, pela oportunidade de trabalhar com um compilador de produção e pela confiança depositada, que muitas vezes superou a minha própria e certamente contribuiu para a concretização deste trabalho.

Aos professores e funcionários do Instituto de Computação da UNICAMP.

Ao pessoal da Conexant Systems Inc., pela excelente oportunidade de estágio e pela concessão do compilador e *benchmarks* que possibilitaram a realização deste trabalho.

Aos meus pais, Hans e Zélia, e às minhas irmãs, Gabriela e Mariana, por me apoiarem sempre e pela compreensão nos (vários) momentos em que estive ausente; à minha sobrinha Samantha, pelos momentos "terapêuticos" que passamos juntas, e que não têm preço.

Ao grande amigo Rodrigo Ferreira, pela amizade, pelo convívio sempre agradável e por tanto ter me ajudado, ao dividir comigo os seus conhecimentos.

Ao colega Marcelo Cintra, pelas inúmeras e tão preciosas "dicas", antes e durante o meu estágio, que muito me ajudaram.

Ao Flávio Miana, que, mesmo às voltas com os seus próprios compromissos e *deadlines*, se dispos a ler alguns capítulos desta dissertação; por suas sugestões e também por ter entrado na minha vida com um *timing* mais que perfeito.

A todos os meus amigos, em especial aos de Belo Horizonte, pelos divertidos momentos em que nos encontramos ou falamos ao telefone durante a minha estada em Campinas.

Natália Viana Fargasch

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

1.1 Visão Geral

Os processadores normalmente utilizados em computadores pessoais e estações de trabalho são chamados processadores de propósito geral, porque se destinam a executar um vasto conjunto de aplicações. Para que possam realizar diversos tipos de tarefas, esses processadores devem ser capazes de prover as funcionalidades necessárias à realização de todas elas. Isso acaba por gerar arquiteturas que ocupam uma área grande de silício, resultando em processadores caros e com um elevado consumo de energia.

Uma outra classe de processadores, conhecidos como processadores para aplicações específicas, vêm conquistando um espaço bastante significativo no mercado de produtos eletrônicos e em telecomunicações [17]. Esses processadores provêm somente a funcionalidade suficiente para executar de maneira eficiente um determinado conjunto de aplicações, e esse é um dos fatores responsáveis por algumas de suas principais características, tais como: baixo preço e reduzido tamanho e consumo de energia, que são bem menores quando comparados aos processadores de propósito geral.

Os processadores para aplicações específicas estão presentes em aparelhos eletrônicos como telefones celulares e câmeras digitais, dispositivos médicos de monitoração, modems, sistemas militares de radar, componentes eletrônicos em automóveis, *set-top boxes*, brinquedos, etc. Um subconjunto importante dessa classe de processadores é formado pelos *processadores de sinais digitais*, ou DSPs[1], que são arquiteturas destinadas à execução de operações matemáticas com sinais, ou seqüencias de amostras, representados em forma digital [16].

[1] Do inglês: ***D**igital **S**ignal **P**rocessor.*

As diferenças existentes entre essas duas classes de processadores, como será exposto mais adiante, não estão limitadas apenas à área ocupada pelo processador, ao consumo de energia, ao seu preço ou ao tipo de aplicativo a cuja execução se destinam. Essas diferenças, juntamente com algumas características particulares dos processadores para aplicações específicas, em especial dos DSPs, influenciam diretamente a organização da *datapath* e o conjunto de instruções dessas arquiteturas e, conseqüentemente, causam um impacto na maneira como programas executam nesses processadores e como estes são otimizados.

1.2 Motivação

O mercado de produtos baseados em DSPs tem sofrido um crescimento acelerado, especialmente devido à crescente demanda por telefones celulares digitais. Recentemente tem também aumentado a demanda por aparelhos portáteis de reprodução de músicas no formato MP3 (*MP3 players*). Foi estimado que o mercado desses dispositivos, dentro de cada um dos quais encontra-se um DSP, além de um micro-controlador, um conversor analógico-digital e memória suficiente para armazenar uma hora ou mais de música, chegará a atingir 15 milhões de unidades em 2003 [47].

Se for considerado o lucro total resultante da venda de *chips* de DSPs, o valor estimado para 2003 é de 13,6 bilhões de dólares [47]. É provável, no entanto, que esses números venham a se mostrar bastante conservativos, pois não consideram a introdução da telefonia celular de terceira geração (3G), a difusão da HDTV[2] e a incorporação de novas tecnologias aos aparelhos de telefones celulares, o que certamente irá contribuir para um aumento nas vendas de dispositivos contendo em seu interior um DSP.

Não somente o volume de venda dos aparelhos eletrônicos tem apresentado um rápido crescimento; o mercado desses produtos é também caracterizado por uma complexidade cada vez maior das suas aplicações. Existe ainda um outro fator que torna o mercado de aparelhos eletrônicos mais dinâmico: os grandes fabricantes desses produtos vivem em uma constante corrida para chegar ao mercado primeiro. Assim, os processadores embutidos devem ser capazes não somente de atender a uma grande demanda do mercado, mas de fazê-lo em um curto intervalo de tempo.

Há uma tendência atual dos projetos de sistemas de DSPs serem baseados em *software* que executa em um processador programável, no lugar de um *hardware* dedicado [38]. Entretanto, o desenvolvimento de *software* para DSPs representa um gargalo no processo de projeto desses sistemas, pois a programação destes em *assembly* ainda é predominante.

[2]Do inglês: ***H**igh **D**efinition **T**elevision.*

A razão pela qual isto ocorre é o fato de que os compiladores atuais para DSPs produzem código objeto grande e de baixo desempenho [58]. Assim sendo, existe uma crescente demanda por técnicas eficientes de compilação para DSPs que melhore a qualidade do código gerado.

1.3 Objetivo

Este trabalho tem como objetivo principal tornar mais eficientes os programas que executam em DSPs, especialmente quando esses possuíem duas ou mais unidades funcionais com capacidade de operação simultânea sobre diferentes partes do conjunto de dados de entrada. Em geral, a eficiência de um programa pode ser medida em função do seu tempo de execução (número de ciclos de máquina) e/ou da memória por ele ocupada (número de instruções). Neste trabalho os dois fatores são considerados, mas em alguns casos uma redução considerável no tempo de execução pode vir acompanhada de um aumento no tamanho do código gerado. Entretanto, os resultados obtidos a partir dos testes realizados mostraram que esse aumento não é significativo, e é compensado pela redução no número de ciclos gastos na execução dos programas em que esse fato foi observado. O compromisso entre tamanho de código e velocidade de execução é também uma preocupação em [29].

A fim de ilustrar mais claramente o objetivo deste trabalho, um exemplo simples será utilizado a seguir. Por agora, basta observar a diferença entre as versões de pseudo-código fonte apresentadas, como se houvesse uma correspondência de um para um entre os comandos em linguagem de alto-nível e as instruções em linguagem *assembly* da arquitetura alvo.

```
(1)       i = 0
(2)    LOOP:
(3)       t = x[i] * h[i]
(4)       z = z + t
(5)       i = i + 1
(6)       if i < 64 goto LOOP
```

Figura 1.1: Produto interno entre dois vetores — multiplicação e acumulação em duas instruções separadas.

Considere o trecho de código na Figura 1.1, bastante comum nas aplicações de DSPs, que determina o produto interno entre dois vetores. O laço presente nesse trecho de código é uma soma de produtos realizada da seguinte forma:

- Em (3) o produto dos elementos de índice `i` dos *arrays* `x` e `h` é realizado, e o resultado armazenado em `t`.
- Em (4) o valor de `t` é acumulado na variável escalar `z`.

Esse laço contém 4 instruções e executa 64 vezes. Além da correspondência de um para um entre comandos em alto-nível e instruções *assembly*, considere ainda que a execução de cada comando/instrução gasta um ciclo de máquina. Assim, para se obter o produto interno dos vetores `x` e `h` seriam necessárias 5 instruções e $4 * 64 + 1$ ou 257 ciclos.

Considere agora o trecho de código na Figura 1.2 que determina o mesmo produto interno entre os mesmos dois vetores, porém realizando a multiplicação e a acumulação em uma única instrução. Como será mostrado com mais detalhes no Capítulo 2, essa é uma seqüência de operações bastante comum nas aplicações dos DSPs. Por isso, o conjunto de instruções de todos eles possui uma instrução especializada que realiza essas duas operações em uma única instrução, e, com a utilização de recursos de *pipeline*, pode fornecer um novo resultado a cada ciclo de máquina. Essa instrução é chamada de *instrução de MAC*[3].

```
(1)       i = 0
(2)   LOOP:
(3)       z += x[i] * h[i]
(4)       i = i + 1
(5)       if i < 64 goto LOOP
```

Figura 1.2: Produto interno entre dois vetores — multiplicação e acumulação em uma única instrução.

O laço presente nesse trecho de código também é uma soma de produtos, realizada da seguinte forma:

- Em (3) o produto dos elementos de índice `i` dos *arrays* `x` e `h` é realizado e acumulado na variável escalar `z`, em apenas uma instrução.

[3]Do inglês: **M***ultiply and* **A**c*umulate*.

Esse laço contém 3 instruções e se repete 64 vezes. Assim, para se obter o produto interno dos vetores `x` e `h` em `z` seriam necessários $3 * 64 + 1$ ou 193 ciclos de máquina. O tamanho deste código é de 4 instruções. Portanto, a utilização de uma instrução de MAC no lugar de duas instruções (uma multiplicação e uma adição) separadas pode proporcionar um ganho de até 25% no tempo de execução de um laço e até 20% no tamanho do seu código.

Considere agora o trecho de código na Figura 1.3 que determina o mesmo produto interno entre os mesmos dois vetores, porém realizando a multiplicação e a acumulação de dois elementos dos vetores em paralelo, em uma única instrução e gastando apenas um ciclo de máquina.

```
(1)       i = 0
(2)    LOOP:
(3)       t0 += x[i] * h[i] || t1 += x[i+1] * h[i+1]
(4)       i = i + 2
(5)       if i < 64 goto LOOP
(6)       z = t0 + t1
```

Figura 1.3: Produto interno entre dois vetores — multiplicação e acumulação em paralelo de dois elementos dos vetores em uma única instrução.

O laço presente nesse trecho de código também é uma soma de produtos, realizada da seguinte forma:

- Em (3) o produto dos elementos de índice `i` dos *arrays* `x` e `h` é realizado e acumulado na variável escalar `t0`; o produto dos elementos de índice `i+1` dos *arrays* `x` e `h` é realizado e acumulado na variável escalar `t1`, em apenas uma instrução.

Ao final de todas as iterações do laço, o produto dos elementos de índice par dos *arrays* encontra-se acumulado em `t0` e o produto dos elementos de índice ípar dos *arrays* encontra-se acumulado em `t1`. Dessa forma, é necessário adicionar `t0` e `t1` em `z`, se quisermos obter nessa variável o produto interno entre os vetores `x` e `h`.

Esse laço contém 3 instruções e se repete 32 vezes (note que o incremento de `i` é de 2, e não mais de 1). Assim, para se obter o produto interno dos vetores `x` e `h` em `z` seriam necessários $3 * 32 + 2$ ou 98 ciclos de máquina. O tamanho deste código é de 5 instruções. Portanto, a utilização de uma instrução de MAC que opera em paralelo sobre dois elementos dos vetores, no lugar de duas instruções (uma multiplicação e uma

adição) separadas, pode proporcionar um ganho de até 62% no tempo de execução de um laço. Neste caso o tamanho do código não sofreu alteração, mas veremos no Capítulo 5 que a utilização desse tipo de instrução implica um certo *overhead*, e cabe ao programador decidir se vale ou não a pena utilizá-la. Na maioria dos casos, como já foi dito, o ganho em desempenho é bastante significativo e compensa o *overhead* introduzido com essa instrução.

Tendo analisado este exemplo, não é difícil perceber que se duas ou mais unidades funcionais com capacidade de operação simultânea estão presentes no processador, é importante que um compilador para esse processador seja capaz de explorar essa funcionalidade. Para isso ele deve ser capaz de gerar código que executa duas ou mais iterações de um laço ao mesmo tempo, reduzindo, assim, o número de iterações do mesmo. As circunstâncias que permitem esse tipo de transformação em um programa sem modificar a sua semântica são analisadas nesta dissertação. O código na Figura 1.3 faz uso de uma instrução vetorial de MAC e reduz à metade o número de iterações do laço.

A contribuição deste trabalho concentra-se, portanto, na integração da capacidade de gerar instruções de multiplicação e acumulação em um compilador para DSP que não apresenta essa funcionalidade, e na paralelização dessa instrução em uma arquitetura SIMD[4].

1.4 Organização

Esta dissertação está organizada da seguinte maneira:

- O Capítulo 2 apresenta as principais características dos DSPs no que diz respeito a sua arquitetura, sistema de memória, organização da *datapath*, conjunto de instruções e modos de endereçamento. Descreve o funcionamento das suas unidades de multiplicação e acumulação (MAC) de uma maneira geral e introduz as unidades de MAC de duas famílias de DSPs: `ADSP2100` da *Analog Devices* [3] e `TMS320C2X` da *Texas Instruments* [51]. Também aborda os requisitos e as dificuldades presentes durante a geração de código para DSPs como conseqüência de suas características.

- O Capítulo 3 contém uma breve descrição do processador utilizado nos exemplos desta dissertação; seus registradores, modos de endereçamento, unidades funcionais e linguagem *assembly*.

- O Capítulo 4 descreve a implementação da detecção/geração de instruções de MAC e os resultados obtidos com a otimização.

[4]Do inglês: ***S**ingle **I**nstruction **M**ultiple **D**ata*.

- O Capítulo 5 descreve como foi implementada a vetorização das instruções de MAC, além de apresentar alguns exemplos de código gerado e uma análise dos resultados obtidos com as otimizações.
- O Capítulo 6 apresenta as principais contribuições deste trabalho e sugestões de extensão.
- O Capítulo 7 traz uma breve conclusão do que foi apresentado.

Capítulo 2

Características Gerais dos DSPs

2.1 Arquitetura

Os algoritmos básicos em telecomunicações são aqueles em que é preciso realizar operações com (ou processar) sinais elétricos. Tipicamente, os sinais são coletados em intervalos regulares de tempo, convertidos para a forma digital e processados usando operações aritméticas[1]. Alguns exemplos de processamento digital incluem: filtragem de sinais, convolução (mistura de dois sinais), correlação (comparação de dois sinais), cancelamento de eco, correção e amplificação, entre outros. Para que sejam capazes de realizar essas transformações eficientemente, os DSPs possuem certas características que os diferem dos processadores de propósito geral:

- Integram em um mesmo *chip*: processador, ROM e RAM, esta última geralmente estática, ou seja, mais rápida.
- Não possuem *cache* ou sistema de memória virtual.
- A maioria possui apenas aritmética de ponto-fixo [40] e [57], pois unidades de ponto-flutuante, geralmente presentes nos processadores de propósito geral, requerem uma área adicional de silício e dissipam mais energia.
- Podem realizar uma operação de multiplicação e uma operação de adição (instrução de MAC), no mesmo tempo que gastam para executar uma única instrução.
- Possuem a capacidade de realizar movimentação de dados de/para a unidade aritmética ao mesmo tempo em que realizam operações aritméticas e atualizações de apontadores.

[1] Daí o nome processamento de sinais digitais.

- Permitem mais de um acesso à memória em um único ciclo de máquina.
- Possuem modos de endereçamento, conjunto de instruções e unidades funcionais especializadas [18].
- Suas *datapaths* geralmente possuem uma topologia irregular, como reflexo da necessidade de otimizar a utilização das unidades funcionais e dos registradores, ao mesmo tempo em que é necessário minimizar a área total de silício e o consumo de energia do processador.
- Seus conjuntos de instruções são em geral bastante irregulares, utilizando diversos formatos de instruções, modos de endereçamento e registradores de uso específico.
- Executam aplicações que tipicamente processam uma seqüência de sinais captados por microfones, sensores, câmeras de vídeo, telefones, etc. armazenando-os na forma digital e submetendo-os a extensiva computação numérica para obter o resultado desejado. Uma operação bastante utilizada em DSPs é o produto interno de dois vetores A e B de tamanho N, dado por

$$A.B = \sum_{i=0}^{N-1} A[i] * B[i]$$

As seções seguintes discutem algumas dessas características com maiores detalhes, apresentando o seu impacto na geração de código eficiente para DSPs. Elas descrevem também como já foram tratados alguns dos problemas decorrentes destas. Referências mais completas sobre as características dos DSPs, e os problemas de geração de código para esse tipo de arquitetura podem ser encontradas em [8], [18] e [34].

2.2 Arquiteturas Vetoriais

Alguns DSPs possuem características de processadores vetoriais, como o *Countach40*, para o qual foi gerado código neste trabalho. Os chamados *processadores vetoriais* provêm instruções de alto-nível que operam sobre *vetores*. Enquanto os processadores de propósito geral possuem uma unidade genérica de execução que processa uma sub-função de cada vez (por isso são também chamados *processadores escalares*), os processadores vetoriais possuem múltiplas unidades de execução, uma para cada sub-função. Essa diferença ficará mais clara ao observar como esses dois processadores executam a soma de dois vetores. A Figura 2.1 é um trecho de código que realiza a soma elemento a elemento de dois vetores de 64 elementos e armazena o resultado em um terceiro vetor.

```
(1)       i = 0
(2)    LOOP:
(3)       z[i] = x[i] + h[i]
(4)       i = i + 1
(5)       if i < 64 goto LOOP
```

Figura 2.1: Soma elemento a elemento dos vetores `x` e `h`.

Considerando o trecho de código na Figura 2.1, a Figura 2.2 (a) ilustra como a adição de dois vetores de 64 elementos é realizada por uma arquitetura escalar: como apenas uma unidade de soma está presente neste processador, são executadas 64 iterações de um laço que executa a soma dos elementos `x[i]` e `h[i]` armazenando-a em `z[i]`. A Figura 2.2 (b) ilustra como a adição de dois vetores de 64 elementos é realizada por uma arquitetura vetorial com 64 unidades de soma. Neste caso, apenas uma instrução é executada que processa a soma em paralelo de todos os elementos dos vetores `x` e `h` armazenando os resultados em todos os elementos de `z` ao mesmo tempo.

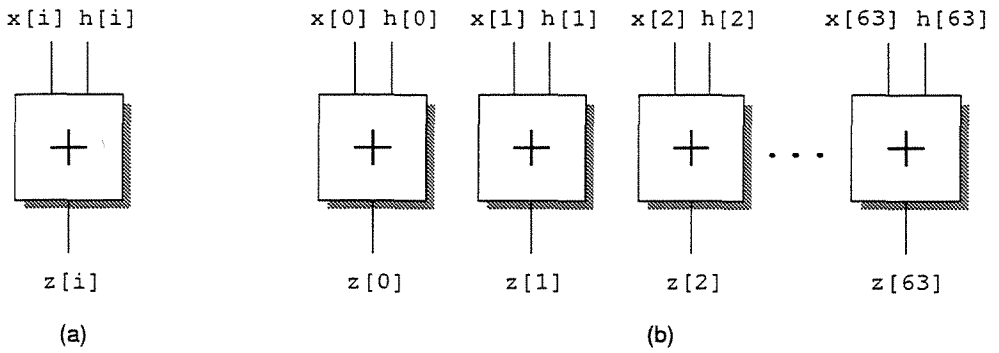


Figura 2.2: (a) Soma de dois vetores de 64 elementos realizada em uma arquitetura escalar. (b) Soma de dois vetores de 64 elementos realizada em uma arquitetura vetorial que possui 64 unidades de execução.

A seguir são apresentadas algumas das características mais importantes dos processadores vetoriais:

- Presença de múltiplas unidades funcionais operando em paralelo, cada uma apresentando recursos de *pipeline* e podendo terminar uma operação a cada ciclo de máquina.

- Diversos barramentos para transferência de dados, endereços e sinais de controle.
- Uma unidade de controle responsável pela decodificação das instruções e coordenação das unidades funcionais.
- Apresentam um conjunto de registradores vetoriais, cada um dos quais é um banco de tamanho fixo, contendo um único vetor de diversos elementos (por exemplo 64 palavras). Todas as operações são realizadas sobre registradores vetoriais.
- Apresentam um conjunto de registradores escalares. Eles podem ser usados para diversos fins, como por exemplo servir de *buffer* de entrada de dados para os registradores vetoriais.
- Uma única instrução vetorial pode corresponder à execução de um laço inteiro do programa.
- O tamanho do vetor sobre o qual as unidades funcionais operam é variável. Geralmente existe um registrador que armazena o tamanho máximo permitido para cada vetor. Se o tamanho de um vetor ultrapassar esse limite, o compilador será responsável por quebrá-lo e processá-lo separadamente. Essa técnica é chamada *strip-mining* e será apresentada em mais detalhes no Capítulo 5.

A exploração da natureza vetorial de um processador que apresenta essas características é essencial para o desempenho do código. Isso geralmente é feito por meio do chamado paralelismo de dados, que é a execução em paralelo da mesma operação sobre diferentes partes do conjunto de dados de entrada. A transformação de código escalar em código que utilize operações vetoriais é chamada vetorização, e é discutida em [26]. Detalhes sobre esse tipo de arquitetura e algumas otimizações que visam maximizar o uso das instruções vetoriais podem ser encontrados em [9] e [56].

2.3 Sistema de Memória

Uma estrutura bastante comum para o sistema de memória dos processadores de propósito geral é a de um único banco de memória, que o processador acessa por meio de um único conjunto de barramentos de endereço e dados [20] e [21]. Este modelo é conhecido como arquitetura de *Von Neumann* [46]. Neste modelo dados e instruções são armazenados em um mesmo banco de memória e apenas um acesso à memória ocorre a cada ciclo.

Uma operação típica em um DSP é uma instrução de multiplicação e acumulação (MAC), realizada em um único ciclo. Essa instrução requer que dois operandos sejam buscados da memória e multiplicados, e que seu produto seja somado ao valor presente no acumulador. As variações existentes em torno das unidades de multiplicação e acumulação e execução desta instrução serão apresentadas na Seção 2.4. Em uma arquitetura que segue o modelo de *Von Neumann* não é possível buscar a instrução e os dados no mesmo ciclo, do ponto de vista da memória principal, na ausência de *cache*. Este é um dos motivos pelos quais processadores convencionais não executam aplicações para DSPs de maneira eficiente.

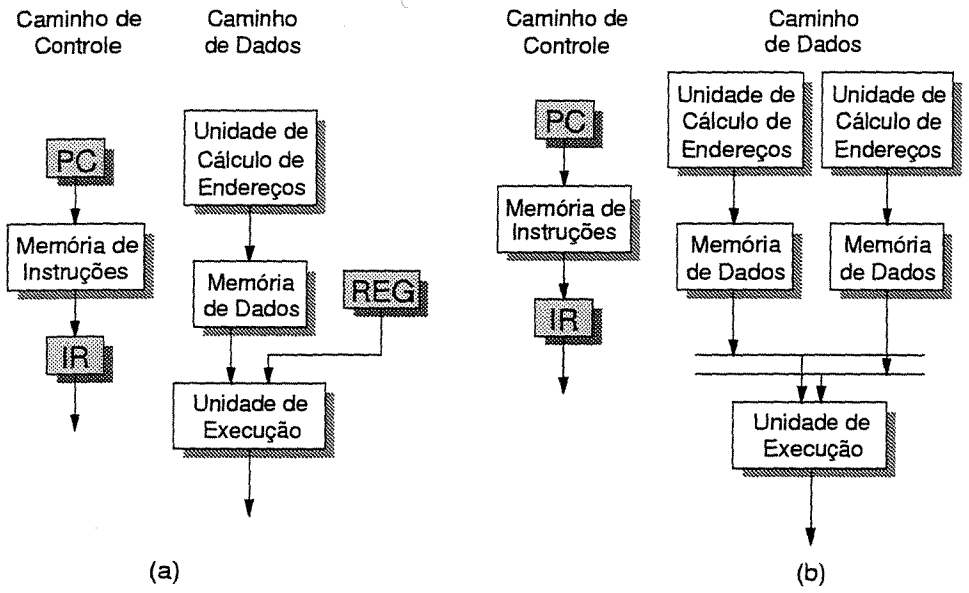


Figura 2.3: (a) Arquitetura Harvard. (b) Arquitetura Harvard modificada.

A solução que permite acessos paralelos à memória é conhecida como arquitetura Harvard e arquitetura Harvard modificada [12]. Basicamente esses modelos de arquitetura possuem memória de dados separada da memória de instruções, e permitem que instruções e dados sejam buscados da memória em apenas um ciclo. A arquitetura Harvard possibilita buscar uma palavra na memória de instruções e uma palavra na memória de dados durante o mesmo ciclo; este sistema requer quatro barramentos, dois de endereço e dois de dados. Um diagrama esquemático deste modelo pode ser visto na Figura 2.3 (a). Exemplos desta arquitetura são os DSPs da família `TMS320C2x` da *Texas Instruments* [51].

O TMS320C25 possui 128Kb de memória de dados, dividida em 512 páginas, cada uma contendo 128 palavras de 16 bits. Um registrador especial, DP, armazena o número da página corrente. Dessa forma, cada acesso à memória deve assegurar que DP contém o número da página que vai ser acessada. Isto é feito pela instrução LDPK, que carrega uma constante em DP. Uma otimização apresentada em [49] utiliza análise de fluxo de dados para remover as instruções do tipo LDPK que sejam redundantes, e assim reduzir o *overhead* introduzido por essas instruções. Um compilador otimizante para o TMS320C25 é apresentado em [33].

A Figura 2.3 (b) apresenta um diagrama esquemático da arquitetura conhecida como Harvard modificada. Exemplos desta arquitetura são os DSPs da família ADSP2100 da *Analog Devices* [3], DSP56000 da *Motorola* [35] e μPD77016 da *NEC* [37]. Dois dados são agora buscados da memória em um único ciclo de máquina. Uma vez que não é possível acessar um mesmo banco de memória no mesmo ciclo (quando isto é possível existem restrições quanto à localização dos operandos na memória), esta implementação requer três bancos de memória, um de instruções e dois de dados, geralmente chamados de bancos X e Y, cada um com seu conjunto individual de barramentos de dados e endereço.

Esta configuração aumenta a largura de banda dos acessos à memória, pois permite que três acessos ocorram em paralelo, um a instrução e dois a dados. No entanto, existe uma limitação neste sistema de memória: para que dois acessos à memória de dados sejam realizados em um único ciclo de máquina, os mesmos devem ser feitos a bancos diferentes, já que cada banco possui apenas uma porta de acesso. Além disto, há ainda uma série de restrições quanto à alocação de registradores nas arquiteturas que implementam esse sistema de memória. O problema de utilização eficiente dos dois bancos de memória foi abordado em [44] e [45], ao passo que [48] trata tanto do problema de utilização eficiente dos bancos de memória, quanto do problema de alocação de registradores para essas arquiteturas.

2.4 Unidades de Multiplicação e Acumulação

Uma das operações específicas das quais os DSPs fazem uso bastante intensivo é o produto interno entre dois vetores. A unidade funcional que realiza esta operação é conhecida como unidade de multiplicação e acumulação, ou unidade de MAC[2]. As unidades de multiplicação e acumulação normalmente utilizam recursos de *pipeline*, permitindo que o resultado de um produto e de uma adição, ou subtração, esteja disponível após um ciclo de máquina.

[2]Do inglês: **M***ultiply and* **AC***cumulate.*

Existem algumas diferenças entre unidades de multiplicação e acumulação de alguns fabricantes de DSPs. Como foi visto na seção anterior uma arquitetura Harvard possui apenas um banco de memória de dados (M), permitindo a busca de apenas um operando da memória a cada instrução. O outro operando deve estar armazenado em algum registrador (R) da *datapath*. Essa arquitetura é também chamada de *memory-register*, e é utilizada pela família de DSPs TMS320CXX da *Texas Instruments* [51], [52], [53] e [54]. Como um operando é sempre lido da memória, a latência das instruções pode ser grande. A Figura 2.4 (a) ilustra esse tipo de arquitetura.

Uma arquitetura Harvard modificada possui dois bancos de memória de dados (MX e MY) e permite a busca de dois operandos da memória a cada instrução. Com a utilização de recursos de *pipeline* esses dois operandos podem ser armazenados em registradores (RX e RY), ficando disponíveis para a instrução no ciclo seguinte. Enquanto os operandos são lidos dos registradores, dois novos acessos à memória de dados podem buscar dois novos operandos que passam novamente a residir nestes registradores. Por esse motivo, essa arquitetura é também chamada de *register-register*, ou *load-store*, e é utilizada pelas famílias de DSPs ADSP2100 da *Analog Devices* [3] e DSP56000 da *Motorola* [35]. O fato de os operandos serem sempre lidos de registradores não resulta em nenhuma latência na execução das instruções. A Figura 2.4 (b) ilustra esse tipo de arquitetura.

Uma outra diferença existente entre as unidades de multiplicação e acumulação está relacionada com o registrador P, que pode ser visto nas Figuras 2.4 (a) e 2.4 (b). O registrador P é um registrador de *pipeline* que armazena o produto dos dois operandos da operação de multiplicação.

Nos DSPs da família ADSP2100, o registrador P é apenas um registrador interno ao *pipeline* e não é visível ao conjunto de instruções do DSP. Já no caso dos DSPs da família TMS320CXX, o registrador P é visível ao conjunto de instruções do DSP e pode ser manipulado diretamente pelo programador, se este assim o desejar. A arquitetura para a qual foi gerado código neste trabalho [22] possui comportamento semelhante ao da família ADSP2100 da *Analog Devices*.

Independente da configuração da memória de dados dos DSPs, se um ou dois bancos estão presentes, e do funcionamento do registrador P, se visível ou não ao conjunto de instruções do processador, as suas unidades de multiplicação e acumulação utilizam-se de recursos de *pipeline* de dados para disponibilizar o resultado de uma instrução de multiplicação e acumulação após um ciclo de máquina, sem necessidade de aumentar a duração do ciclo. As unidades de MAC realizam ao mesmo tempo três operações independentes em cada ciclo de máquina. São elas:

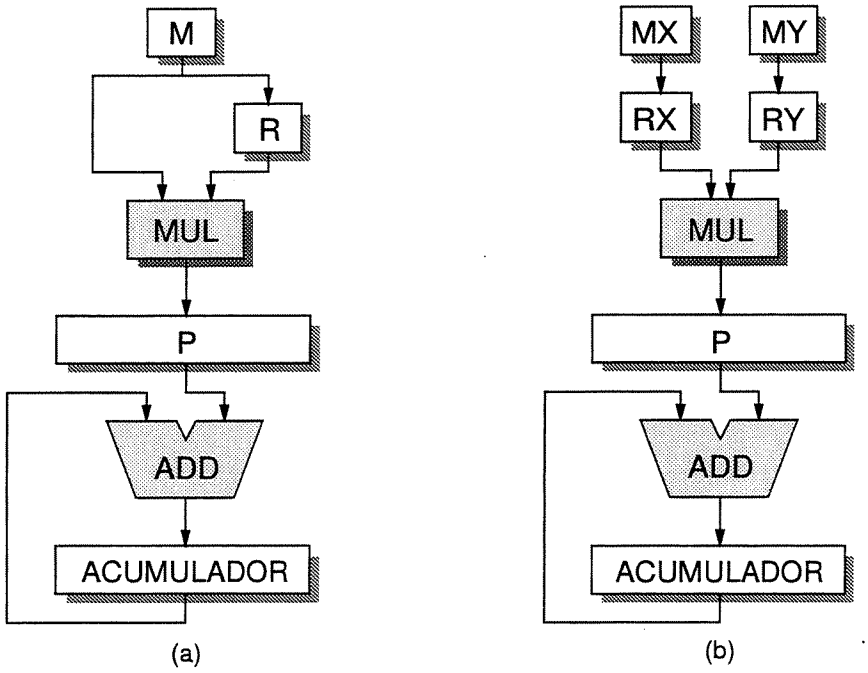


Figura 2.4: (a) Unidade de multiplicação e acumulação de uma arquitetura *memory-register* utilizando apenas um banco de memória. (b) Unidade de multiplicação e acumulação de uma arquitetura *register-register* utilizando dois bancos de memória.

1. Carrega nos registradores `R` ou `RX` e `RY` os operandos para a multiplicação que ocorrerá no ciclo seguinte.

2. Realiza o produto dos valores presentes em `R` e `M` ou `RX` e `RY` e armazena no registrador `P` o resultado obtido.

3. Adiciona o conteúdo do registrador `P`, armazenado no ciclo anterior, ao valor presente em `ACC`.

Com isto a unidade de multiplicação não precisa esperar que os operandos sejam carregados em `R` ou `RX` e `RY` para só então realizar o seu produto: eles terão sido carregados no ciclo anterior, sem nenhum *overhead*. Da mesma forma, a unidade de soma não precisa esperar pelo resultado do produto para realizar a acumulação com o valor presente em `ACC`: ele terá sido armazenado em `P` no ciclo anterior. A Figura 2.5 ilustra esses três estágios do *pipeline* de dados presente nas unidades de MAC. Em paralelo com o produto atual, a unidade de MAC realiza a acumulação do produto anterior e carrega os operandos para o próximo produto.

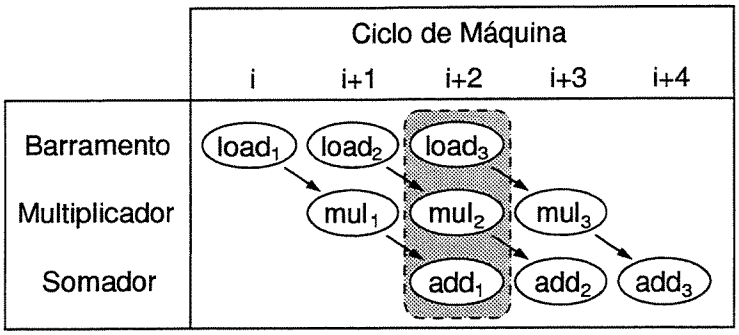


Figura 2.5: *Pipeline* de dados para uma instrução de MAC.

2.5 Modos de Endereçamento

Com a finalidade de preservar o desempenho do processador, a arquitetura dos DSPs possui uma unidade funcional especializada para o cálculo de endereços: a unidade de endereçamento, ou unidade geradora de endereços (AGU[3]). A presença de uma unidade dedicada exclusivamente para o endereçamento da memória permite que os endereços sejam calculados rapidamente, em paralelo com outros cálculos realizados na *datapath* do processador, minimizando assim o *overhead* que a geração de endereços representaria se essa operação fosse realizada pela mesma unidade funcional que executa os cálculos aritméticos.

Devido à necessidade de se manter compacto o tamanho do código, os DSPs são geralmente projetados de tal forma que as instruções sejam codificadas em apenas uma palavra. Em decorrência disto, o modo de endereçamento absoluto não é utilizado, mas sim o indireto, no qual um registrador especial de endereçamento, contendo o endereço do operando na memória, é especificado na instrução. Desta forma, para se acessar um dado na memória é necessário alocar um registrador de endereçamento e realizar operações aritméticas especializadas, como será visto a seguir.

Uma unidade de endereçamento típica utiliza operações aritméticas para realizar os cálculos dos endereços necessários para acessar operandos residentes em memória. Para isto, ela contém alguns registradores especializados, além de uma unidade aritmética simples. Seus registradores estão classificados em três grupos: registradores de endereçamento (RE), que armazenam endereços utilizados como apontadores para a memória, registradores de deslocamento (RD), que armazenam valores de deslocamentos utilizados para atualizar os RE, e registradores de módulo (RM), que armazenam valores de módulo

[3]Do inglês: *Address Generation Unit*.

utilizados para atualizar os RE. A unidade aritmética de uma AGU é capaz de implementar aritmética linear, em módulo e de *carry* reverso. Para tal ela possui três somadores completos: um somador de deslocamento, que pode somar um, menos um, o conteúdo de um RD ou o complemento de dois do conteúdo de um RD a um RE; um somador em módulo, que aplica a função $mod(RE) = RE - RM$ ao RE sempre que o resultado do somador de deslocamento for maior que $RM - 1$[4]; e um somador de *carry* reverso, que pode somar um, menos um, o conteúdo de um RD ou o complemento de dois do conteúdo de um RD a um RE, com o *carry* sendo propagado na direção inversa, isto é, do bit mais significativo para o menos significativo[5]. A Figura 2.6 mostra exemplos de como seria feito o endereçamento utilizando cada uma das operações aritméticas especializadas da unidade de endereçamento.

Na Figura 2.6 (a) o acesso ocorre a cada cinco posições de memória, já que foi usado o modo de endereçamento linear e RD contém o valor 5. Na Figura 2.6 (b) a tentativa de endereçar a memória além do limite superior (limite inferior + módulo) no modo de endereçamento em módulo, ocasiona o retorno ao limite inferior. Na Figura 2.6 (c) a utilização do modo de endereçamento de *carry* reverso resultou na inversão dos 4 bits menos significativos de RE (já que RD contém o valor 8) seguida de incremento de 1 e novamente da inversão dos 4 bits menos significativos.

Como é possível observar, os DSPs possuem unidades de endereçamento bastante especializadas que possibilitam diversos padrões de acesso à memória, visando melhorar o desempenho das aplicações. Uma vez que é muito comum que os dados estejam dispostos seqüencialmente na memória, (como por exemplo os *arrays* ou as variáveis automáticas no *frame*), os modos de endereçamento mais utilizados são os de auto-incremento e auto-decremento. É, portanto, essencial que o compilador seja capaz de explorar estes modos. Entretanto, para gerar código denso e eficiente que faça intenso uso dos modos de endereçamento de auto-incremento e auto-decremento, o compilador deve realizar uma cuidadosa disposição dos dados na memória. Alguns trabalhos já foram desenvolvidos com essa finalidade, como por exemplo [42], que propõe otimizar a seqüência de acesso das variáveis aplicando transformações algébricas sobre as árvores de expressão, a fim de minimizar o número de instruções de aritmética de endereços. Já [31] formula o mesmo problema como um problema de cobertura de grafos. Uma abordagem mais genérica é encontrada em [28] e [41], ao passo que o problema específico de gerar instruções de auto-incremento e auto-decremento para acessos a *arrays* foi estudado em [7], [11], [14] e [27].

[4] O endereçamento em módulo é útil na criação de *buffers* circulares, estruturas de dados muito comuns nas aplicações dos DSPs.

[5] O endereçamento de *carry* reverso é útil na computação da *Transformada de Fourier* (FFT, Fast Fourier Transform) de um sinal.

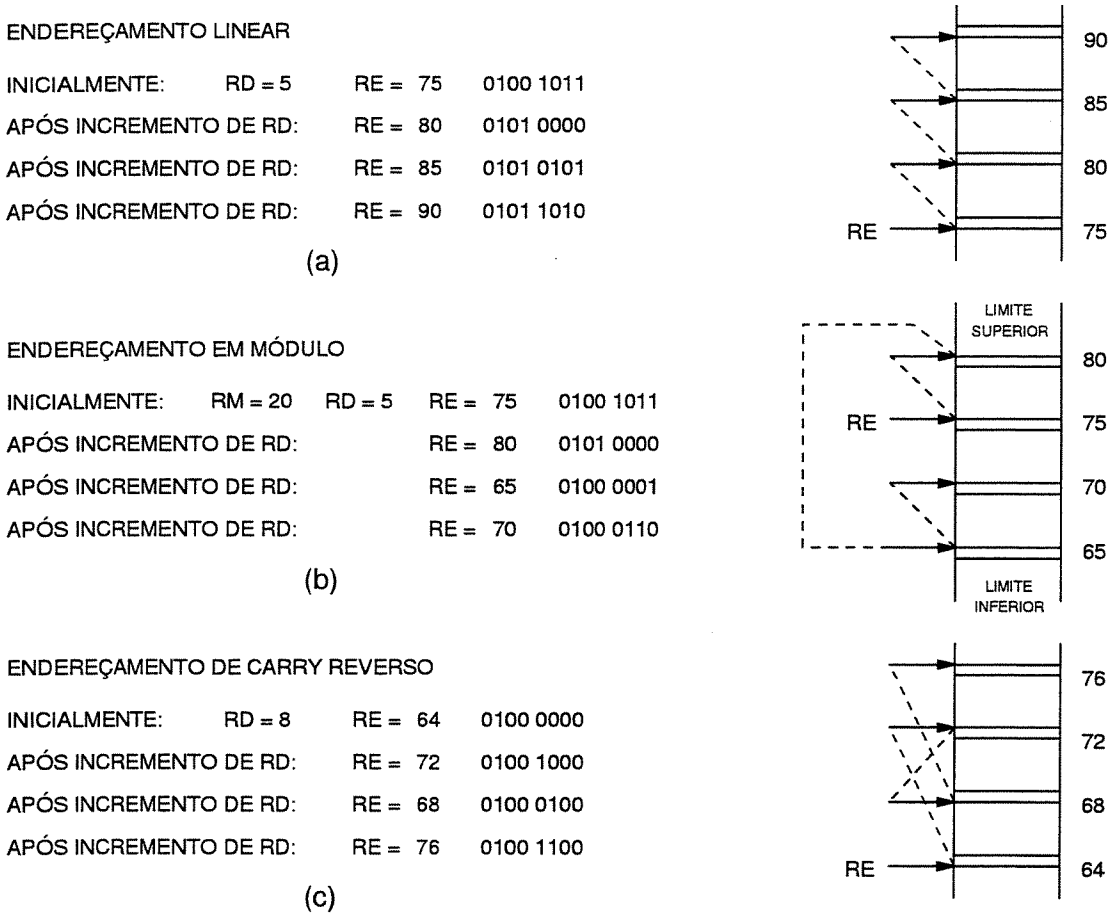


Figura 2.6: (a) Endereçamento utilizando aritmética linear. (b) Endereçamento utilizando aritmética em módulo. (c) Endereçamento utilizando aritmética de *carry* reverso.

2.6 Conjunto de Registradores

Quase todos os processadores de propósito geral possuem um conjunto de registradores, que são utilizados como operandos de diversas instruções sem restrições ou limitações. Os registradores de um conjunto são equivalentes, isto é, o seu uso não é determinado pela instrução sendo executada ou pela unidade funcional em operação naquele momento. Um conjunto de registradores com essa característica é chamado de homogêneo e nesse caso, via de regra, qualquer instrução pode ler ou escrever em qualquer um dos registradores.

Em funcão de sua natureza especializada, a maioria dos DSPs geralmente disponibiliza um pequeno número de registradores dedicados, ou seja, cujo uso está associado a um tipo de instrução e/ou a unidades funcionais específicas. Como exemplo, considere uma arqui-

tetura em que as instruções aritméticas devem ter como operandos apenas registradores pertencentes a um determinado conjunto, e cujo resultado deve ser sempre atribuído a um acumulador. Por essa razão, o conjunto de registradores dos DSPs que apresentem essa particularidade é chamado de heterogêneo, e essa característica faz com que as etapas de seleção de instruções e alocação de registradores sejam altamente dependentes uma da outra, o que torna a tarefa de geração de código para DSPs ainda mais complexa.

O problema de alocação de registradores para arquiteturas heterogêneas foi tratado em [5], [6] e [23]. A geração de código que explore essa estrutura heterogênea dos registradores não é um problema trivial. O uso ineficiente dos registradores é a principal razão da baixa qualidade do código gerado por compiladores comerciais para DSPs.

2.7 Conjunto de Instruções

Como foi mostrado nessa seção, os DSPs possuem algumas características que lhes são peculiares. Ao estudar o conjunto de instruções de alguns dos DSPs comerciais, é possível ainda verificar que eles disponibilizam certas operações bastante específicas, que não existem nos conjuntos de instruções de processadores de propósito geral. Essa característica é conhecida como especialização aritmética, e permite que os algoritmos de processamento de sinais digitais sejam executados em um número reduzido de ciclos de máquina.

A existência de instruções especializadas no conjunto de instruções dos DSPs contribui para a execução eficiente de algoritmos de processamento de sinais digitais tais como filtragem, convolução, correlação, entre outros, que frequentemente executam nesse tipo de processador. Por outro lado, isso dificulta a tarefa de geração de código para DSPs, e por isso vários trabalhos já foram realizados cujo principal objetivo é a seleção de instruções para DSPs, tais como [10], [30], [32] e [39].

Até pouco tempo, os programadores que desenvolviam aplicações de processamento de sinais digitais escreviam seu código em *assembly*, sem contar com nenhuma ferramenta que automatizasse seu trabalho. Com o aumento do tamanho dessas aplicações, escrever programas em *assembly* para DSPs tornou-se uma tarefa bastante difícil, e a possibilidade de cometer erros parece aumentar na mesma proporção. É interessante, então, poder escrever aplicativos para DSPs em uma linguagem de alto nível, e tirar proveito da modularidade, maior facilidade de implementação e portabilidade, entre outras vantagens decorrentes da utilização de uma linguagem de programação de alto nível.

Hoje os programadores que desenvolvem aplicativos para DSPs podem escrever seus programas em uma linguagem de alto nível, geralmente C, e também contar com ambientes de desenvolvimento que integram compilador, *linker*, depurador, *profiler* e simulador. Mas

para aproveitar as vantagens de programar em alto nível, é necessário que o compilador seja capaz de gerar todas as construções do conjunto de instruções da arquitetura alvo; caso contrário, certas funcionalidades estarão sendo desperdiçadas. É esta uma das motivações deste trabalho, ao tentar explorar ao máximo a possibilidade de gerar instruções de multiplicação e acumulação. Para conhecer outros trabalhos cujo enfoque é a geração de código eficiente para DSPs o leitor deve se referir a [15], [43] e [50].

A seguir são apresentados alguns tipos de instruções presentes no conjunto de instruções de alguns DSPs, e que servem como exemplos da especialização aritmética mencionada no início da seção.

2.7.1 MACs

As instruções de multiplicação e acumulação (MACs) são executadas pelas unidades de MAC descritas na Seção 2.4. São correspondentes, em baixo nível, ao comando `z += x * h`, que levaria dois ciclos para executar em um processador de propósito geral. No entanto, em DSPs cujas unidades de MAC possuam recursos de *pipeline*, um novo produto pode ser acumulado a cada ciclo. O MAC é a operação básica dos DSPs, e muitos deles são projetados de modo a prioritizar a execução eficiente dessa instrução, como por exemplo os DSPs da família `DSP56000` da *Motorola* [35]. A arquitetura do `DSP56K` possui a forma de uma operação de MAC: dois operandos são enviados para a unidade de multiplicação e o resultado é somado ao valor presente em um acumulador. Como pode ser observado na Figura 2.7 esse processo é realizado com a utilização de duas memórias separadas, (`X` e `Y`) que alimentam uma unidade de MAC. Como as memórias e a unidade de MAC são independentes, o DSP pode realizar duas movimentações de dados, um produto e uma soma em uma única operação.

2.7.2 *Hardware Loops*

As aplicações que executam em DSPs tipicamente processam longas seqüencias de sinais armazenados na forma digital, submetendo-os a extensiva computação numérica. É portanto natural que os programas que manipulam esses sinais contenham diversos laços, já que os sinais são armazenados como elementos de *arrays* e operações matemáticas devem ser realizadas sobre cada um desses elementos. Entretanto um certo *overhead* está associado à execução de laços, pois estes possuem uma instrução de teste ao final de cada iteração, e isso resulta em um impacto negativo sobre o desempenho do laço. Visando eliminar este *overhead*, os DSPs possuem um *hardware* especial para execução de laços cujo número de iterações pode ser determinado em tempo de compilação. Essa pequena

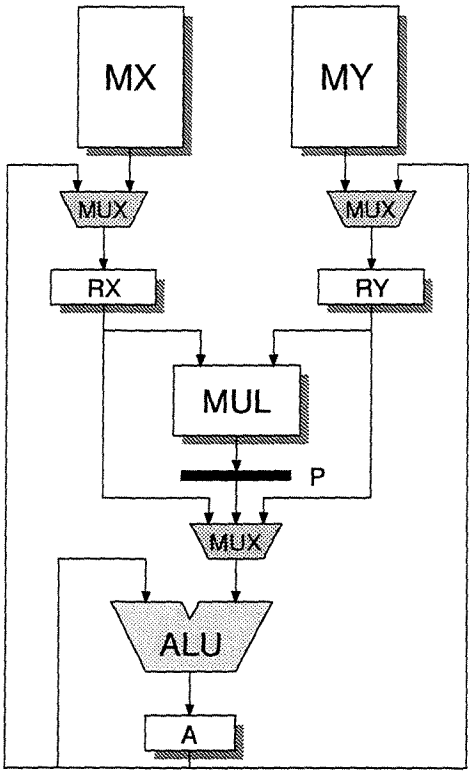
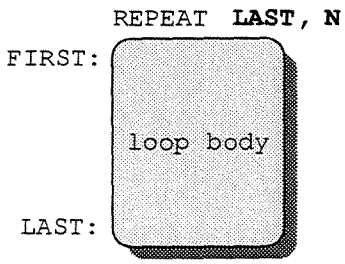


Figura 2.7: Arquitetura dos DSPs da família DSP56000 da *Motorola*.

unidade geralmente possui três registradores: LS, que armazena o endereço da primeira instrução do laço, LF, que armazena o endereço da última instrução do laço, e LC, que armazena o número de iterações do laço. Um *Hardware* de controle possibilita a execução dos chamados *hardware loops*, ou *zero-overhead loops*, que são laços cujo *overhead* de teste é zero.

A Figura 2.8 (a) é o esquema de uma instrução do tipo REPEAT presente no conjunto de instruções de um DSP capaz de executar *hardware loops*. Essa instrução possui dois operandos: o endereço da última instrução do laço, *LAST*, e o número de iterações do laço, N.

A execução de um *hardware loop* pela unidade de controle de laços segue o algoritmo apresentado na Figura 2.8 (b). O endereço da primeira instrução do laço, FIRST, é armazenado no registrador LS, o endereço da última instrução do laço, LAST, é armazenado no registrador LF e o número de iterações do laço, N, é armazenado no registrador LC. A cada iteração, PC é comparado com LF a fim de determinar se a última instrução do laço foi lida. Se PC for diferente de LF, PC é incrementado de 1 e a execução prossegue com



(a)

```
hardware_loop() {
  LS = FIRST;
  LF = LAST;
  LC = N;
  while (TRUE) {
    if (PC == LF) {
      if (LC == 1) {
        PC = LF + 1;
        break;
      }
      LC = LC - 1;
      PC = LS;
    }
    PC = PC + 1;
  }
}
```

(b)

Figura 2.8: (a) Exemplo de utilização de uma instrução do tipo REPEAT em linguagem *assembly*. (b) Algoritmo da execução de um *hardware loop*.

a próxima instrução do laço. Se PC for igual a LF, então a última instrução do laço foi buscada e o contador do laço (LC) é comparado com 1. Se LC for igual a 1, a execução do laço chegou ao seu fim e a execução continua na instrução seguinte ao laço (LF + 1). Se LC for diferente de 1, esse registrador é então decrementado de 1 e PC é carregado com o valor de LS, a primeira instrução do laço, para a execução de mais uma iteração.

Algumas unidades de *hardware loop* permitem a presença de laços aninhados, e para tal devem salvar o conteúdo dos registradores PC, LS, LF e LC na pilha do sistema antes de carregá-los com os valores necessários à execução de uma instrução de REPEAT. Ao final da última iteração do laço o conteúdo de cada um desses registradores é restaurado com os valores presentes no topo da pilha. O nível máximo de aninhamento permitido varia com o tamanho da pilha e outras características do processador.

2.7.3 *Bit Reversal*

A instrução de *bit reversal*, que realiza a inversão bit a bit do seu operando (bit_0 torna-se bit_{31}, bit_1 torna-se bit_{30}, e assim por diante, para um operando de 32 bits), está presente no conjunto de instruções de vários DSPs a fim de auxiliar na computação da transformada de Fourier, uma operação matemática bastante freqüente em DSPs.

2.7.4 Instruções Condicionais

O recurso de execução condicional está presente na maioria dos DSPs com a finalidade de evitar os atrasos causados por instruções de desvio em um programa. A execução das chamadas instruções condicionais é determinada pelo estado de um registrador, FLAG, e por um campo, COND, presente em cada uma dessas instruções. O campo COND indica qual bit no registrador FLAG deve estar em 0 ou 1 para que a instrução seja executada. A utilização das instruções condicionais transforma dependências de controle em um programa em dependências de dados, minimizando o *overhead* das instruções de desvio. O problema de geração de instruções condicionais foi estudado em [1] e [25].

```
(1)   int if_then(short a[], int codeword, int mask, short theta)
(2)   {
(3)     int i, sum, cond;
(4)     sum = 0;
(5)
(6)     for (i = 0; i < 32; i++) {
(7)       cond = codeword & mask;
(8)       if (theta == cond)
(9)         sum += a[i];
(10)      else
(11)        sum -= a[i];
(12)      mask = mask << 1;
(13)    }
(14)    return sum;
(15)  }
```

Figura 2.9: Um laço contendo um comando do tipo *if-then-else*. O elemento a[i] do vetor tanto pode ser somado quanto subtraido de sum.

O trecho de código na Figura 2.9 é um exemplo em que a utilização de instruções condicionais traz benefícios para o desempenho do código gerado. Uma possibilidade ao gerar código para a função mostrada é utilizar instruções de desvio do tipo *branch*: desviar a execução do programa para uma instrução de ADD se a condição for verdadeira e para uma instrução de SUB se a condição for falsa. No entanto, como cada instrução de desvio tem a ela associada diversos *delay slots* (o número exato depende da profundidade do *pipeline* do processador), essa abordagem pode se mostrar ineficiente, por exigir mais ciclos de máquina do que os necessários à execução das instruções. A utilização de instruções condicionais, por outro lado, elimina a necessidade de desviar a execução do programa

para o trecho adequado de código depois de verificar se a condição é verdadeira ou falsa. Com essa abordagem as instruções de `ADD` e `SUB` são substituídas pelas instruções `CADD` e `CSUB`, condicionais ao valor de um bit no registrador `FLAG`, especificado pelo campo `COND` na instrução. Esse método também permite utilizar uma técnica conhecida como *software pipelining* ([9], [21] e [24]) para alterar a estrutura do laço e com isso obter um desempenho ainda maior se comparado ao código original com instruções de desvio.

Capítulo 3

Arquitetura DSP-V

Este capítulo contém a descrição da arquitetura modelo *DSP-V*, (*DSP Vetorial*), que foi criada a fim de ilustrar os exemplos nesta dissertação. Detalhes específicos sobre o processador utilizado para realizar este trabalho (*Countach40*, descrito em [22]), incluindo o seu conjunto de instruções, são propriedade intelectual da *Conexant Systems Inc.* Somente alguns aspectos da arquitetura DSP-V serão descritos, em especial aqueles que forem necessários à compreensão das otimizações que serão apresentadas nos Capítulos 4 e 5.

3.1 Características Gerais

A arquitetura DSP-V, cujo código *assembly* figura nos exemplos desta disertação, é um processador do tipo DSP de ponto-fixo. Suas características mais importantes estão destacadas a seguir:

- Arquitetura SIMD, capaz de realizar a mesma operação sobre múltiplos dados em um único ciclo de máquina.

- O sistema de memória segue o esquema de Arquitetura Harvard, como descrito no Capítulo 2, em que a memória de dados e a memória de programa constituem bancos independentes (os blocos "Memória de Dados"e "ROM"podem ser vistos na Figura 3.1), cada qual com os seus próprios barramentos de dados e de endereços, permitindo acesso simultâneo às instruções e aos dados.

- O bloco de memória de dados possui três portas, como mostra a Figura 3.1, que permitem duas leituras e uma escrita em um único ciclo de máquina.

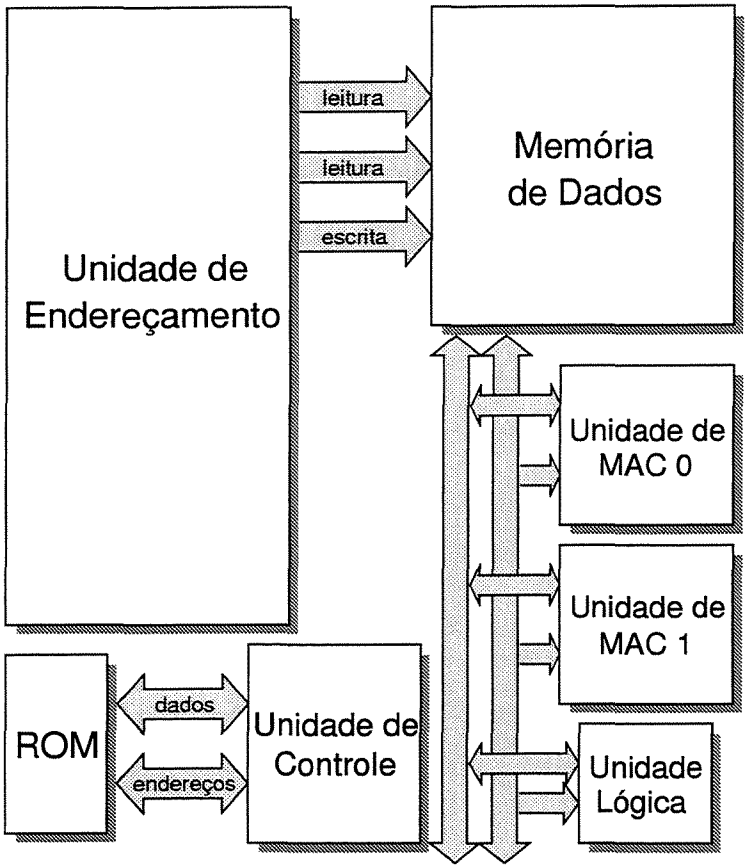


Figura 3.1: A arquitetura modelo DSP-V.

- Cada instrução executa em um ciclo de máquina, a não ser que ocorra um *pipeline stall.*

- A unidade de endereçamento opera em paralelo com as outras unidades, minimizando o *overhead* de cálculo de endereços. Ela possui capacidade de realizar auto-incremento e auto-decremento nos registradores de endereçamento à memória, reduzindo o número de instruções necessárias para acessar dados que ocupam posições contíguas na memória.

- A unidade de controle na Figura 3.1, inclui o registrador PC e uma pilha de execução que permite até 16 chamadas à subrotinas.

3.2 Unidades Funcionais

3.2.1 Unidades de MAC

As unidades de MAC realizam operações aritméticas envolvendo adição e multiplicação. Executam todas as operações em um ciclo e utilizam recursos de *pipeline*. Na arquitetura DSP-V existem duas unidades de multiplicação e acumulação como a que está sendo mostrada na Figura 3.2, com capacidade de operação simultânea sobre dados diferentes, cada uma contendo um somador de 32 bits, um multiplicador de 16 bits e dois acumuladores de 32 bits. Estão presentes ainda na unidade de MAC dois registradores (`X` e `Y`) que são os operandos para o multiplicador e um registrador interno, registrador `P` na Figura 3.2, que armazena o produto dos operandos submetidos ao multiplicador. Quando a segunda unidade de MAC é ativada por de uma instrução especial de controle, como será visto na Seção 3.5.5 diz-se que estão operando em *modo dual* e nesse caso os operandos são enviados para as unidades da seguinte maneira:

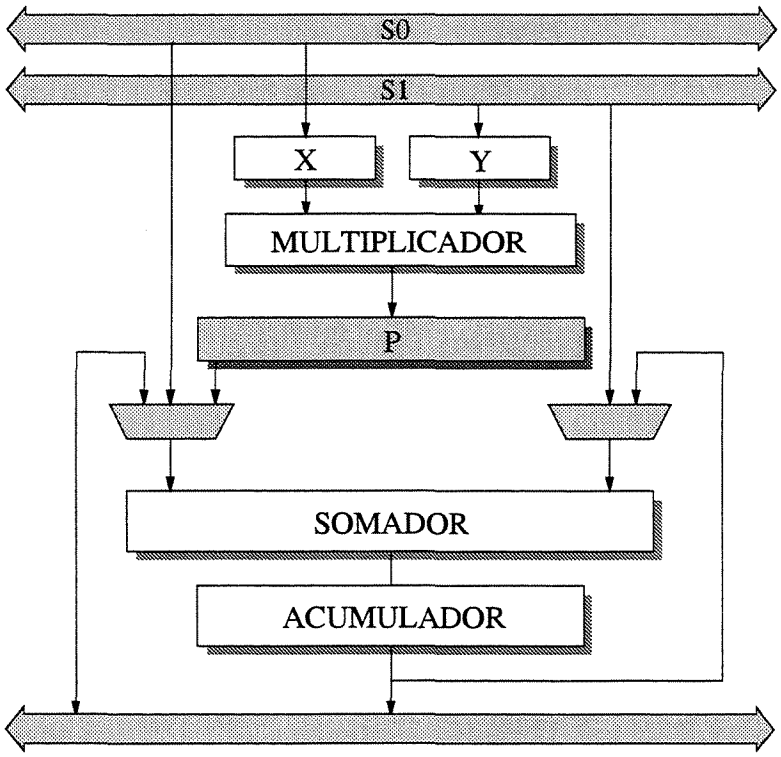


Figura 3.2: Uma unidade de MAC da arquitetura modelo DSP-V.

(a) Operandos de 32 bits: os 16 bits mais significativos dos operandos são enviados para a primeira unidade de MAC e os 16 bits menos significativos são enviados para a segunda unidade de MAC.

(b) Operandos de 16 bits: os operandos são duplicados no barramento e em seguida os seus 16 bits mais significativos são enviados para a primeira unidade de MAC e os 16 bits menos significativos são enviados para a segunda unidade de MAC, o mesmo comportamento de operandos de 32 bits; os dados são, portanto, replicados nas duas unidades de MAC.

O modo de operação das unidades de MAC é controlado pelo programador por meio do bit 15 do registrador `cntrl`. As unidades de MAC tomam seus operandos de registradores ou da memória, mas armazenam o resultado de suas operações apenas em um dos seus acumuladores internos.

3.2.2 Unidade Lógica

A arquitetura DSP-V contém uma unidade lógica de 32 bits, com capacidade de operação sobre um operando de 32 bits ou operação simultânea sobre dois operandos de 16 bits, quando especificado o modo dual. O modo de operação da unidade lógica é controlado pelo programador por meio do bit 14 do registrador `cntrl`. A unidade lógica toma seus operandos de registradores ou da memória e armazena o resultado de suas operações em registradores ou na memória, em um ciclo de máquina.

3.3 Registradores

Os seguintes registradores estão presentes no DSP-V:

- `acc0-acc3`: 4 acumuladores de 32 bits, 2 em cada uma das unidades de multiplicação. Os 16 bits mais/menos significativos dos acumuladores podem ser acessados separadamente, usando a sintaxe `accxH` para os 16 bits mais significativos e `accxL` para os 16 bits menos significativos.

- `r0-r31`: 32 registradores de 16 bits, de propósito geral.

- `p0-p7`: 8 registradores de endereçamento, que podem apontar para uma ou duas palavras de memória.

- `cntrl`: registrador de 16 bits de *flags*, subdividido da maneira a seguir, e mostrado na Figura 3.3.

  - Bits 0-7: controle de acesso à memória, programa o tamanho do operando apontado pelo registrador p correspondente — 0 = 16 bits e 1 = 32 bits.
  - Bit 8, *carry1*: igual a 1 se um "vai-um" houver sido gerado por uma operação aritmética executada pela segunda unidade de MAC.
  - Bit 9, *neg1*: igual a 1 se o resultado de uma operação aritmética executada pela segunda unidade de MAC for um número negativo.
  - Bit 10, *zero1*: igual a 1 se o resultado de uma operação aritmética executada pela segunda unidade de MAC for igual a zero.
  - Bit 11, *carry0*: igual a 1 se um "vai-um" houver sido gerado por uma operação aritmética executada pela primeira unidade de MAC.
  - Bit 12, *neg0*: igual a 1 se o resultado de uma operação aritmética executada pela primeira unidade de MAC for um número negativo.
  - Bit 13, *zero0*: igual a 1 se o resultado de uma operação aritmética executada pela primeira unidade de MAC for igual a zero.
  - Bit 14: modo de operação da unidade lógica — 0 = simples e 1 = dual.
  - Bit 15: modo de operação das unidades de multiplicação e acumulação — 0 = simples e 1 = dual.

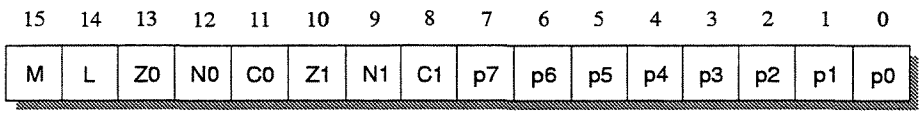


Figura 3.3: O registrador de controle `cntrl` e como é subdividido.

3.4 Modos de Endereçamento

A unidade de endereçamento à memória, que pode ser vista na Figura 3.1, oferece diversos métodos de acesso à memória de dados, para tipos de 16 e 32 bits. Os seguintes modos de endereçamento estão disponíveis no DSP-V:

- Direto: o operando é o registrador ou o endereço de memória no qual reside o dado a ser utilizado pela instrução.

  Ex1: `add acc0, r0, r1`

  Ex2: `mov r0, 0x42`

- Indireto: o operando é o registrador no qual reside o endereço de memória contendo o dado a ser utilizado pela instrução.

  Ex: `add acc0, r0, *p0`

- Indireto com auto-incremento/auto-decremento: o operando é o registrador no qual reside o endereço de memória contendo o dado a ser utilizado pela instrução. Após a execução da instrução o conteúdo do registrador é incrementado/decrementado de um valor igual ao tamanho da palavra (em *bytes*).

  Ex1: `add acc0, r0, *p0++`

  Ex2: `add acc0, r0, *p1--`

- Imediato: o operando é o dado a ser utilizado pela instrução.

  Ex: `add acc0, r0, 0x1`

3.5 Conjunto de Instruções

3.5.1 Instruções Aritméticas

- add: realiza a soma de dois operandos e armazena o resultado em um acumulador.
  Sintaxe:
  Modo simples:

  ```
  add   accN, src0, src1
  ```

  Modo dual:

  ```
  add   accN, src0, src1 || add   accN+2, src0, src1
  ```

- addi: realiza a soma de dois operandos, o segundo dos quais é um imediato, e armazena o resultado em um acumulador.
  Sintaxe:
  Modo simples:

  ```
  addi  accN, src0, immed
  ```

  Modo dual:

  ```
  addi  accN, src0, immed || addi  accN+2, src0, immed
  ```

- `sub`: realiza a subtração dos operandos e armazena o resultado em um acumulador.
  Sintaxe:
  Modo simples:
  ```
  sub   accN, src0, src1
  ```
  Modo dual:
  ```
  sub   accN, src0, src1 || sub   accN+2, src0, src1
  ```

- `subi`: realiza a subtração de dois operandos, o segundo dos quais é um imediato, e armazena o resultado em um acumulador.
  Sintaxe:
  Modo simples:
  ```
  subi  accN, src0, immed
  ```
  Modo dual:
  ```
  subi  accN, src0, immed || subi  accN+2, src0, immed
  ```

- `mul`: realiza o produto de dois operandos e armazena o resultado em um acumulador.
  Sintaxe:
  Modo simples:
  ```
  mul   accN, src0, src1
  ```
  Modo dual:
  ```
  mul   accN, src0, src1 || mul   accN+2, src0, src1
  ```

- `mac`: realiza o produto de dois operandos, armazena o resultado em um registrador de *pipeline*, invisível ao programador, soma esse resultado ao valor presente no acumulador e armazena o resultado no mesmo acumulador.
  Sintaxe:
  Modo simples:
  ```
  mac   accN, src0, src1
  ```
  Modo dual:
  ```
  mac   accN, src0, src1 || mac   accN+2, src0, src1
  ```

- `cmp`: realiza a subtração de dois operandos sem armazenar o resultado em qualquer acumulador, afetando o bit 11 do registrador `cntrl`, de acordo com o resultado.
  Sintaxe:
  Modo simples:
  ```
  cmp   src0, src1
  ```
  Modo dual:
  ```
  cmp   src0, src1 || cmp   src0, src1
  ```

- `cmpi`: realiza a subtração de dois operandos, o segundo dos quais é um imediato, sem armazenar o resultado em qualquer acumulador, afetando o bit 11 do registrador `cntrl`, de acordo com o resultado.
  Sintaxe:
  Modo simples:
  ```
  cmpi  src0, immed
  ```
  Modo dual:
  ```
  cmpi  src0, immed || cmpi  src0, immed
  ```

3.5.2 Instruções Lógicas

- `or`: realiza a função lógica OR de dois operandos e armazena o resultado em um registrador ou na memória.
  Sintaxe:
  Modo simples:
  ```
  or    dst, src0, src1
  ```
  Modo dual:
  ```
  or    dst, src0, src1 || or    dst, src0, src1
  ```

- `ori`: realiza a função lógica OR de dois operandos, o segundo dos quais é um imediato, e armazena o resultado em um registrador ou na memória.
  Sintaxe:
  Modo simples:
  ```
  ori   dst, src0, immed
  ```
  Modo dual:
  ```
  ori   dst, src0, immed || ori   dst, src0, immed
  ```

- and: realiza a função lógica AND de dois operandos e armazena o resultado em um registrador ou na memória.
  Sintaxe:
  Modo simples:
  ```
  and   dst, src0, src1
  ```
  Modo dual:
  ```
  and   dst, src0, src1 || and   dst, src0, src1
  ```

- `andi`: realiza a função lógica AND de dois operandos, o segundo dos quais é um imediato, e armazena o resultado em um registrador ou na memória.

Sintaxe:
Modo simples:

```
andi  dst, src0, immed
```

Modo dual:

```
andi  dst, src0, immed || andi  dst, src0, immed
```

- `lsr`: realiza o *shift* lógico de `shamt` posições para a direita de um operando e armazena o resultado em um registrador ou na memória.
  Sintaxe:
  Modo simples:

```
lsr   dst, src, shamt
```

  Modo dual:

```
lsr   dst, src, shamt || lsr   dst, src, shamt
```

- `lsl`: realiza o *shift* lógico de `shamt` posições para a esquerda de um operando e armazena o resultado em um registrador ou na memória.
  Sintaxe:
  Modo simples:

```
lsl   dst, src, shamt
```

  Modo dual:

```
lsl   dst, src, shamt || lsl   dst, src, shamt
```

- `asr`: realiza o *shift* aritmético de `shamt` posições para a direita de um operando e armazena o resultado em um registrador ou na memória.
  Sintaxe:
  Modo simples:

```
asr   dst, src, shamt
```

  Modo dual:

```
asr   dst, src, shamt || asr   dst, src, shamt
```

3.5.3 Instruções de Movimentação de Dados

- `mov`: move o conteúdo do operando fonte para o operando destino.
  Sintaxe: `mov  dst, src`
  Ex: `mov  *p0, r0`
  Move o conteúdo do registrador `r0` para a posição de memória apontada pelo registrador `p0`.

- `load`: carrega o conteúdo da posição de memória especificada pelo operando fonte no operando destino.

Sintaxe: `load dst, src`
Ex: `load p0, 0x1BA4`
Carrega o conteúdo da posição de memória `0x1BA4` no registrador `p0`.

- `loadi`: carrega o imediato especificado pelo operando fonte no operando destino.
  Sintaxe: `loadi dst, immed`
  Ex: `loadi p0, 0x1BA4`
  Carrega o valor `0x1BA4` no registrador `p0`.

3.5.4 Instruções de Controle

- `jmp`: desvia a execução do programa incondicionalmente para o endereço especificado pelo operando da instrução.
  Sintaxe: `jmp tgt`
  Ex: `jmp 0x42`

- `jset`: desvia a execução do programa para o endereço especificado pelo operando da instrução se o n-ésimo bit do registrador `cntrl` estiver em 1 (para $8 <= N <= 13$).
  Sintaxe: `jset #N, tgt`
  Ex: `jset #11, 0x42`

- `jclr`: desvia a execução do programa para o endereço especificado pelo operando da instrução se o n-ésimo bit do registrador `cntrl` estiver em 0 (para $8 <= N <= 13$).
  Sintaxe: `jclr #N, tgt`
  Ex: `jclr #12, 0x42`

- `jsr`: desvia a execução do programa para o endereço especificado pelo operando da instrução salvando o endereço de retorno $(PC + 1)$ na pilha de execução.
  Sintaxe: `jsr tgt`
  Ex: `jsr 0xff`

- `rsr`: desvia a execução do programa para o endereço de retorno no topo da pilha de execução, previamente armazenado pela instrução `jsr`.
  Sintaxe: `rsr`

- `noop`: utiliza um ciclo e não possui qualquer efeito.
  Sintaxe: `noop`

Observação: as instruções de controle, com exceção de `noop`, apresentam um *delay slot*.

3.5.5 Instruções de Ajuste dos Modos de Operação

As unidades funcionais do DSP-V possuem dois modos de operação: o modo simples e o modo dual. Para alternar entre esses dois modos de operação é necessário ajustar o bit correspondente no registrador `cntrl` (a Figura 3.3 mostra como esse registrador é subdividido). O bit 14 controla o modo de operação da unidade lógica e bit 15 controla o modo de operação das unidades de MAC. A seguir são mostradas as instruções que produzem esse efeito.

- `ori cntrl, cntrl, 0x8000`: ajusta o modo de operação dual das unidades de multiplicação e acumulação.
- `andi cntrl, cntrl, 0x7FFF`: ajusta o modo de operação simples das unidades de multiplicação e acumulação.
- `ori cntrl, cntrl, 0x6000`: ajusta o modo de operação dual da unidade lógica.
- `andi cntrl, cntrl, 0xBFFF`: ajusta o modo de operação simples da unidade lógica.

Os registradores de endereçamento do DSP-V podem ser configurados para acessar operandos de 16 ou 32 bits na memória, bastando para tal ajustar em 0 ou 1 o bit de número correspondente ao registrador de endereçamento no registrador de controle `cntrl`, bit 0 para o registrador `p0`, bit 1 para o registrador `p1`, e assim por diante. Abaixo são mostrados alguns exemplos de instruções que controlam o acesso à memória para alguns dos registradores de endereçamento.

- `ori cntrl, cntrl, 0x3`: ajusta os registradores `p0` e `p1` para acessarem 32 bits de memória.
- `andi cntrl, cntrl, 0xFFFC`: ajusta os registradores `p0` e `p1` para acessarem 16 bits de memória.
- `ori cntrl, cntrl, 0x80`: ajusta o registrador `p7` para acessar 32 bits de memória.
- `andi cntrl, cntrl, 0xFF7F`: ajusta o registrador `p7` para acessar 16 bits de memória.

3.6 Exemplos

A Figura 3.4 mostra uma função em C que realiza a convolução de dois *arrays* de sinais que são passados como parâmetros. A operação de convolução corresponde a realizar o produto interno do primeiro vetor pelo segundo invertido. O produto dos elementos é acumulado na variável y e retornado pela função.

A Figura 3.5 mostra a mesma função que realiza a convolução de dois *arrays* de sinais, porém em código *assembly* do DSP-V. No corpo do laço — comandos de números (8) a (21) — é realizada a operação de MAC (linha 13) sobre dois elementos dos *arrays* px e ph, acumulando os resultados parciais em acc1 e acc3; ph (p0) é decrementado de um e px (p1) e i (acc0) são incrementados de um, até que este último atinja o valor de LENGTH/2. Como duas instruções de MAC são realizadas a cada iteração do laço, apenas metade das iterações do programa original em C são necessárias. Os comandos de números (10) e (16) ajustam o modo de operação das unidades de MAC e o acesso à memória pelos registradores de endereçamento p0 e p1. Terminado o laço, em (23) é realizada a redução de y, isto é, a soma dos seus valores parciais; seu valor final é armazenado em acc0, a convenção do *assembly* do DSP-V para armazenar o valor retornado por uma função.

A Figura 3.6 mostra uma função em C que realiza o cálculo da média aritmética dos elementos de um *array* que é passado como parâmetro. A soma de todos os seus elementos é armazenada na variável avg, dividida pelo número de elementos do *array*, definido como LENGTH e retornada pela função.

A Figura 3.7 mostra a mesma função que realiza a média aritmética dos elementos de um *array*, porém em código *assembly* do DSP-V. No corpo do laço — comandos de números (8) a (21) — é realizada a soma dos elementos de índice par de ph em acc0 e dos elementos de índice ímpar de ph em acc2; ph (p0) e i (acc1) são incrementados de um, até que este último atinja o valor de LENGTH/2. Os comandos de números (10) e (16) ajustam o modo de operação das unidades de MAC e o acesso à memória pelo registrador de endereçamento p0. Terminado o laço, em (23) é realizada a redução de avg em acc0; em (25) avg (acc0) é dividido por LENGTH. Como LENGTH = 16, a operação corresponde a um deslocamento de 4 bits para a direita. O resultado é armazenado no registrador r1. Em (29) r0 recebe o valor 1 se avg era inicialmente negativo (acc0 < 0); em (32) avg (r1) é ajustado de r0 e seu valor novamente armazenado em acc0.

```
(1)   #define LENGTH 16
(2)
(3)   int convolution(int *px, int *ph)
(4)   {
(5)     int i, y = 0;
(6)
(7)     for (i = 0; i < LENGTH; i++)
(8)       y += *px++ * *ph--;
(9)
(10)    return y;
(11)  }
```

Figura 3.4: Convolução de dois sinais — código C.

```
(1)   _convolution:
(2)      loadi  acc1, 0x0          // y = 0
(3)      loadi  acc3, 0x0          // y = 0
(4)      loadi  acc0, 0x0          // i = 0
(5)      mov    p1, r30            // p1 = px
(6)      mov    p0, r31            // p0 = ph
(7)
(8)   L_4:
(9)      // modo dual das unidades de MAC; p0 e p1 -> 32 bits
(10)     ori    cntrl, cntrl, 0x8003
(11)
(12)     // MAC de px[j] e ph[k] e MAC de px[j+1] e ph[k-1]
(13)     mac    acc1, *p1++, *p0-- || mac    acc3, *p1++, *p0--
(14)
(15)     // modo simples das unidades de MAC; p0 e p1 -> 16 bits
(16)     andi   cntrl, cntrl, 0x7FFC
(17)
(18)     addi   acc0, acc0, 0x1   // i++
(19)     cmpi   acc0, 0x8         // i == 8 ?
(20)     jclr   #12, L_4          // se não, itera
(21)     noop
(22)
(23)     add    acc0, acc3, acc1  // redução de y
(24)
(25)     rsr    // retorna acc0
(26)     noop
```

Figura 3.5: Convolução de dois sinais — código *assembly* do DSP-V.

```
(1)   #define LENGTH 16
(2)
(3)   int average(int *ph)
(4)   {
(5)     int i, avg = 0;
(6)
(7)     for (i = 0; i < LENGTH; i++) {
(8)       avg += *(ph + i);
(9)     }
(10)
(11)    avg /= LENGTH;
(12)    return avg;
(13)  }
```

Figura 3.6: Média aritmética dos elementos de um *array* — código C.

```
(1)   _average:
(2)      loadi  r0, 0x0            // para somar a avg
(3)      loadi  acc0, 0x0          // avg = 0
(4)      loadi  acc2, 0x0          // avg = 0
(5)      loadi  acc1, 0x0          // i = 0
(6)      mov    p0, r31            // p0 = ph
(7)
(8)   L_1:
(9)      // modo dual das unidades de MAC; p0 -> 32 bits
(10)     ori    cntrl, cntrl, 0x8001
(11)
(12)     // soma de avg e ph[i] e soma de avg e ph[i+1]
(13)     add    acc0, acc0, *p0++ || add     acc2, acc0, *p0++
(14)
(15)     // modo simples das unidades de MAC; p0 -> 16 bits
(16)     andi   cntrl, cntrl, 0x7FFE
(17)
(18)     addi   acc1, acc1, 0x1   // i++
(19)     cmpi   acc1, 0x8         // i == 8 ?
(20)     jclr   #13, L_1          // se não, itera
(21)     noop
(22)
(23)     add    acc0, acc0, acc2  // redução de avg
(24)
(25)     asr    r1, acc0, 0x4     // avg / 16
(26)     cmpi   acc0, 0x0         // avg < 0 ?
(27)     jclr   #12, L_2          // se não, soma 0
(28)     noop
(29)     loadi  r0, 0x1           // se sim, soma 1
(30)
(31)  L_2:
(32)     add    acc0, r1, r0      // truncamento de avg
(33)     rsr                      // retorna acc0
(34)     noop
```

Figura 3.7: Média aritmética dos elementos de um *array* — código *assembly* do DSP-V.

Capítulo 4

Geração de MACs

4.1 Algoritmos de Análise de Fluxo de Dados

Nesta seção serão apresentados alguns conceitos e algoritmos nos quais se apóiam o nosso algoritmo para detecção e a geração das instruções de MAC. Uma referência completa sobre algoritmos de análise de fluxo de dados pode ser encontrada em [2], [4] e [36].

4.1.1 Grafo de Fluxo de Controle

Grafos de fluxo de controle são ferramentas bastante úteis na coleta de informações sobre a execução de um programa. Um grafo de fluxo de controle é um grafo dirigido cujos nodos representam computações (cada nodo do grafo contém um bloco básico do programa) e cujas arestas representam o fluxo de controle no programa. Existe uma aresta ligando o nodo `B` ao nodo `C` se `C` pode vir imediatamente em seguida de `B` em algum caminho de execução do programa. O grafo na Figura 4.1 é o grafo de fluxo de controle para o programa da Figura 2.9 na Seção 2.7, página 24.

4.1.2 Equações de Fluxo de Dados

A análise de fluxo de dados é um processo através do qual um compilador otimizante coleta informações acerca de um programa e as distribui para cada nodo do grafo de fluxo de controle. A coleta de informações pode ser feita montando e resolvendo um sistema de equações, em que as variáveis são conjuntos que contém informações sobre vários pontos do programa.

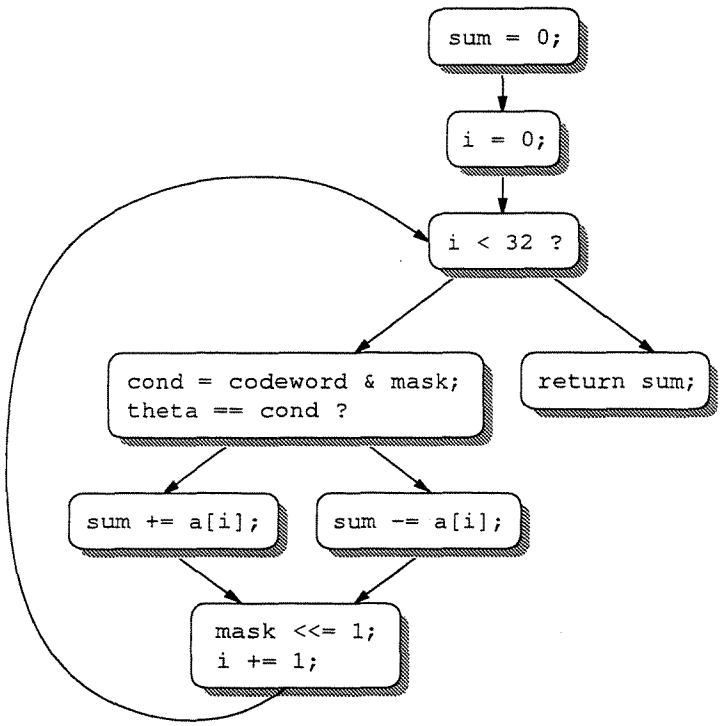


Figura 4.1: Grafo de fluxo de controle para o programa da Figura 2.9 na Seção 2.7.

Essas equações são chamadas de equações de fluxo de dados e um exemplo típico é a equação a seguir, que representa a informação ao final de um comando `S` (`out[S]`) como tendo sido gerada pelo comando (`gen[S]`) ou como tendo entrado em `S` (`in[S]`) sem ter sido destruída dentro dele (`kill[S]`):

$$out[S] = gen[S] \; U \; (in[S] - kill[S])$$

4.1.3 *Reaching Definitions/Use-Def Chains*

A definição `d` de uma variável (um comando que atribui um valor a essa variável) alcança um ponto `P` se existir algum caminho do ponto imediatamente posterior a `d` até `P` e `d` não for destruída no decorrer desse caminho. Uma definição de uma variável é destruída se entre dois pontos de um caminho houver uma atribuição a ela. A Figura 4.2 (a) ilustra esse conceito. A definição d_0 de `x` alcança o ponto P_1. Já a definição d_1 de y não alcança o ponto P_1, pois é destruída entre os pontos P_0 e P_1; entretanto, as definições d_2 e d_3 de y alcançam o ponto P_1.

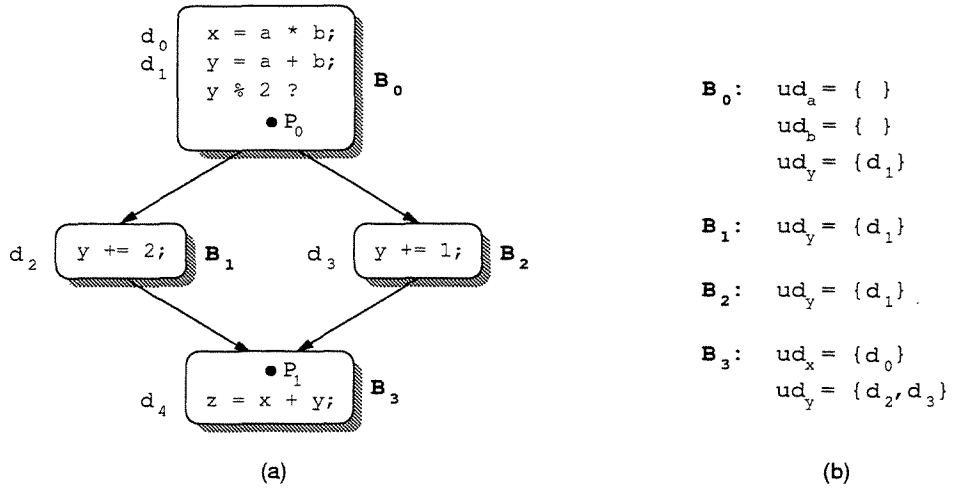


Figura 4.2: (a) *Reaching definitions.* (b) *UD-Chains.*

Use-Def Chains ou *UD-Chains* são listas, para cada uso de uma variável, de todas as definições que alcançam o ponto imediatamente anterior ao uso. As *UD-Chains* são uma maneira bastante conveniente de armazenar as informações de *reaching definitions*, e são utilizadas nos algoritmos de propagação de cópias, detecção de computações invariantes em laços, detecção de variáveis de indução e também no algoritmo de detecção de instruções de MAC, como será visto na Seção 4.2. A Figura 4.2 (b) mostra as *UD-chains* para o uso de cada uma das variáveis na Figura 4.2 (a).

4.1.4 *Def-Use Chains*

Em algumas situações pode ser útil determinar, dada a definição de uma variável, qual o seu próximo uso. É possível obter essa informação a partir das *ud-chains*: basta percorrer cada uma das listas, seguindo a ordem com que os usos aparecem no programa, até que a definição em questão seja encontrada. O uso correspondente será dado pelo comando contendo a variável em cuja *ud-chain* a definição foi encontrada. Essa abordagem não é direta, tampouco eficiente. Visando coletar essa informação de maneira eficiente, um procedimento inverso ao de *reaching definitions* é utilizado. Se imaginarmos *reaching definitions* como um algoritmo que percorre um programa de baixo para cima, a partir do ponto imediatamente anterior ao uso de uma variável, para encontrar as definições que alcançam esse ponto, então o seu inverso consiste em percorrer um programa a partir do ponto imediatamente posterior à definição de uma variável, para encontrar todos usos alcançados por essa definição. A Figura 4.3 (a) ilustra esse conceito.

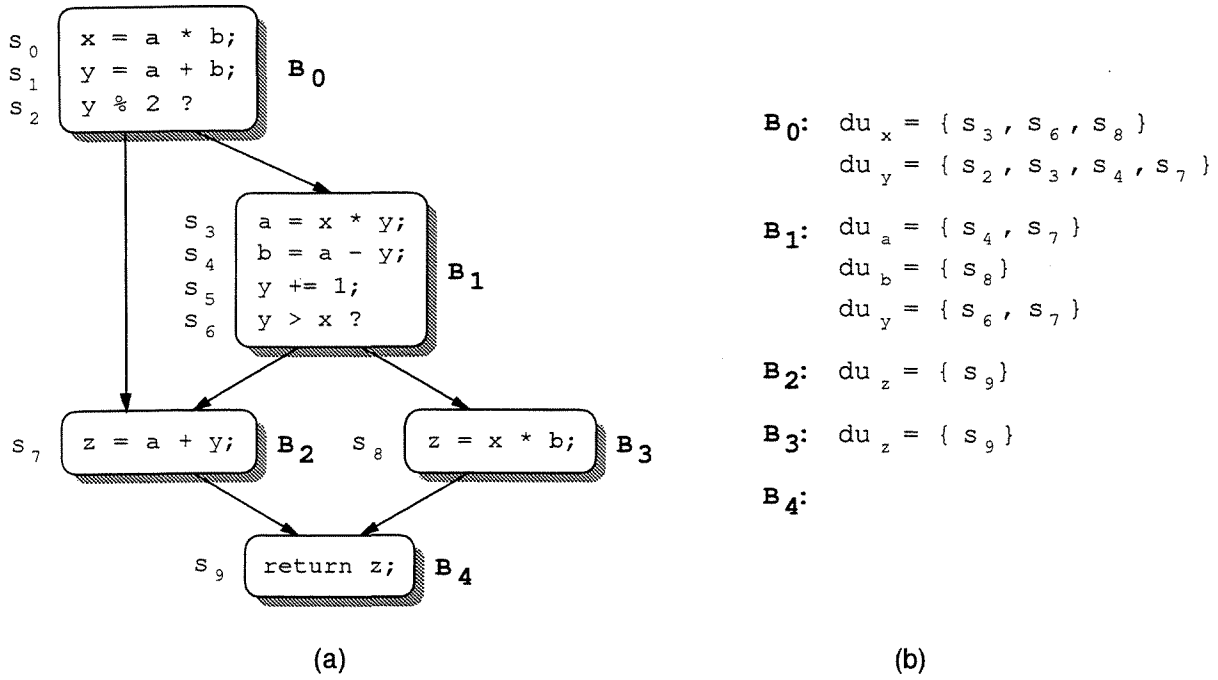


Figura 4.3: (a) Grafo de fluxo de controle. (b) *DU-Chains*.

Def-Use Chains ou *DU-Chains* são listas, para cada definição de uma variável em um programa, de todos os usos alcançados por esta, sem que haja uma redefinição da variável no caminho. As *DU-Chains* são utilizadas nos algoritmos de propagação de cópias e de detecção de instruções de MAC, como será visto na seção seguinte. A Figura 4.3 (b) mostra as *DU-chains* para cada uma das definições na Figura 4.3 (a).

4.1.5 Dominadores e Pós-Dominadores

Dominância e pós-dominância [4] e [36] são duas noções importantes para a análise de fluxo de dados em um programa, que são úteis no algoritmo para detecção de instruções de MAC. Dizemos que um bloco básico `A` domina um bloco básico `B`, em um grafo de fluxo de controle, se todo caminho de execução iniciado no primeiro bloco básico do programa e chegando a `B` passa, necessáriamente, por `A`. Por definição, todo bloco básico domina a si próprio. Dizemos que um bloco básico `A` pós-domina um bloco básico `B`, em um grafo de fluxo de controle, se todo caminho de execução que parte de `B` e chega ao último bloco básico do programa passa, necessáriamente, por `A`. Por definição, todo bloco básico pós-domina si próprio.

4.2 Implementação

As otimizações implementadas neste trabalho foram integradas a um compilador de produção de propriedade da *Conexant Systems Inc.* Esse compilador é um compilador Fixed-C [40] e [57] que realiza a maioria das otimizações descritas em [2], [4] e [36], como eliminação de subexpressões comuns, propagação de expressões constantes, eliminação de comandos de cópia, remoção de código invariante, eliminação de código inútil, etc. O compilador gera código para o processador *Countach40* [22], desenvolvido e fabricado pela mesma empresa. Esse processador é um DSP de características bastante semelhantes à arquitetura DSP-V apresentada no Capítulo 3 e é utilizado no projeto de modems, placas de vídeo e de áudio, GPS, além de outros dispositivos portáteis.

A geração de instruções de MAC foi implementada em duas etapas:

1. Detecção do padrão de multiplicação e acumulação a nível de representação intermediária do programa fonte e criação de um novo tipo de operação na representação intermediária que encapsula essas duas operações sempre que o padrão for encontrado.

2. Reconhecimento do nodo de MAC a nível de gerador de código (*back-end* do compilador) e geração da instrução de MAC do processador sempre que esse nodo for encontrado.

Essas etapas são descritas em maiores detalhes a seguir.

4.2.1 Representação Intermediária

Para detectar o padrão de multiplicação e acumulação associado com instruções do tipo `t := x * h` e `z := z + t` são utilizadas as informações armazenadas pelas listas *ud-chains* e *du-chains*, de usos e definições das variáveis. A Figura 4.4 traz um algoritmo em alto-nível para a detecção de MACs na representação intermediária do programa fonte. Ao encontrar uma instrução de multiplicação (linha 5), a lista *du-chain* da variável definida pela instrução (`t`) é examinada. Nesse momento duas condições devem ser verdadeiras: essa lista deve ser unitária (linha 5) e o comando em que a variável é usada deve ser uma operação de soma (linha 7). É fácil compreender a segunda condição, já que estão sendo procurados padrões de multiplicação e acumulação. O motivo pelo qual a primeira condição existe é bem simples, embora não tão óbvio: como uma única instrução será gerada em substituição às duas anteriores, o valor do temporário `t` criado pela instrução de multiplicação deixará de existir; portanto, se este estiver sendo usado em qualquer

outra instrução que não a de soma, as instruções de multiplicação e acumulação não poderão ser transformadas em uma instrução de MAC. Uma situação em que há violação dessa condição está sendo mostrada na Figura 4.5 (a). Nesse exemplo, a existência do comando `k := z * t` impede a geração de uma instrução de MAC, já que isso deixaria a variável `t` indefinida.

```
(1)   MAC_detect(control_flow_graph CFG)
(2)   for each block B in CFG
(3)     for each statement S in B
(4)       if S is multiply and |S.du_chains| = 1
(5)         A = extract S.du_chains
(6)         if A is add and A.def ⊂ A.use and |A.ud_chains| = 1
(7)           if A.block in dom(S) and S.block in pdom(A)
(8)             create_node(S, mac)
(9)             remove_node(temp_decl)
(10)            remove_node(A)
(11)          end if
(12)        end if
(13)      end if
(14)    end for each
(15)  end for each
```

Figura 4.4: Algoritmo para a detecção das instruções de MAC.

Se as duas condições forem verdadeiras, como no exemplo da Figura 4.6 (a), então a inspeção prossegue com a instrução de soma. Aqui, mais uma vez, duas condições se aplicam: a lista *ud-chains* da variável definida pela instrução de multiplicação (`t`) deve ser unitária (linha 7) e a variável definida pela instrução de soma (`z`) deve ser também um de seus operandos (linha 7). A segunda condição é justamente a característica de uma instrução de acumulação: somar um valor a uma variável e armazenar o resultado da computação nela própria. Já a primeira condição é necessária para garantir que nenhuma instrução será marcada como código inútil e removida, como conseqüência da remoção da instrução que faz uso da variável que define (`t`); assim, se esta estiver sendo definida em qualquer outra instrução que não a de multiplicação, as instruções de multiplicação e acumulação não podem ser transformadas em uma instrução de MAC. Uma situação em que há violação dessa condição está sendo mostrada na Figura 4.5 (b). Nesse exemplo, a existência da definição `t := a + b` de `t` impede a geração de uma instrução de MAC, pois isso removeria o comando `z := z + t`, que usa a variável `t`, e, conseqüentemente, o comando `t := a + b`, que seria considerado inútil.

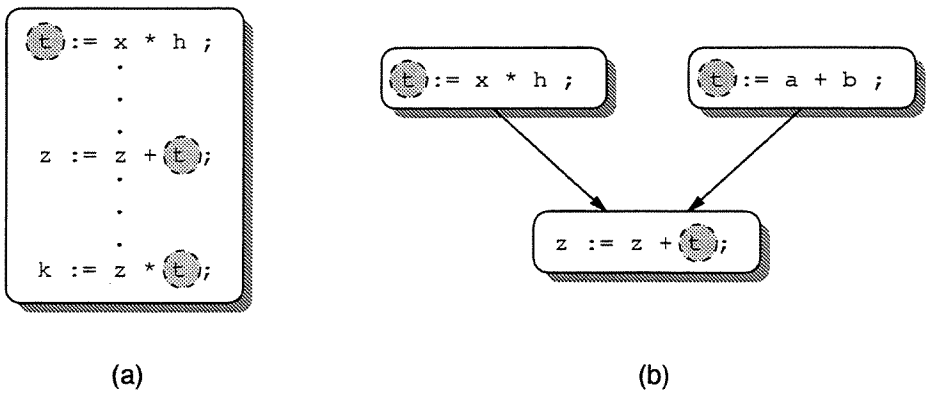


Figura 4.5: (a) Violação da condição de que a lista *du-chains* de `t` na instrução de multiplicação deve ser unitária. (b) Violação da condição de que a lista *ud-chains* de `t` na instrução de soma deve ser unitária.

Novamente, se as duas condições forem verdadeiras, como no exemplo da Figura 4.6 (b), o algoritmo de geração da instrução de MAC prossegue; entretanto, um outro cuidado deve ainda ser tomado a fim de garantir a correção do código gerado. Como as informações contidas nas listas *ud-chains* e *du-chains* são globais, isto é, cruzam as fronteiras entre os blocos básicos, é possível, embora não muito freqüente, de acordo com experimentos realizados, que as instruções de multiplicação e acumulação estejam em blocos básicos diferentes. Por essa razão, as informações de dominância e pós-dominância devem ser também consultadas. Para que a instrução de MAC possa ser gerada, mais duas condições devem ser satisfeitas: o bloco básico contendo a instrução de multiplicação deve dominar o bloco básico contendo a instrução de soma (linha 8) e o bloco básico contendo a instrução de soma deve pós-dominar o bloco básico contendo a instrução de multiplicação (linha 8). Essas duas condições garantem, respectivamente, que se a instrução de soma foi executada, a de multiplicação também terá sido, e que se a instrução de multiplicação for executada, a de soma também o será.

Se todas as condições tiverem sido satisfeitas, então a instrução de adição é removida e a de multiplicação é finalmente transformada em uma instrução de multiplicação com acumulação (instrução de MAC). Essa instrução possui três operandos: os dois operandos da multiplicação (`x` e `h`) e a variável em que está sendo armazenada a soma de todos os produtos (`z`): `z += x * h` ou `mac z, x, h`.

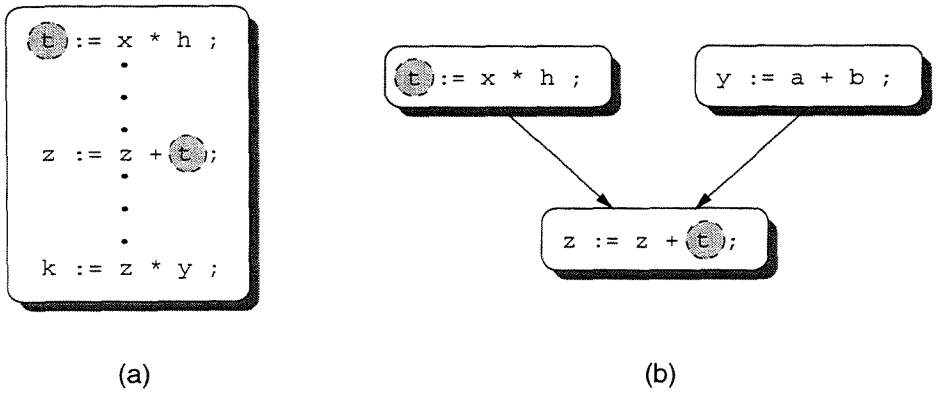


Figura 4.6: (a) Condição de que a lista *du-chains* de t na instrução de multiplicação deve ser unitária é satisfeita. (b) Condição de que a lista *ud-chains* de t na instrução de soma deve ser unitária é satisfeita.

4.2.2 Gerador de Código

O gerador de código do compilador utilizado nos experimentos é sintetizado a partir de uma gramática semelhante à apresentada em [19]. O nodo de MAC foi reconhecido por meio da introdução de uma nova regra na gramática e a instrução de MAC foi gerada pelas ações semânticas correspondentes. Ao encontrar um nodo de MAC o gerador de código emite as instruções necessárias e sintetiza um atributo, que é o resultado da operação de MAC.

A avaliação de expressões constantes foi realizada em tempo de compilação, para evitar computações desnecessárias. A operação para a qual o código está sendo gerado, `c = c + a * b` (`MAC c, a, b`), possui três operandos, dois dos quais (`a` e `b`) podem ser constantes. Foram tratadas quatro possibilidades, portanto:

1. `a` e `b` são ambos constantes: o produto `const = a * b` é efetuado e código é gerado para realizar a operação `c = c + const`.

2. `a` (ou `b`) é constante e igual a 1: código é gerado para realizar a operação `c = c + a` (ou `c = c + b`).

3. `a` (ou `b`) é constante e diferente de 1: uma instrução de MAC é gerada para realizar a operação `c += a * const` (ou `c += b * const`).

4. Nenhum operando é constante: uma instrução de MAC é gerada para realizar a operação `c += a * b`.

4.3 Exemplo

A seguir é apresentado um exemplo de programa em que fica fácil observar o ganho obtido com a geração de uma instrução de MAC. O programa compilado é uma função que realiza o produto interno entre dois vetores passados como parâmetro, uma aplicação típica em DSPs. A Figura 4.7 mostra o código fonte, em C, dessa função.

```
(1)   #define LENGTH 16
(2)
(3)   int dot_prod (int *px, int *ph)
(4)   {
(5)     int i;
(6)     int z = 0;
(7)
(8)     for (i = 0; i < LENGTH; i++)
(9)       z += *px++ * *ph++;
(10)
(11)    return z;
(12)  }
```

Figura 4.7: `dot_prod.c` — código fonte.

O código gerado pela versão original do compilador (Figura 4.8) realiza a multiplicação e a acumulação de `px` e `ph` em `z` em duas instruções separadas (linha 9) e (linha 12). Entretanto, mais uma instrução é necessária (linha 13) para salvar o valor de `acc0`, já que este não é mantido vivo durante todas as iterações do laço. Ao final da execução do laço é ainda necessário armazenar o valor final de `z` em `acc0` (linha 20), que, por convenção do *assembly*, é onde deve ser armazenado o retorno da função.

Na Figura 4.9 pode ser visto o código otimizado com instrução de MAC (linha 9) para o programa `dot_prod.c`. O laço possui 2 instruções a menos que o do programa original, o que já representa uma diferença de 32 ciclos de máquina na execução de 16 iterações. Ao final do laço, torna-se desnecessário mover o resultado da acumulação para `acc0`, uma vez que este já se encontra lá.

O programa `dot_prod.c` compilado com a versão do compilador sem a otimização executa em $4 + (7 * 16) + 3 = 119$ ciclos e possui 14 instruções. Quando compilada com a versão que realiza geração de MACs, executa em $4 + (5 * 16) + 2 = 86$ ciclos e possui 11 instruções. Isso significa um *speedup* de $27,73\%$ no tempo de execução do programa e uma redução de $21,43\%$ no tamanho do código gerado.

```
(1)    _dot_prod:
(2)       loadi  r2, 0x0          // z = 0
(3)       loadi  acc1, 0x0        // i = 0
(4)       mov    p1, r30          // p1 = px
(5)       mov    p0, r31          // p0 = ph
(6)
(7)    L_4:
(8)       // multiplicação de *px e *ph
(9)       mul    acc0, *p1++, *p0++
(10)
(11)      // acumulação em z
(12)      add    acc0, acc0, r2
(13)      mov    r2, acc0L
(14)
(15)      addi   acc1, acc1, 0x1  // i++
(16)      cmpi   acc1, 0x20       // i == 16 ?
(17)      jclr   #13, L_4         // se não, itera
(18)      noop
(19)
(20)      mov    acc0, r2
(21)      rsr                     // retorna acc0
(22)      noop
```

Figura 4.8: `dot_prod.c` — código *assembly* do DSP-V sem a otimização.

```
(1)   _dot_prod:
(2)      loadi  acc0, 0x0          // z = 0
(3)      loadi  acc1, 0x0          // i = 0
(4)      mov    p0, r31            // p0 = ph
(5)      mov    p1, r30            // p1 = px
(6)
(7)   L_1:
(8)      // multiplicação e acumulaçao de px e ph em z
(9)      mac    acc0, *p1++, *p0++
(10)
(11)     addi   acc1, acc1, 0x1    // i++
(12)     cmpi   acc1, 0x10         // i == 16 ?
(13)     jclr   #13, L_1           // se não, itera
(14)     noop
(15)
(16)     rsr                       // retorna acc0
(17)     noop
```

Figura 4.9: `dot_prod.c` — código *assembly* do DSP-V otimizado com MACs.

4.4 Resultados Experimentais

Como os DSPs são utilizados para execução de aplicações bastante específicas e que geralmente são propriedade intelectual de grandes empresas, o conjunto de *benchmarks* de domínio público disponível não é muito vasto. Os *benchmarks* adotados para avaliar o sucesso deste trabalho incluem *kernels* de alguns programas utilizados pela *Conexant Systems Inc.* em seus produtos, alguns programas da *Texas Instruments* e vários programas pertinentes ao mais conhecido *benchmark* para DSPs, o DSPStone [58]. Ele implementa os núcleos de diversas aplicações que executam nos DSPs, e por isso foi considerado o método ideal de avaliação das técnicas aqui apresentadas.

Os resultados foram analisados considerando os fatores tempo de execução e tamanho do código gerado, em comparação com os resultados apresentados pela compilação e execução dos mesmos programas utilizando a versão original do compilador, antes da integração das otimizações das instruções de MAC. As medidas de número de ciclos foram realizadas utilizando um simulador do processador *Countach40*.

A Tabela 4.1 apresenta a contagem de ciclos de máquina para a execução dos programas listados na coluna mais à esquerda, quando compilados com as versões original e otimizada com geração de MACs do compilador. A observação desses valores mostra um *speedup* médio de $4,61\%$ para programas compilados utilizando a geração de MACs, um ganho considerado bastante razoável. A grande variação apresentada pelo *speedup* ($0,11\% - 25,84\%$) é decorrente da estrutura dos programas fonte. Programas como `v34.c` e `madd01.c` apresentaram ganhos percentuais pequenos, por não apresentarem laços. Programas como `dot_prod.c` e `dot_product_array.c`, apesar de apresentarem laços, esses executam poucas iterações, e o ganho obtido com a geração das instruções de MAC não fica tão evidente. Já programas como `matrix1.c` e `matrix2.c`, apresentam laços aninhados que executam diversas iterações, deixando bem explícito o ganho obtido com a otimização.

A Tabela 4.2 apresenta o tamanho do código gerado, em número de instruções, para os programas listados na coluna mais à esquerda, quando compilados com as versões original e otimizada com geração de MACs do compilador. A observação desses valores mostra uma redução média de $4,09\%$ no tamanho do código para os programas compilados utilizando a geração de MACs. A redução geralmente oscila entre 2 e 5 instruções, mas pode ser bem maior. Isso foi observado nos programas `kernels.c`, `index.c` e `fir.c`, em que a geração de uma instrução de MAC, e, conseqüentemente a remoção de uma instrução de multiplicação que armazena seu resultado em um temporário, eliminou diversas instruções de *spilling* [13] de variáveis.

	Original	Otimizado	
Programa	Ciclos	Ciclos	*Speedup*
dot_prod.c	816	810	0,74 %
convolut.c	1139	1093	4,04 %
kernels.c	7299	6929	5,07 %
lms.c	2406	2342	2,66 %
loop5.c	1754	1722	1,82 %
matrix1.c	14705	10905	25,84 %
v34.c	1810	1808	0,11 %
biquad_N_sections.c	1452	1444	0,56 %
biquad_one_section.c	835	833	0,24 %
convolution_array.c	1112	1066	4,14 %
dot_product_array.c	809	804	0,62 %
fir2dim.c	3899	3419	12,31 %
fir_array.c	1293	1277	1,24 %
fir.c	1392	1376	1,15 %
index.c	861	833	3,25 %
int01save.c	20372	19972	1,96 %
madd01.c	797	796	0,13 %
mat1x3.c	954	924	3,14 %
lms_array.c	1634	1555	4,83 %
mat1x3_array.c	1006	985	2,09 %
matrix2.c	15582	11782	24,39 %
n_complex_updates.c	2791	2759	1,15 %
Média	–	–	4,61 %

Tabela 4.1: Desempenho: resultado após a geração de MACs.

	Original	Otimizado	
Programa	Instruções	Instruções	Redução
v34	101	99	1,98 %
biquad_n_sections	205	204	0,49 %
dot_product_array	46	45	2,17 %
matrix2	121	119	1,65 %
n_complex_updates	199	195	2,01 %
madd01	44	43	2,27 %
matrix1	113	112	0,89 %
mat1x3_array	63	57	9,52 %
mat1x3	54	51	5,56 %
loop5	128	126	1,56 %
lms_array	189	179	5,29 %
lms	226	224	0,89 %
kernels	1359	1241	8,68 %
int01save	83	82	1,21 %
index	42	34	19,05 %
fir2dim	227	221	2,64 %
fir_array	99	98	1,01 %
fir	110	95	13,63 %
biquad_one_section	94	92	2,13 %
dot_prod	50	48	4,00 %
convolution_array	54	53	1,85 %
convolut	65	64	1,54 %
Média	–	–	4,09 %

Tabela 4.2: Tamanho de código: resultado após a geração de MACs.

Capítulo 5

Vetorização de MACs

5.1 Geração de Código para Arquiteturas Vetoriais

Nesta seção serão apresentados dois conceitos importantes nos quais se apóia a vetorização das instruções de MAC. Uma referência completa sobre os conceitos e algoritmos envolvidos na geração de código para arquiteturas SIMD e vetoriais e sobre transformações em laços para explorar o paralelismo de dados nessas arquiteturas pode ser encontrada em [9] e [56].

5.1.1 *Strip-Mining*

Strip-mining [56] é uma técnica de restruturação de laços geralmente aplicada a nível do código fonte ou representação intermediária do programa, com a finalidade de ajustar a granularidade de uma operação paralelizável. O *strip-mining* favorece a posterior geração de código que executa em paralelo em arquiteturas SIMD e vetoriais. A transformação consiste em decompor um laço em dois: um mais externo, que percorre "faixas" de iterações consecutivas e um mais interno, que itera sobre os elementos de cada "faixa".

Se o número de iterações do laço original não for um múltiplo inteiro do número de unidades funcionais da arquitetura alvo, as iterações restantes são adicionadas ao final do laço transformado por *strip-mining*. Um exemplo dessa situação para uma arquitetura vetorial com 6 unidades funcionais e um laço que executa 32 iterações pode ser visto na Figura 5.1. A Figura 5.1 (b) mostra o código *strip-mined* para o laço da Figura 5.1 (a).

O laço na Figura 5.1 (a) armazena no *array* a a soma dos elementos dos *arrays* b e c. Ele executa 32 iterações; portanto, só não precisa ser restruturado se o código estiver sendo gerado para uma arquitetura vetorial com 32 unidades funcionais. Como nossa

```
(1)  for (i=0; i<32; i++) {
(2)    a[i] = b[i] + c[i];
(3)  }
```

(a)

```
(1)  for (i=0; i<30; i=i+6) {
(2)    for (j=i; j<i+6 && j<30; j++) {
(3)      a[j] = b[j] + c[j];
(4)    }
(5)  }
(6)  for (i=30; i<32; i++) {
(7)    a[i] = b[i] + c[i];
(8)  }
```

(b)

Figura 5.1: (a) Laço antes da realização de *strip-mining*. (b) Laço após a realização de *strip-mining*.

arquitetura alvo neste exemplo tem 6 unidades funcionais, o laço deve sofrer *strip-mining*. A Figura 5.1 (b) reflete essa transformação. O laço mais externo (linha 1) executa 5 iterações (o resultado da divisão inteira de 32 por 6) sobre as faixas de 6 elementos (número de unidades funcionais da arquitetura) percorridas no laço mais interno (linha 2). Esses dois laços executam 30 iterações das 32 do código original, restando ainda duas iterações (o resto da divisão inteira de 32 por 6). Essas duas iterações são executadas pelo laço adicional (linha 6). O código após o *strip-mining* corresponde integralmente ao código original, mas agora pode ser vetorizado. O laço mais interno (linha 2) pode ser substituído por uma única instrução vetorial da arquitetura alvo, resultando em um código que executa apenas 7 iterações em vez de 32. Entretanto, quando o número de iterações do laço original não for um múltiplo inteiro do número de unidades funcionais da arquitetura alvo, o código gerado será um pouco maior que o original.

5.1.2 Redução para Escalar

Sempre que resultados parciais forem acumulados em um registrador vetorial, como foi o caso para os programas `average.c` e `convolut.c` apresentados nos exemplos da Seção 3.6, o registrador vetorial deve ser reduzido a um escalar após a execução do laço. Essa operação é chamada de redução para escalar, e pode ser realizada por alguma instrução especializada presente no conjunto de instruções da arquitetura, por um laço que soma os elementos um a um ou por somas parciais utilizando registradores vetoriais menores. A Figura 5.2 mostra um trecho de código *assembly* do DSP-V em que a redução para escalar (linha 10) é feita após a execução de um laço.

```
(1)    loop:
(2)       ori    cntrl, cntrl, 0x8003
(3)       mac    acc0, *p1++, *p0++ || mac    acc2, *p1++, *p0++
(4)       andi   cntrl, cntrl, 0x7FFC
(5)       addi   acc1, acc1, 0x1
(6)       cmpi   acc1, 0x8
(7)       jclr   #13, loop
(8)       noop
(9)
(10)      add    acc0, acc0, acc2  // redução
(11)
(12)      rsr
(13)      noop
```

Figura 5.2: Código *assembly* do DSP-V mostrando a redução para escalar no acumulador `acc0`.

5.2 Implementação

A vetorização das instruções de MAC foi implementada em duas etapas:

1. Detecção de laços e coleta de informações sobre eles, tais como: *header*, *pre-header*, variável de indução, conjunto de blocos, condição de saída, valor inicial da variável de indução, valor final da variável de indução, incremento e número de iterações [2]. Isso foi feito a nível da representação intermediária do programa fonte (durante a fase de otimização do compilador), assim como a vetorização dos laços contendo apenas uma instrução de MAC em seu interior. Foi criado um novo tipo de operação na representação intermediária, chamado VMAC (*Vector MAC*) que encapsula a operação vetorial de MAC e substitui os nodos de MAC sempre que os laços forem paralelizados.

2. Reconhecimento da operação VMAC pelo gerador de código (*back-end* do compilador) e geração da instrução vetorial de MAC do processador sempre que esse nodo for encontrado.

Essas etapas são descritas em maiores detalhes a seguir.

5.2.1 Representação Intermediária

A detecção de laços e a coleta das informações necessárias à paralelização dos mesmos foram realizadas conforme os algoritmos apresentados em [2] e poderão também ser utilizadas em otimizações sobre laços que ainda venham a ser implementadas, como as apresentadas em [56].

Antes de prosseguir com a geração de instruções vetoriais de MAC, é necessário verificar se o laço sendo analisado possui apenas uma instrução de MAC, além, é claro, das instruções de atualização e teste da variável de indução. Essa condição é necessária neste momento, pois a vetorização do código está limitada apenas às instruções de MAC (por enquanto; o Capítulo 6 sugere que seja estendida também para outras instruções). Se ela for satisfeita, então o laço passa pelo procedimento de *strip-mining*, como apresentado na Seção 5.1.1. Para que seja possível realizar essa restruturação do laço, devem ser conhecidas quatro informações: o valor inicial da variável de indução, o valor final da variável de indução, o incremento do laço e o número de iterações do mesmo. Se alguma dessas informações não for conhecida, o processo de *strip-mining* torna-se inviável e a otimização não pode prosseguir.

Após a realização de *strip-mining* sobre o laço, o nodo de MAC é substituído por um de VMAC e a seqüência de árvores de expressões está pronta para ser submetida ao gerador de código.

5.2.2 Gerador de Código

A maior parte do esforço de compilação ao realizar a vetorização das instruções de MAC é feita a nível da representação intermediária. Durante a fase de geração de código o nodo da instrução vetorial VMAC é reconhecido por meio da introdução de uma nova regra na gramática. A instrução vetorial de MAC é gerada pelas ações semânticas correspondentes. Ao encontrar um nodo VMAC o gerador de código emite as instruções necessárias e sintetiza um atributo, que é o resultado da operação de MAC. Como mostram os dois exemplos a seguir, o código gerado é diferente para os casos em que a acumulação está sendo armazenada em um escalar ou em posições contíguas de um *array*.

5.3 Exemplos

5.3.1 Acumulação em um Escalar

Na Figura 5.4 pode ser visto o código otimizado com instrução vetorial de MAC (linha 13) para o mesmo programa (`dot_prod.c`) apresentado na Seção 4.3. Seu código fonte está replicado na Figura 5.3, nesta seção. O laço possui o mesmo número de instruções que o laço original sem a otimização, 2 instruções a mais (linha 10) e (linha 16) que a versão com instrução de MAC simples. Entretando ele executa apenas metade das iterações (linha 19), compensando o *overhead* introduzido pelas instruções de ajuste do modo de operação das unidades de MAC e controle de acesso à memória (linha 10) e (linha 16). Ao final do laço é feita a redução de `z` a escalar em `acc0` (linha 23).

```
(1)   #define LENGTH 16
(2)
(3)   int dot_prod (int *px, int *ph)
(4)   {
(5)     int i;
(6)     int z = 0;
(7)
(8)     for (i = 0; i < LENGTH; i++)
(9)       z += *px++ * *ph++;
(10)
(11)    return z;
(12)  }
```

Figura 5.3: `dot_prod.c` — código fonte.

O programa `dot_prod.c` compilado com a versão do compilador sem a otimização executa em $4 + (7 * 16) + 3 = 119$ ciclos e possui 14 instruções. Quando compilada com a versão que realiza geração de instruções vetoriais de MAC, executa em $5 + (7 * 8) + 3 = 64$ ciclos e possui 15 instruções. Isso significa um *speedup* de $46,22\%$ no tempo de execução do programa ao preço de um aumento de $7,14\%$ no tamanho do código gerado.

```
(1)   _dot_prod:
(2)      loadi  acc0, 0x0          // z = 0
(3)      loadi  acc2, 0x0          // z = 0
(4)      loadi  acc1, 0x0          // i = 0
(5)      mov    p0, r31            // p0 = ph
(6)      mov    p1, r30            // p1 = px
(7)
(8)   L_1:
(9)      // modo dual das unidades de MAC; p0 e p1 -> 32 bits
(10)     ori    cntrl, cntrl, 0x8003
(11)
(12)     // multiplicação e acumulaçao de px e ph em z
(13)     mac    acc0, *p1++, *p0++ || mac    acc2, *p1++, *p0++
(14)
(15)     // modo simples das unidades de MAC; p0 e p1 -> 16 bits
(16)     andi   cntrl, cntrl, 0x7FFC
(17)
(18)     addi   acc1, acc1, 0x1   // i++
(19)     cmpi   acc1, 0x8         // i == 8 ?
(20)     jclr   #13, L_1          // se não, itera
(21)     noop
(22)
(23)     add    acc0, acc0, acc2  // redução de z
(24)
(25)     rsr                      // retorna acc0
(26)     noop
```

Figura 5.4: `dot_prod.c` — código *assembly* do DSP-V otimizado com instruções vetoriais de MAC.

5.3.2 Acumulação em Posições Contíguas de um *Array*

A seguir é apresentado um exemplo de programa em que a acumulação é feita não em um escalar, como nos exemplos anteriores, mas em posições contíguas de um *array*. Nesse caso não é mais necessário realizar a redução a escalar ao final do laço, mas os resultados de cada operação de MAC devem ser armazenados na posição correspondente do *array* de destino logo após a realização da operação. O programa compilado é uma função (`vec_mpy`, extraída de `kernels.c`) que realiza o produto de cada elemento de um vetor (`x`) por um fator escalar (`scaler`), acumulando com o elemento de índice `i` de um outro vetor (`y`). A Figura 5.5 traz o código fonte, em `C`, dessa função.

O código gerado pela versão original do compilador, que está sendo mostrado na Figura 5.6 realiza a multiplicação e a acumulação de `x` e `scaler` em `y[i]` em duas instruções separadas (linha 8) e (linha 11). Mais uma instrução é necessária (linha 12) para armazenar em `y[i]` o resultado dessas operações. A função altera o parâmetro `y` e retorna.

Na Figura 5.7 pode ser visto o código otimizado com instrução vetorial de MAC (linha 15) para o programa `vec_mpy.c`. O laço possui 4 instruções a mais que o laço original sem a otimização. Entretando ele executa apenas metade das iterações (linha 25), compensando o *overhead* introduzido pelas instruções de ajuste do modo de operação das unidades de MAC e controle de acesso à memória (linha 8) e (linha 18) e das instruções que carregam os valores iniciais nos acumuladores (linha 11) e (linha 12). Quando a acumulação é feita em posições contíguas de um *array* não há necessidade de redução ao final do laço, mas a cada iteração os resultados devem ser escritos nas posições correspondentes do *array* (linha 21) e (linha 22).

O programa `vec_mpy.c` compilado com a versão do compilador sem a otimização executa em $3 + (7 * 16) + 2 = 117$ ciclos e possui 12 instruções. Quando compilado com a versão que realiza geração e vetorização de MACs, executa em $3 + (11 * 8) + 2 = 93$ ciclos e possui 16 instruções. Isso significa um *speedup* de $20,51\%$ no tempo de execução do programa ao preço de um aumento de $33,33\%$ no tamanho do código gerado.

Os ganhos obtidos com a vetorização de MACs para os casos em que a acumulação é feita em posições contíguas de um *array* devem ser cuidadosamente ponderados, principalmente quando houver restrições quanto ao tamanho do código gerado. O *speedup* não é tão grande se comparado com os casos em que a acumulação é feita em um escalar. O fato de serem necessárias instruções de inicialização dos acumuladores e de ajuste de modo de operação e acesso à memória contribui para que o código fique maior que o gerado sem a vetorização de MACs.

```
(1)   #define LENGTH 16
(2)
(3)   void vec_mpy(int y[], const int x[], int scaler)
(4)   {
(5)     int i;
(6)     int *py = y;
(7)     int *px = x;
(8)
(9)     for (i = 0; i < LENGTH; i++)
(10)      *py++ += scaler * *px++;
(11)
(12)    return z;
(13)  }
```

Figura 5.5: `vec_mpy.c` — código fonte.

```
(1)   _vec_mpy:
(2)      mov    p0, r29          // p0 = y
(3)      mov    p1, r30          // p1 = x
(4)      loadi  acc1, 0x0        // i = 0
(5)
(6)   L_4:
(7)      // multiplicação de x e scaler
(8)      mul    acc0, *p1++, r31
(9)
(10)     // acumulação em y[i]
(11)     add    acc0, acc0, *p0
(12)     mov    *p0++, acc0
(13)
(14)     addi   acc1, acc1, 0x1  // i++
(15)     cmpi   acc1, 0x10       // i == 16 ?
(16)     jclr   #13, L_4         // se não, itera
(17)     noop
(18)
(19)     rsr
(20)     noop
```

Figura 5.6: `vec_mpy.c` — código *assembly* do DSP-V sem a otimização.

```
(1)   _vec_mpy:
(2)      mov    p0, r29          // p0 = y
(3)      mov    p1, r30          // p1 = x
(4)      loadi  acc1, 0x0        // i = 0
(5)
(6)   L_4:
(7)      // modo dual das unidades de MAC; p1 -> 32 bits
(8)      ori    cntrl, cntrl, 0x8002
(9)
(10)     // carrega valores iniciais nos acumuladores
(11)     loadi  acc0, p0++       // acc0 = y[i]
(12)     loadi  acc2, p0--       // acc2 = y[i+1]
(13)
(14)     // multiplicação de x e scaler
(15)     mac    acc0, *p1++, r31 || mac      acc2, *p1++, r31
(16)
(17)     // modo simples das unidades de MAC; p1 -> 16 bits
(18)     andi   cntrl, cntrl, 0x7FFD
(19)
(20)     // armazena resultado em y
(21)     mov    *p0++, acc0      // y[i] = acc0
(22)     mov    *p0++, acc2      // y[i+1] = acc2
(23)
(24)     addi   acc1, acc1, 0x1  // i++
(25)     cmpi   acc1, 0x8        // i == 8 ?
(26)     jclr   #13, L_4         // se não, itera
(27)     noop
(28)
(29)     rsr
(30)     noop
```

Figura 5.7: `vec_mpy.c` — código *assembly* do DSP-V otimizado com instruções vetoriais de MAC.

5.4 Resultados Experimentais

A Tabela 5.1 apresenta a contagem de ciclos de máquina para a execução dos programas listados na coluna mais à esquerda, quando compilados com as versões original e otimizada com geração e vetorização de MACs do compilador. A observação desses valores mostra um *speedup* médio de 14,89% para programas compilados utilizando a vetorização de MACs, um ganho considerável. A grande variação apresentada pelo *speedup* (4,07% – 30,60%) é decorrente da estrutura dos programas fonte. Programas como `index.c` e `matlx3.c` apresentaram ganhos percentuais pequenos, por apresentarem laços com poucas iterações, além de acumularem em *arrays*, o que, como foi visto na Seção 5.3.2, contribui para o *overhead* da vetorização de MACs. Programas como `kernels.c` e `lms.c`, são bastante grandes e possuem diversas funções, muitas das quais não apresentam instruções de MAC, de tal forma que o ganho obtido com a vetorização das instruções de MAC não fica tão evidente. Já programas como `matrix1.c` e `matrix2.c`, apresentam laços aninhados que executam diversas iterações, deixando bem explícito o ganho obtido com a otimização.

A Tabela 5.2 apresenta o tamanho do código gerado, em número de instruções, para os programas listados na coluna mais à esquerda, quando compilados com as versões original e otimizada com geração e vetorização de MACs do compilador. A observação desses valores mostra um aumento de, em média, 7,83% no tamanho do código para os programas compilados utilizando a vetorização de MACs. O aumento geralmente oscila entre 6 e 8 instruções, mas pode ser bem maior. Isso foi observado nos programas `fir2dim.c` e `lms.c`, em que a vetorização das instruções de MAC revelou uma falha no algoritmo de alocação de registradores utilizado pelo compilador, ocasionando uma explosão no tamanho do código. Essa falha foi corrigida e novos testes não deverão produzir um aumento tão acentuado no tamanho do código [1]. Já os programas `kernels.c` e `index.c` apresentaram redução no tamanho do código devido à presença de funções em que a vetorização de MACs não foi possível. Isso ocorre quando as instruções de MAC não estão sozinhas no interior dos laços ou quando a realização de *strip-mining* não foi possível, por falta de alguma das informações necessárias (Seção 5.2.1). Nestes casos, a redução no tamanho do código devido á geração de MACs superou o *overhead* causado pela vetorização das instruções de MAC.

[1] Até o momento da escrita desta dissertação não tivemos acesso á nova versão do compilador com a correção no algoritmo de alocação de registradores, implementada por uma equipe de engenheiros da *Conexant Systems Inc.*

A Tabela 5.3 apresenta a contagem de ciclos de máquina para a execução dos laços dos programas listados na coluna mais à esquerda, quando compilados com as versões original e otimizada com geração e vetorização de MACs do compilador. A observação desses valores mostra um *speedup* médio de $21,23\%$ para os laços dos programas compilados utilizando a vetorização de MACs, um ganho considerável. Essa abordagem para a visualização do *speedup* resultante das duas otimizações leva em consideração apenas os laços em que a geração de MACs foi realizada, seguida ou não de vetorização. Aqui a grande variação apresentada pelo *speedup* ($1,69\% - 46,85\%$) é decorrente unicamente da estrutura dos programas fonte. Programas como `biquad_N_sections.c` e `int01save.c` apresentam laços que executam poucas iterações, e o ganho obtido com a geração/vetorização das instruções de MAC não fica tão evidente. Já programas como `lms.c`, `matrix1.c` e `matrix2.c`, apresentam laços aninhados que executam diversas iterações, deixando bem explícitos os ganhos obtidos com as otimizações.

	Original	Otimizado	
Programa	Ciclos	Ciclos	*Speedup*
convolut	1139	1081	5,09 %
kernels	7299	6159	15,62 %
lms	2406	2002	16,79 %
matrix1	14705	10205	30,60 %
fir2dim	3899	3355	13,95 %
index	861	826	4,07 %
mat1x3	954	915	4,09 %
matrix2	15582	11082	28,88 %
Média	–	–	14,89 %

Tabela 5.1: Desempenho: resultado após a vetorização de MACs.

	Original	Otimizado	
Programa	Instruções	Instruções	Redução
convolut	65	73	-12,31 %
kernels	1359	1248	8,16 %
lms	226	297	-31,42 %
matrix1	113	120	-6,19 %
fir2dim	227	241	-6,17 %
index	42	41	2,38 %
mat1x3	54	61	-12,96 %
matrix2	121	126	-4,13 %
Média	–	–	-7,83 %

Tabela 5.2: Tamanho de código: resultado após a vetorização de MACs.

	Original	Otimizado	
Programa (laços)	Ciclos	Ciclos	Speedup
dot_prod	31	25	19,35 %
convolut	154	96	37,66 %
kernels	2744	2054	25,15 %
lms	666	354	46,85 %
loop5	471	441	6,37 %
matrix1	11013	6513	40,86 %
biquad_N_sections	473	465	1,69 %
convolution_array	232	186	19,83 %
dot_product_array	31	26	16,13 %
fir2dim	2160	1648	23,70 %
fir_array	378	363	3,97 %
fir	387	372	3,88 %
index	101	66	34,65 %
int01save	19610	19210	2,04 %
mat1x3	130	91	30,00 %
lms_array	155	139	10,32 %
mat1x3_array	182	161	11,54 %
matrix2	10998	6498	40,92 %
n_complex_updates	1404	1372	2,23 %
Média	–	–	21,23 %

Tabela 5.3: Desempenho dos laços: resultado após a geração/vetorização de MACs.

Capítulo 6

Contribuições e Trabalhos Futuros

6.1 Contribuições

Esta dissertação trata de um problema importante na geração de código para DSPs: a geração de instruções de multiplicação e acumulação e a paralelização dessas instruções em uma arquitetura SIMD.

Como contribuições deste trabalho podem ser citadas:

- Documentação sobre características gerais dos DSPs e apresentação dos problemas de geração de código eficiente para esses processadores pode servir como material de referência para trabalhos futuros.

- Algoritmos de detecção e vetorização de instruções de multiplicação e acumulação poderão ser incorporados a compiladores para DSPs já existentes, quando esses não forem capazes de gerar esse tipo de instrução ou não explorarem o potencial de paralelismo de dados das arquiteturas vetoriais (se for o caso). Até o presente momento (circa Fev. 2001) desconhecemos qualquer publicação em que a detecção de instruções de MAC tenha sido feita utilizando análise de fluxo de dados.

- Integração das otimizações implementadas a um produto comercial, o compilador Fixed-C [40] e [57] para o DSP *Countach40* [22] da *Conexant Systems Inc.* O compilador também será utilizado por outras empresas, como a *Sharp* e a *Nortel*, cujos produtos possuem em seu interior esse DSP.

- Maior eficiência dos aplicativos que executam no referido DSP e, conseqüentemente, menor consumo de energia nos sistemas que utilizam esse processador.

6.2 Trabalhos Futuros

6.2.1 Otimização dos Modos de Operação

Todos os exemplos de código *assembly* apresentados nesta dissertação, quando utilizavam instruções vetoriais de MAC, continham no interior dos laços duas instruções para controlar o modo de operação das unidades de MAC (simples/dual). Como foi descrito no Capítulo 5, são necessárias uma dessas instruções antes da instrução de MAC, para ajustar o modo de operação dual, e outra logo em seguida, para restaurar o modo de operação simples. Essas instruções acrescentam um *overhead* indesejado ao corpo do laço.

Instruções de controle de modo de operação dentro de um laço deveriam se beneficiar de uma análise de fluxo dos modos de operação, sendo removidas do interior do laço quando possível; as instruções desse tipo que fossem desnecessárias/redundantes, também seriam removidas posteriormente. Uma maneira de se realizar isto, é testar a existência de uma instrução para restaurar o modo de operação simples seguida imediatamente de uma instrução para ajustar o modo de operação dual; ambas poderiam ser removidas, sem implicar penalidades para o código resultante. Esse problema é similar ao de configuração dos dados na memória do DSP `TMS320C25` da *Texas Instruments* [51], onde uma análise semelhante é feita para minimizar o número de instruções de mudança de página de dados de um programa.

Quando diversas instruções que fazem uso do modo de operação dual estão presentes em um laço e encontram-se intercaladas por instruções que fazem uso do modo simples, o *overhead* introduzido pelas instruções de ajuste de modo de operação é grande, mas nada pode ser feito para removê-las. Entretanto, uma técnica chamada *loop fission* [56] poderia ser aplicada para quebrar um laço em dois, separando assim as instruções de modo dual das de modo simples. A viabilidade e a eficiência dessa transformação em um laço precisam ser cuidadosamente avaliadas.

6.2.2 Vetorização em Larga Escala

No presente trabalho foram vetorizadas apenas as instruções de MAC no interior de laços. Uma extensão imediata a essa otimização seria a tentativa de exploração de todas as oportunidades de execução de código em paralelo pelas unidades funcionais de um processador. As transformações decorrentes de uma vetorização em larga escala [56] possuem um impacto maior no código; portanto, para garantir a corretude do código gerado, é necessário que uma análise seja feita para coletar informações de dependência de dados entre *arrays* no interior de um laço e através de iterações do mesmo.

Existem duas abordagens para essa análise:

1. Construir o grafo de dependência de dados (DDG[1]) em um laço e verificar se o mesmo é acíclico. A maioria dos problemas estão em arestas que representam dependências do tipo RAW entre iterações sucessivas. A implementação é mais rápida, porém fica restrita ao problema de vetorização.

2. Construir o grafo de dependências do programa (PDG[2]) e procurar componentes conexos no mesmo. Essa abordagem requer um projeto mais longo, mas também é útil para detecção de paralelismo a nível de instrução em arquiteturas do tipo VLIW[3]. A infra-estrutura de compilação *Trimaran* [55] utiliza o PDG.

Pretendemos adotar uma dessas abordagens no futuro, para que a vetorização possa ser estendida para múltiplas instruções contendo dependências de dados mais complexas.

[1]Do inglês: **D***ata* **D***ependence* **G***raph.*
[2]Do inglês: **P***rogram* **D***ependence* **G***raph.*
[3]Do inglês: **V***ery* **L***ong* **I***nstruction* **W***ord.*

Capítulo 7

Conclusão

Devido à restrição de que a área de silício ocupada pelos processadores embutidos em dispositivos eletrônicos deve ser pequena, a memória total disponível nesses processadores não é grande (a memória de programa varia entre 48Kb e 64Kb em média e a memória de dados entre 128Kb e 256Kb, ao passo que o padrão atual para o tamanho da memória em computadores pessoais é de cerca de 64Mb). Além disso, como a maioria desses dispositivos é operada com bateria, é também desejável que os programas para execução nesses processadores sejam eficientes, (em termos de número de ciclos de máquina e acessos à memória), a fim de minimizar o consumo de energia e dessa forma estender o tempo de vida das baterias.

A tecnologia atual de compilação ainda não é capaz de gerar código denso e de alto desempenho para essa classe de processadores. Por esta razão, uma grande parte dos desenvolvedores de aplicações para DSPs utiliza-se de programação em *assembly* para tentar explorar todas as funcionalidades da arquitetura alvo. Entretanto, à medida em que os programas tornam-se maiores e mais complexos, codificá-los diretamente em *assembly* torna-se uma tarefa bastante complicada e sujeita a erros. Como conseqüência, exige-se dos compiladores para DSPs que eles sejam capazes de gerar as suas instruções especializadas, e diversos trabalhos de pesquisa começaram a ser realizados com este objetivo.

Nesta dissertação foram apresentadas duas otimizações que visam explorar oportunidades de utilização das instruções de multiplicação e acumulação, que fazem parte do conjunto de instruções de todos os DSPs. A geração e a vetorização dessas instruções por um compilador apóiam-se em algoritmos de análise de fluxo de dados e em técnicas de reestruturação de laços.

As duas otimizações desenvolvidas neste trabalho foram integradas em um compilador de produção da *Conexant Systems Inc.*, utilizado para gerar código otimizado para um de seus DSPs, o *Countach40* [22], que é utilizado em *modems*, placas de áudio e vídeo, GPS, etc. Essas otimizações mostraram-se eficientes, pois contribuíram para o aumento do desempenho do código gerado na maior parte do conjunto de testes realizados.

Bibliografia

[1] R. Adams e S. Gray. Using conditional execution to exploit instruction level concurrency. *j-SPE*, 25(9):1003–1019, Setembro de 1995.

[2] A. V. Aho, R. Sethi, e J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1988. Boston, MA.

[3] Analog Devices. *ADSP-2100 Family User's Manual*, terceira edição, Setembro de 1995.

[4] Andrew W. Appel. *Modern Compiler Implementation in C: Basic Techniques.* Cambridge University Press, Cambridge, United Kingdom, 1998.

[5] G. Araujo e S. Malik. Optimal code generation for embedded memory non-homogeneous register architectures. Em *Proc. 8^{th} International Symposium on System Synthesis*, pp. 36–41, Setembro de 1995.

[6] G. Araujo, S. Malik, e M. Lee. Using register-transfer paths in code generation for heterogeneous memory-register architectures. Em *Proc. 33^{rd} Design Automation Conference*, pp. 591–596, Junho de 1996.

[7] G. Araujo, A. Sudarsanam, e S. Malik. Instruction set design and optimizations for address computation in DSP processors. Em *9th International Symposium on Systems Synthesis*, pp. 31–37. IEEE, Novembro de 1996.

[8] Guido Araújo e Sharad Malik. Code generation for fixed-point DSPs.

[9] David F. Bacon, Susan L. Graham, e Oliver J. Sharp. Compiler transformations for high-performance computing. Em *ACM Computing Surveys*, volume 26, pp. 46–57, Dezembro de 1994.

[10] Steven Bashford e Rainer Leupers. Constraint driven code selection for fixed-point DSPs. Em *Proceedings of the 36^{th} Design Automation Conference*, Junho de 1999.

[11] Anupam Basu, Rainer Leupers, e Peter Marwedel. Array index allocation under register constraints in DSP programs. Em *Proceedings of the Int. Conf. on VLSI Design*. IEEE, Janeiro de 1999.

[12] J. Bier. DSP Processors and Cores: The Options Multiply. URL: http://www.eedesign.com/EEdesign/FocusReport9506.html.

[13] G. Chaitin. Register allocation and spilling via graph coloring. Em *Proc. of the ACM SIGPLAN'82 Symposium on Compiler Construction*, pp. 98–105, Junho de 1982.

[14] Marcelo Silva Cintra. Alocação Global de Registradores de Endereçamento Usando Cobertura do Grafo de Indexação e uma Variação da Forma SSA. Dissertação de Mestrado, IC - UNICAMP, Abril de 2000.

[15] Guido Costa Souza de Araújo. *Code Generation Algorithms for Digital Signal Processors*. Tese de Doutorado, Princeton University, Maio de 1997.

[16] DSPnet. Tech Online. URL: http://www.dspnet.com.

[17] Editorial. The future of computing. *The Economist Magazine*, pp. 79–81, Setembro de 1998.

[18] G. Goossens et al. Embedded software in real-time signal processing systems: Design technologies. Em *Proc. IEEE, Special Issue on Hardware/Sortware Co-Design*, 1997.

[19] R.S Glanville e S.L Graham. A new method for compiler code generation. Em *Proc. 5^{th} Annual ACM Symposium on Principles of Programming Languages*, pp. 231–240, Janeiro de 1978.

[20] J. L. Hennessy e D. A. Patterson. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., segunda edição, Setembro de 1997.

[21] J. L. Hennessy e D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., segunda edição, 1998.

[22] Peter Housel. *Countach-40 Architecture Reference - Version 1.2.1*. Conexant Systems Inc., Março de 2000.

[23] D. Kolson, A. Nicolau, Dutt N., e K. Kennedy. Optimal register assignment to loops for embedded code generation. *ACM Transactions on Design Automation of Electronic Systems*, 1(2):251–279, Abril de 1996.

[24] M. S. Lam e R. P Wilson. Limits of control flow on parallelism. Em *Proceedings of the 19th Annual International Symposium on Computer Architecture*, volume 20, pp. 46–57, Gold Coast, Australia, Maio de 1992. SIGARCH Computer Architecture News.

[25] Rainer Leupers. Exploiting conditional instructions in code generation for embedded VLIW processors. Em *Design Automation and Test in Europe (DATE)*, Munich, Germany, Março de 1999.

[26] Rainer Leupers. Code selection for media processors with SIMD instructions. Em *Design Automation and Test in Europe (DATE)*, Paris, France, Março de 2000.

[27] Rainer Leupers, Anupam Basu, e Peter Marwedel. Optimized array index computation in DSP programs. Em *Proceedings of the ASP-DAC*. IEEE, Fevereiro de 1998.

[28] Rainer Leupers e Fabian David. A uniform optimization technique for offset assignment problems. Em *Proceedings of the ACM SIGDA 11th International Symposium on System Synthesis*, pp. 3–8, Dezembro de 1998.

[29] Rainer Leupers e Peter Marwedel. Funtion inlining under code size constraints for embedded processors. Em *International Conference on Computer-Aided Design (ICCAD)*, San Jose, CA, 1999.

[30] S. Liao, S. Devadas, K. Keutzer, e S. Tjiang. Instruction selection using binate covering. Em *Proc. of the International Conference on Computer-Aided Design*, pp. 393–399, 1995.

[31] S. Y. Liao, S. Devadas, K. Keutzer, A. Wang, e S Tjiang. Storage assignment to decrease code size. *ACM Transactions on Programming Languages and Systems*, 18:235–253, Maio de 1996.

[32] C. Liem, Trevor M, e Paulin P. Instruction-set matching and selection for DSP and ASIP code generation. Em *European Design and Test Conference*, pp. 31–37, 1994.

[33] W. Lin. An Optimizing Compiler for the TMS320C25 DSP Processor. Dissertação de Mestrado, University of Toronto, 1995.

[34] Peter Marwedel. Code generation for embedded processors: An introduction. Em P. Marwedel e G. Goossens, editores, *Code Generation for Embedded Processors*. Kluwer Academic Publisher, Boston, MA, junho de 1995.

[35] Motorola, Inc., Austin, TX. *DSP56000 24-Bit Digital Signal Processor Family Manual*, 1990.

[36] Steven. S. Muchnick. *Advanced Compiler Design and Implementation.* Academic Press / Morgan Kaufmann Publishers Inc., Julho de 1997.

[37] NEC. *μPD77016 User's Manual*, 1993.

[38] P. Paulin, M. Cornero, e et al. C. Liem. Trends in embedded systems technology. Em M. G. Sami e G. De Micheli, editores, *Hardware/Software Co-Design.* Kluwer Academic Publishers, 1996.

[39] Johan Van Praet, Gert Goossens, Dirk Lanneer, e Hugo De Man. Instruction set definition and instruction selection for ASIPs. Em *European Design and Test Conference*, pp. 11–16, 1994.

[40] Bores Signal Processing. Fixed-Point DSPs. URL: http://www.bores.com/chips/fixed.htm.

[41] Amit Rao e Santosh Pande. Storage assignment optimizations to generate compact and efficient code on embedded DSPs. Em *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pp. 128–138, Maio de 1999.

[42] Amit Rao e Santosh Pande. Storage assignment using expression tree transformations to generate compact and efficient DSP code. Em *INTERACT - 3rd Workshop on Interaction between Compilers and Computer Architectures*, 1999.

[43] K.E. Rimey. *A Compiler for Application-Specific Signal Processors.* Tese de Doutorado, Princeton University, Setembro de 1987.

[44] Mazen A. R. Saghir, Paul Chow, e Corinna G. Lee. Exploiting dual data-memory banks in digital signal processors. Em *SIGARCH Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pp. 234–243, Boston, MA, Outubro de 1996.

[45] Mazen A. R. Saghir, Paul Chow, e Corinna G. Lee. Exploiting dual memory banks using automatic data partitioning and data duplication. Em *ACM SIGPLAN Conference on Programming Languages Design and Implementation*, 1996.

[46] J. N. Shurkin. *Engines of the Mind: The Evolution of the Computer from Mainframes to Microprocessors.* WWNorton & Co, 1996.

[47] Will Strauss. The 1999 DSP Market Heats Up. *Forward Concepts*, 1999. URL: http://www.forwardconcepts.com.

[48] A. Sudarsanam e S. Malik. Memory bank and register allocation in software synthesis for ASIPS. Em *International Conference on Computer Aided Design*, pp. 388–392, 1995.

[49] A. Sudarsanam, S. Malik, S. Tjiang, e S. Liao. Optimization of embedded DSP programs using post-pass data-flow analysis. Em *IEEE International Conference on Acoustics, Speech, and Signal Processing*, Abril de 1997.

[50] Ashok Sudarsanam. *Code Optimization Libraries for Retargetable Compilation for Embedded Digital Signal Processors.* Tese de Doutorado, Princeton University, Novembro de 1998.

[51] Texas Instruments. *TMS320C2x User's Guide*, 1990.

[52] Texas Instruments. *TMS320C5x User's Guide*, 1993.

[53] Texas Instruments. *TMS320C54x User's Guide*, 1995.

[54] Texas Instruments. *TMS320C6x User's Guide*, 1999.

[55] The Trimaran Compiler Infra-Structure. URL: http://www.trimaran.org.

[56] Michael J. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Publishing Company, 1996.

[57] C. R. Yates. Fixed-Point Arithmetic: An Introduction. URL: http://www.shadow.net/~yates/fp.htm.

[58] V. Zivojnovic, J. M. Velarde, C. Sclåager, e H. Meyr. DSPstone – a DSP-oriented benchmarking methodology. Em *International Conference on Signal Processing Applications and Technology (ICSPAT)*, Agosto de 1994.