UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

INSTITUTO DE INFORMÁTICA

CURSO DE PÓS GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

# ALGORITMOS E ARQUITETURAS PARA O DESENVOLVIMENTO DE SISTEMAS COMPUTACIONAIS

por

LUIGI CARRO

Tese submetida como requisito
parcial para a obtenção de grau de
Doutor em Ciência da Computação

Prof. Altamiro Suzim
Orientador

Porto Alegre, maio de 1996

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

CARRO, Luigi.
Algoritmos e arquiteturas para o desenvolvimento de sistemas computacionais Luigi Carro. Porto Alegre: CPGCC da UFRGS, 1996.

140 p.:il.

Tese (doutorado) - Universidade Federal do Rio Grande do Sul. Curso de Pós-Graduação em Ciência da Computação, Porto Alegre, BR-RS, 1996. Orientador: Suzim, Altamiro Amadeu.

1.HW-SW codesign. 2.Microprocessadores ASIP. 3.Arquiteturas RISC. 4.Microcontroladores. 5.Concepção de circuitos integrados. I. Suzim, Altamiro Amadeu. II.Título

Myste had said, *Problems should be solved by those who see them*. There wasn't anybody else.

Stephen Donaldson, A Man Rides Through

quase Hai-Kai: pensando estou na vida,
enquanto faço poesia.

Eu, enquanto acabava a tese.

Ao Brasil, seja lá como seja. À familia

# Agradecimentos

Como é de praxe, muitas pessoas participaram de maneira direta ou indireta para realização deste trabalho. Durante 4 anos vários auxiliares de pesquisa colaboraram para diversos estágios desta tese. A cada um deles meu obrigado, em especial para André Hentz, com sua seriedade e seu já clássico Edllex, e Fernando Soto, com seu Ggmod, que trabalharam nas ferramentas de layout para realização do Risco full-custom; Marcio Rosa da Silva, Marcio Gil Fachin e Paulo Godoy trabalharam no layout simbólico das diferentes células do Rico full-custom. Gustavo Franceschini auxiliou na implementação dos simuladores STX (elétrico) e SLX (lógico), e Antonio Marcos Parisoto colaborou na análise de timing usando o STX. Roberta Burger, como uma locomotiva, trabalhou no Risco Standard Cells, enquanto que Adão Souza, a quem sempre interessava o espírito da coisa, trabalhou no Risco EPLD e no simulador SHC do Risco; Lianete Klauck, com seu constante bom humor, colaborou na implementação do sistema de CAD, realizando a movimentação de operações, enquanto que Daniela Lima abraçava o Montador Universal. Da nova safra, Gilberto Migliorin teve um trabalho incansável no simulador C do Risco, enquanto que Guilherme Pereira bravamente sintetizava o 8051 em VHDL. Seguindo um trabalho iniciado com André Hentz, Cristiano Pinto e Pedro Benoni trabalharam com circuitos analógicos, para na continuação deste trabalho possibilitar a integração de sistemas mistos. Espero que todos tenham sido tão beneficiados em conhecimento quanto eu o fui em trabalho e dedicação. Espero também encontrar a todos no futuro com boas recordações da época em que trabalhamos juntos.

Algumas outras pessoas também auxiliaram em algumas partes desta tese. Marcus Kindel e Carlos Alba trabalharam no Risco-WCS durante uma disciplina da pós-graduação, e Jorge Ferreira colaborou na realização do simulador de barramento em SHC. Ressalte-se que sem o trabalho de Alexandre Junqueira o Risco não teria sido implementado.

Agradeço também a gentileza do Prof. Alain Greiner, da Universidade de Paris VI, que me acolheu por um mês durante as férias de julho, para que pudesse trabalhar na descrição do DLX-WCS. O Prof. Pirouz Barzaghan, também de Paris VI, foi o responsável pela descrição original do DLX, e a ele devo não somente um código facilmente alterável, mas também saudáveis discussões técnicas sobre arquiteturas de processadores. Da mesma universidade, gostaria de agradecer a Jean-Yves Brunel, por me colocar a disposição o sistema Mpsas, e pelas agradáveis conversas sobre HW-SW codesign, assim como Ivan Augè, pela paciência em me explicar o sistema Alma.

Aos professores Flávio Wagner (CPGCC-UFRGS), Julio Salek Aude (NCE-UFRJ) e Marcelo Lubaszweski (DELET-UFRGS) meu muito obrigado por terem aceito participar da banca de avaliação deste trabalho.

No ambiente de trabalho, agradeço a todos aqueles que tiveram de me suportar, seja no CPGCC, seja na Elétrica, e tenho esperanças que compreendam que a seriedade com que encaro as minhas tarefas seja perdoada, e compreendida como uma extrapolação da vontade de acertar. Aos professores Sérgio Bampi e Ricardo Reis, do GME, meu muito obrigado, pelo apoio e incentivo que sempre me dedicaram durante os anos de trabalho conjunto, e em especial nesta tarefa maior de realização da tese. Agradeço também ao prof. e amigo Ramon Poisl, pela impressão deste trabalho, e ao prof. Fernando Barbosa, pelas constantes consultorias sobre conversão de gráficos.

No plano técnico e pessoal, com certeza este trabalho não poderia ter sido realizado sem o apoio de meu orientador, Altamiro Suzim, que resta sempre um jardineiro de idéias. Espero sinceramente que o desenvolvimento desta tese tenha sido gratificante para ele como o foi para mim, e que a integração que atingimos possa ser repetida ao longo dos muitos anos de trabalho em conjunto que estão por vir.

À Dona Teresinha, do Morro da Cruz, cujo nome surgiu como brincadeira de cafezinho, agradeço os anos de suporte através de seus impostos, e acredito estar colaborando para repagá-la em serviços, fazendo um país melhor do que encontrei.

Finalmente, talvez não caiba um agradecimento à família, que é a parte mais importante da minha vida, mas sim um pedido de desculpas pelas constantes ausências. Espero que todos concordem que este foi um sacrifício que valeu a pena.

# Sumário

# Lista de figuras

# Lista de tabelas

## Lista de abreviaturas

AD Analógico Digital
AHDL Altera Hardware Description Language
ASIC Application Specific Integrated Circuit
ASIP Application Specific Instruction-set Processor
ASIS Application Specific Integrated System
BB Basic Blocks
CAD Computer Aided Design
CI Circuito Integrado
CPU Central Procesing Unit
ECSD Environment for Computational Systems Development
EPLD Eraesable Programmable Logic Device
FPGA Field Programmable Gate Array
FSM Finite State Machine
GA Gate Array
HDL Hardware Description Language
HLS High Level Synthesis
HW Hardware
LE Logic Element
PC Parte de Controle
PLD Programmable Logic Device
PID Proporcional-Integral-Direvativo
PO Parte Operativa
RAM Random Access Mammory
ROM Read Only Memory
SC Standard Cells
SW Software
VHDL Very High Speed Integrated Circuit Hardware Description Language
VLIW Very Large Instruction Word
WCS Writable Control Store

# Resumo

Este trabalho trata de arquiteturas e algoritmos para o desenvolvimento de sistemas computacionais. Tais sistemas são constituidos de um microprocessador (específico ou comercialmente disponível), de seu conjunto de programas e de um HW dedicado que será utilizado para otimização do sistema. O objetivo principal desta tese é demonstrar que, presentemente, a linha divisória entre HW e SW é cada vez mais tênue, e a transição entre um e outro pode ser feita de maneira suave pelo projetista de sistemas, na busca de um ponto ótimo no balanço entre custo e desempenho.

Apresenta-se em seqüência o ambiente de CAD, a classificação de rotinas e os métodos de otimização tendo em vista esta classificação para o aumento de desempenho de sistemas computacionais. A seguir são apresentadas técnicas para processadores dedicados de arquitetura Risc, visando a otimização de certos tipos de programas. Os resultados de aceleração são apresentados para um conjunto de exemplos.

Tendo em vista o mercado nacional de eletrônica, fortemente baseado em microcontroladores, estudam-se e mostram-se possibilidades de otimização e integração de sistemas baseados em tais processadores, assim como a aplicabilidade das mesmas técnicas para processadores dedicados. A viabilidade técnica desta realização é discutida através de exemplos baseados em aplicações reais.

Finalmente, a validação de sistemas computacionais, em especial aqueles trabalhados nesta tese, é discutida.

**Palavras-chaves**: HW-SW codesign, Microprocessadoes ASIP, Arquiteturas Risc, Microcontroladores, Concepção de circuitos integrados.

# Title: "Algorithms and architectures to the development of computational systems"

## Abstract

*This work discusses architectures and algorithms for the development of* computational systems, which are based on a microprocessor (custom or off-the-shelf), the set of application programs and a dedicated HW, used to increase the performance of the whole system. The goal of this work is to show that, nowadays, the division line between SW and HW is smooth, and the transition from one to the other can be achieved by the system designer using a specific CAD in order to obtain a trade-off between cost *and performance.*

The CAD environment is presented, followed by routine classification and optimization methods based on the former classification to increase the performance of the system. Techniques devoted to systems based on dedicated Risc processors are showed next, to optimize certain type of programs. Positive results are shown for a set *of examples.*

Since the Brazilian electronics market is strongly based on microcontrollers, the study and results of optimization techniques regarding this type of systems are also presented. The same techniques can be applied to dedicated processors as well. Results of this proposal are obtained for a set of real world examples.

The last topic of this work regards the validation of computational systems, mainly those presented throughout this work.

**Key words**: HW-SW codesign, ASIP like microprocessos, Risc architectures, Microcontrollers, Integrated circuit design.

# 1 Introdução e motivação

## 1.1 Introdução

As disponibilidades tecnológicas da microeletrônica dos últimos anos oferecem uma variedade de opções ao projetista de sistemas desejoso de implementar um projeto através de um ASIC (Application Specific Integrated Circuit). O volume de negócios envolvendo estes circuitos chegou a 7,3 bilhões de dólares em 1990, com perspectivas para 12,4 em 1995 ([ELE 91]).

Os circuitos full custom e os semi-custom, em 1989, foram responsáveis por 29% do mercado de semicondutores ([CLA 90]). Considerando-se que o preço médio dos circuitos integrados nos sistemas eletrônicos é de apenas 7% ([PIS 89]), verifica-se que há ainda um enorme espaço para crescimento, ou seja, existe ainda potencial para o uso da eletrônica em sistemas. Este crecimento provoca por sua vez uma redução nos custos de implementação dos sistemas eletrônicos.

Segundo dados encontrados em [ELE 91], há perspectiva de uma queda do percentual de ASICs realizados com técnica full custom para 25% em 1995, sendo substituídos por projetos com gate array (GA), Standard Cells (SC) ou lógicas programáveis, estas últimas devendo crescer até 6% do mercado total em 1995.

Esta situação reflete o estado do início da década de 90 dos produtos eletrônicos, cujo tempo de vida não ultrapassa os 5 anos na grande maioria dos casos. Logo, as indústrias de sistemas eletrônicos não podem dedicar 2 anos ao desenvolvimento de um produto, que seria um tempo excessivo. O conceito de tempo de chegada de um produto ao mercado consumidor (time to market) torna-se um item fundamental na vida útil de um projeto. Tendo-se em vista que a concorrência também deve lançar seus produtos no mercado, o tempo para amortização dos custos de um projeto torna-se cada vez mais curto ([COM 91]).

Contraditoriamente, o custo para desenvolvimento de CIs, seja em GA, SC ou full custom é presentemente alto. Por custo entenda-se não somente o preço da pastilha em si (visto que GA são bastante acessíveis), mas o custo do desenvolvimento do projeto e sua manutenção.

Empresas cuja principal atividade é a venda de sistemas eletrônicos podem não estar interessadas na organização e manutenção de um departamento de CIs. Por outro lado, a contratação de uma empresa de projetos traz os problemas de sigilo industrial e acompanhamento da evolução do projeto a posteriori. Existe ainda o problema de que, como o ASIC é geralmente projetado isolado do resto do sistema, quando retorna da produção muitas vezes o CI correto do ponto de vista da fabricação não se enquadra no projeto no qual deveria participar ([MAR 90]).

A solução oferecida pelos fabricantes de circuitos integrados parecia ser o Gate Array. Esta técnica de implementação de circuitos, contudo, sofre hoje forte concorrência dos componentes programáveis pelos usuários, os PLDs, EPLDs e FPGAs. Estes oportunizam o conceito de fundição de silício em casa, devido ao baixo tempo entre a idealização de um circuito até a sua prototipação, que pode ser feita na própria

bancada de trabalho do engenheiro de desenvolvimento. Estas tecnologias já permitem grande integração, pois encontram-se FPGAs na faixa dos 2500 Flip-Flops ou 5000 portas equivalentes.

Por outro lado, os grandes fabricantes de CIs atacam em três direções naturais. A primeira é aquela normal e esperada, a da evolução tecnológica. Transistores de 0,7 micra de canal já são utilizados para o projeto de ASICs, enquanto que a geração de memórias estáticas utiliza tecnologias de 0,5 micra. Este apoio tecnológico permite que circuitos com 100 mil transistores sejam realizados agora, podendo-se considerar circuitos de 20/30 mil transistores de pequeno porte face às disponibilidades tecnológicas atuais.

A segunda migração diz respeito aos métodos de projeto. O conceito de tempo de chegada ao mercado tornou-se de tal modo importante que, mesmo empresas que vendem circuitos em larga escala (e portanto a área destes circuitos é fundamental para o preço dos mesmos), utilizam correntemente geradores de módulos e implementação automática em seus projetos. Mesmo que a implementação automática leve a resultados não ótimos em termos de área, potência ou velocidade, o produto final chega mais rapidamente ao mercado.

A terceira e última migração diz respeito ao modo de venda dos CIs, já que o custo de um CI é sempre menor do que o do sistema ao qual está agregado. Em 1989 o percentual médio de custo dos integrados no preço de um sistema eletrônico era de apenas 7% [PIS 89]. É plenamente possível que, com o aumento da integração, o valor agregado do CI no sistema onde ele deva ser utilizado aumente consideravelmente. Um exemplo disto é a nova geração de micros pessoais baseada sobre o processador i486 ou o Pentium. O conjunto de integrados possui um valor agregado alto, pelo elevado número de funções que cada CI realiza, e isto se reflete no preço final do sistema, onde a placa com o processador é responsável por ao menos 33% do preço final.

É o mercado de sistemas eletrônicos que possibilita às empresas de semicondutores a diversificação da produção. Um exemplo evidente é a Motorola, que investe pesadamente na telefonia móvel internacional como mercado cativo para seus circuitos integrados. Outro exemplo de busca de mercado é o de empresas européias de semicondutores, que buscam associações entre si e com fabricantes de sistemas para resistir à concorrência americana e japonesa. Empresas como ST-Microelectronics e Siemens unem-se no projeto de memórias RAM de gigabits, ao mesmo tempo em que estreitam o laço com fabricantes de sistemas (como Philips e Thomson, por exemplo) para obtenção de um mercado cativo para os semicondutores europeus.

Esta migração é naturalmente suportada pelas grandes empresas de semicondutores. O conceito de sistemas em silício (systems on silicon) pressupõe não somente o projeto de um grande integrado e de seus companheiros no sistema, mas sim toda uma nova família de desafios como encapsulamentos especiais, dissipação de potência e outros, problemas estes que são naturalmente estudados e resolvidos em ambientes de fábricas de semicondutores.

## 1.2 O binômio CAD/Tecnologia

Os fatos anteriormente relacionados levam à conclusão que o domínio da tecnologia é um fator determinante para posicionamento no mercado de ASICs. Além disto, pareceria que os circuitos full custom tendem a desaparecer fora do raio de ação das grandes empresas, enquanto que os projetos que necessitassem de um baixo tempo de projeto seriam naturalmente realizados com famílias de lógicas programáveis.

Na verdade, existe espaço para toda uma série intermediária de circuitos. O uso de tecnologias de ponta com transistores de 0,7 ou 0,5 micra em projetos não invalida o uso de tecnologias mais simples e baratas como aquelas com transistores de 1.2 micra. Esta, por sinal, será mais estável e provavelmente com um maior rendimento que a tecnologia mais avançada. Logo, enquanto as grandes empresas de semicondutores produzem circuitos com mais de 2 milhões de transistores, existem pequenas fundições (*foundries*) que oferecem tecnologias capazes de integrar projetos de 50 a 100 mil transistores. Este número é significativamente maior que aquele que os projetos com GAs podem efetivamente alcançar, e muitas vezes maior que o número de portas disponível em famílias de lógica programável simples.

Contudo, o grande gargalo para ocupação deste espaço intermediário parece não ser a tecnologia, mas sim o CAD. Este passou durante os últimos anos da condição de mero auxiliar nas tarefas de projeto à de elemento fundamental para o desenvolvimento de um circuito complexo.

Se existem no mercado de ASICs tecnologias estáveis e de relativo baixo custo, o mesmo não se pode dizer do CAD. O custo de projeto de um CI ainda é alto, visto que devem ser solucionados problemas que vão desde o projeto do algoritmo até a implementação física em uma dada tecnologia. Ainda são necessários engenheiros dedicados e especializados, os métodos de projeto são baseados em metodologias ad hoc, a manutenção de um projeto é, em geral, um problema intransponível, o tempo de projeto e o teste idem. Esta situação é claramente contra o conceito de time to market. Mais ainda, não se tem garantias de que o ASIC funcionará no sistema para o qual foi projetado. Segundo Markowitz ([MAR 90]), 90% dos ASICs projetados são considerados funcionalmente corretos pelo fabricante de semicondutores ou pela análise das especificações iniciais, mas somente cinquenta por cento (50%) dos ASICs funcionará quando colocado na placa. Isto demonstra o quão longe de seu efetivo potencial está a microeletrônica. É o sistema de CAD o responsável pela mudança deste status quo.

Como existem diversas opções tecnológicas, diversos caminhos de implementação devem ser suportados, desde o full custom ao PLD, sem esquecer os integrados standard ou off the shelf até por uma questão de manutenção de antigos projetos. Portanto, o ASIC deve ser considerado como parte de um sistema maior, o sistema eletrônico.

Em relação ao CAD de sistemas para o mercado atual, dois conceitos devem ser ressaltados, o de tempo de chegada ao mercado e o de que o CAD tornou-se um parceiro indispensável no trio projetista/cad/tecnologia, que atuam para resolução de determinado problema. Como tal, as ferramentas de CAD da nova geração devem ser orientadas para exploração do espaço hoje existente no auxílio ao projeto de sistemas em silício.

O espaço mencionado manifesta-se no uso de tecnologia como insumo para os projetistas de sistema. O valor agregado dos sistemas é maior quando a criatividade pode ser explorada ao máximo, e portanto muitas atividades do CAD devem ser voltadas para liberação da criatividade. O CAD serve portanto de ponte para acesso a uma dada tecnologia de implementação à resolução de um projeto de sistema eletrônico.

## 1.3 O desafio atual

As disponibilidades de CAD no mercado atual são intensas. Os vendedores de CAD tem disponível uma enorme variedade de produtos, que cobrem desde o trabalho de layout até a síntese de alto nível. Em resumo, dados os devidos recursos, o desenvolvimento de um projeto depende apenas da capacidade de projetistas de realizá-lo.

No campo dos circuitos complexos realizados principalmente pelas grandes empresas de semicondutores, o projetista de CIs já está acostumado a utilizar-se de Linguagens de Descrição de Hardware (HDLs) para modelagem das placas onde o ASIC trabalhará. O tempo de projeto do ASIC é reduzido drásticamente pelo uso de ferramentas de implementação automática, e se o projeto foi cuidadosamente particionado entre diferentes circuitos e simulado em nível de sistema, as probabilidades de sucesso do CI são altas. Contudo, esta é uma situação de grande empresa de semicondutores.

Este ambiente ideal não corresponde à realidade dos médios e pequenos fabricantes de sistemas por três motivos: o tempo para prototipação, o CAD para sistemas e a falta de costume dos projetistas de sistema no uso de CADs complexos (como HDLs, por exemplo). Em relação ao tempo de prototipação, é fácil verificar que o projetista deve esperar algumas semanas para receber de volta seu protótipo. Durante este tempo, se surgirem modificações do tipo "e se incluíssemos esta outra característica ao projeto" elas serão completamente descartadas, senão pelo tempo, até pelo custo de mais uma rodada de prototipação. FPGAs e EPLD oferecem um tempo de prototipação bastante curto, próximo de zero. Contudo, são limitados no número de portas que podem alcançar e no tipo de circuito alvo (sempre um circuito digital). Os FPGAs oferecem porém uma característica muito agradável aos olhos dos projetistas tradicionais de circuitos: a tentativa e erro. Como são facilmente reprogramáveis, o projetista de sistema sente-se mais seguro ao utilizá-lo. Como são por construção totalmente testáveis, o projetista tem total garantia da sua programação.

Estas características das famílias programáveis fazem com que sejam a opção preferencial para quem deseja projetar um ASIC. Mesmo que toda a lógica não esteja dentro de um único FPGA, pode-se utilizar mais de um em uma placa, obviamente. Para os circuitos complexos, que necessitam operadores especiais (memória extensa, por exemplo), circuitos analógicos ou implementação de algoritmos velozes deve-se migrar para GA ou SC.

Tradicionalmente houve uma divisão entre o projetista de sistemas e o de microeletrônica, visto que o conjunto de ferramentas usado por um e outro era de natureza diferente. Por exemplo, cabe ao projetista de sistema a decisão de qual microprocessador utilizar, se utilizar mais de um processador, se um HW dedicado deve ser requisitado, etc.

A todas estas decisões pode-se chamar de projeto de sistemas. O projetista de microeletrônica parte eventualmente de um mesmo algoritmo, mas verificará a factibilidade de seu produto através de um outro conjunto de restrições. A área do circuito será suficientemente pequena? A tecnologia permite alcançar as frequências desejadas? A dissipação está dentro dos limites do encapsulamento especificado?

Sistemas eletrônicos nos dias atuais são na sua grande maioria microprocessados. Isto devido não somente ao conceito de um sistema digital programável, e portanto facilmente modificável, mas também pelo baixo custo e alto desempenho de microprocessadores, aliada à padronização do HW necessário para o funcionamento de uma placa. Ao se realizarem sistemas em silício tem-se duas visões de atividades: do lado do projetista de sistemas, significa mais e mais a passagem de microprocessadores, suas memórias e periféricos para dentro do CI, enquanto que para os fabricantes de semicondutores, pode significar realmente algoritmos complexos.

## 1.4 Proposta e organização do trabalho

Este trabalho propõe-se a estudar soluções, investigar algoritmos e arquiteturas de implementação de sistemas segundo um modelo a ser apresentado, e validar os mesmos em um ambiente de CAD. Além disto, é demonstrado que com pequenas alterações arquiteturais um processador RISC pode ser visto como um ASIP, isto é, um processador dedicado à aplicação. Basicamente, tenta-se estabelecer uma ponte entre o HW e SW em nível de micro-arquitetura, mostrando a intercambialidade entre ambos.

É importante ressaltar que os ambientes de CAD para HLS atualmente aceitam entradas que descrevem o comportamento do HW e tentam mapeá-lo para um conjunto de estruturas. O conceito de sistema presente neste trabalho transcende a simples síntese de HW, e abrange a produção de sistemas eletrônicos complexos. Ferramentas de síntese automática são uma parte do ambiente global.

Um dos objetivos do trabalho é também o aumento da vida útil dos projetos eletrônicos, pela reutilização do HW. Antes da troca completa de um ambiente de projeto realizam-se pequenas alterações de modo a ampliá-lo ad hoc. A prototipação rápida e todo o ambiente de CAD deve ser capaz de suportar a pergunta "e se fosse modificado o item ...", cuja resposta seria a variação dos principais parâmetros de projeto como área, consumo, desempenho, encapsulamento, etc.

Neste trabalho serão estudadas técnicas de otimização de sistemas computacionais. A definição destes sistemas no âmbito do trabalho e uma visão geral do CAD proposto é apresentada no próximo capítulo. O capítulo 3 mostra os diferentes algoritmos para otimização de sistemas que possuam um processador, tentando otimizar o uso do processador e de um eventual HW dedicado. São apresentadas estratégias de classificação de rotinas tendo em vista sua otimização. O método de implementação destas otimizações depende da classificação.

O capítulo 4 apresenta as modificações arquiteturais necessárias para a otimização de rotinas com muitas instruções de salto ou com muitos acessos à memória de dados. Estas modificações exigem a presença de um microprocessador de arquitetura

proprietária que possa ser modificada, e correspondente suporte de SW. Resultados destas mudanças arquiteturais são também apresentados.

O capítulo 5 apresenta um método de implementação de sistemas baseado em processadores standard, sobre cuja arquitetura não se podem realizar modificações, tendo em vista a compatibilidade de SW e eventualmente a temporização de interrupções, por exemplo.

Um método de validação para estas estratégias de otimização é apresentado no capítulo 6, onde se discutem também as características necessárias do ambiente de desenvolvimento de sistemas computacionais modernos. Finalmente, o capítulo 7 apresenta as conclusões deste trabalho e discute perspectivas futuras tendo em vista sua continuidade.

# 2 Sistemas computacionais e proposta de trabalho

Neste capítulo busca-se a definição do conceito de sistemas eletrônicos no âmbito do trabalho. A descrição e partição de sistemas é discutida, com ênfase para os sistemas microprocessados, cujas técnicas de implementação serão discutidas nos próximos capítulos.

## 2.1 O conceito de sistema eletrônico

Uma vez que se pretende atacar o problema da implementação de sistemas, é importante precisar-se o conceito de sistema eletrônico. Sistema, segundo Ogata ([OGA 70]), é uma combinação de componentes que atuam conjuntamente e realizam um certo objetivo. O conceito de sistema extrapola a realização física, podendo ser aplicado a fenômenos abstratos como em fenômenos sociais e econômicos. Um sistema eletrônico seria aquele em que seus componentes fossem eletrônicos. Recursivamente, é preciso definir os componentes do sistema e seus objetivos.

O objetivo de um sistema informa sobre a finalidade maior do mesmo. Pode-se dizer informalmente que se realiza um sistema para o controle de plantas industriais, para a transmissão de imagens compactadas, para resolução de equações diferenciais, etc. Os componentes de um sistema informam sobre a natureza de sua implementação: eletrônicos, mecânicos, químicos. A tecnologia moderna quase sempre exige que existam sistemas complexos mistos. Um automóvel é um sistema mecânico de locomoção com partes eletrônicas importantes. Já o computador é considerado um sistema eletrônico com partes mecânicas importantes, como a refrigeração ou o teclado, por exemplo.

Boute, em [BOU 91], propõe uma classificação de sistemas segundo seus modelos matemáticos primários. Assim, sistemas podem ser divididos em:

- Estáticos, ou seja, em relação a fenômenos independentes do tempo, e Dinâmicos, que abrangem fenômenos que variam com o tempo. Um processo é um modelo matemático do comportamento de um sistema dinâmico.
- Contínuos, envolvendo valores Contínuos (ou conjuntos incontáveis) versus Discretos, envolvendo conjuntos enumeráveis.

Em paralelo com a classificação abstrata contínuo-discreto, existe a divisão mais concreta analógico-digital, onde o sistema é visto em termos dos blocos que o constituem.

Existe uma ambiguidade de termos quanto a especificação. É comum encontrarem-se descrições como sinônimos de especificações (com o uso, por exemplo, de VHDL [IEE 87]), agregando-se ainda um conjunto de requisitos de realização, com alto nível de abstração, descrevendo características externas, em contraste com especificações de projeto, em nível mais concreto, descrevendo realização interna. A figura 2.1 apresenta um diagrama que resume a divisão entre a especificação e a realização de sistemas eletrônicos.

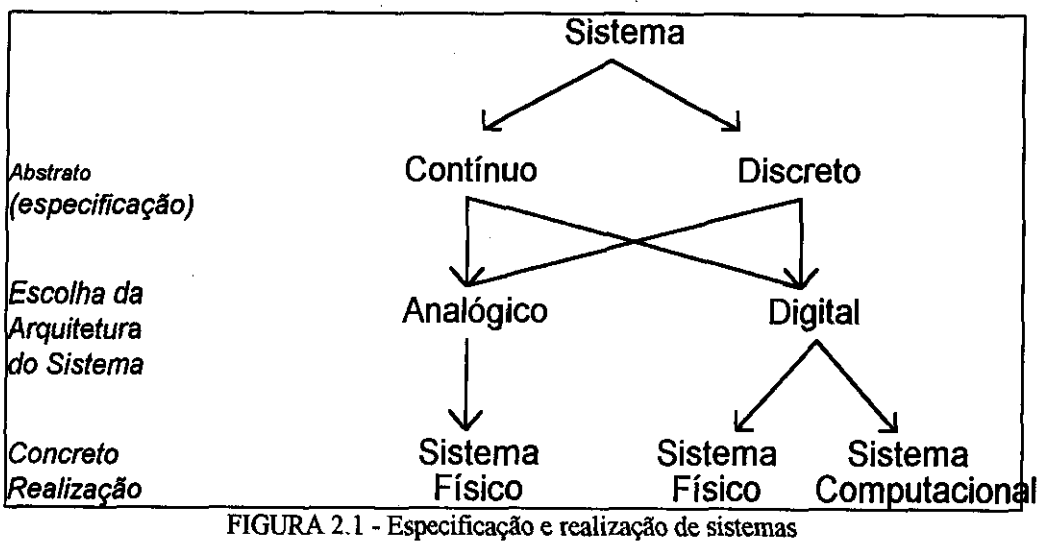



FIGURA 2.1 - Especificação e realização de sistemas

O diagrama é recursivo: um filtro eletrônico é geralmente especificado no domínio analógico, através de uma função de transferência ou de requisitos sobre a banda passante e de corte, mas pode ser realizado de várias maneiras:

- como um circuito analógico com componentes passivos ou ativos;
- como um circuito digital (somadores, multiplexadores, registradores);
- como um sistema computacional através de um processador e de um programa a ser por ele interpretado.

Supondo-se a implementação do sistema computacional com um processador de sinais, este por sua vez pode ser visto como um sistema discreto, especificado através de seu comportamento desejado como interpretador e realizado como um sistema digital (a não ser que um nível adicional de interpretação seja inserido). Mais uma vez, os blocos que compõem seja o sistema digital seja o processador são sistemas discretos, que por sua vez são realizados por meio de circuitos digitais construídos com componentes analógicos consistindo de resistores, capacitores e transistores.

Pelo diagrama da figura 2.1 nota-se que um algoritmo não é somente uma forma para especificar um sistema. Na verdade, um algoritmo é a própria realização do sistema, visto que já se decidiu por um sistema computacional (seja ele processador ou sistema digital sintetizado a partir de VHDL, por exemplo).

No âmbito deste trabalho, a ênfase será dada na implementação de sistemas computacionais, visto que são os mais gerais e de maior aplicação nos tempos atuais. As tarefas para realização de um sistema compreendem portanto a implementação de HW, de SW e a integração entre ambos. Os objetivos dos sistemas computacionais são os mais variados: controle de equipamentos domésticos e de alarmes, injeção eletrônica de automóveis, televisão de alta resolução, controle de processos industriais, transmissão de dados, impressoras de última geração, máquinas de fax, terminais bancários, instrumentos eletrônicos e muitos outros exemplos.

Um projetista de sistemas computacionais deve ser capaz de combinar componentes programáveis, como microprocessadores e seus periféricos, em um conjunto capaz de suportar o SW que deve ser interpretado por estes componentes físicos.

## 2.2 A implementação de sistemas computacionais

O projeto lógico foi consolidado anos atrás, no final da década de 80. O projeto VLSI (Very Large Scale Integration) tornou-se largamente disponível mais recentemente, com os projetos multi-usuários. Inicia-se nesta década um esforco de ensinar projeto de alto nível ou seja, aquele feito a partir de Linguagens de Descrição de Hardware (HDL) seguidas de simulação e síntese ([GAJ 93]). É preciso agora definir ambientes que explorem o projeto de SW e HW em conjunto.

Em [SCH 93] apresenta-se o uso de diagramas de tempo e lógica temporal para especificar sistemas. Contudo, discute-se a aplicação de tais métodos a todos os sistemas: em protocolos pode ser interessante, mas em sistemas que requeiram muitas computações, como em ambientes para processamento digital de sinais, pode ser custoso em termos da quantidade de passos de tempo a serem levados em consideração.

Outras referências classificam sistemas como uma coleção de processos digitais comunicando-se assincronamente (por exemplo, [JER 91] e [VAH 91]). Este é um modelamento cuja implementação apresentada é restritiva. O sistema é uma comunicação entre máquinas que cooperam para realização de alguma função, mas todas as máquinas são fisicas, quando algumas poderiam ser interpretadas (SW).

SegundoVahid, em [VAH 92], dada uma descrição comportamental de um sistema, o objetivo geral é sua implementação em HW, SW ou uma combinação de ambos. A partição comportamental de um sistema pode atingir um ou mais objetivos. Por exemplo, deseja-se utilizar uma tecnologia específica, como CMOS ou GaAs, processadores off-the-shelf ou Asics; o compromisso é entre desempenho, factibilidade e custo de manutenção e produção. A partição entre HW e SW é estudada neste item.

Além dos objetivos mencionados, a partição é útil também para descobrir se o sistema poderá ocupar o espaço definido de um encapsulamento, pinagem e potência. Tecnologias diferentes terão restrições diferentes quanto a área da cavidade, pinagem e dissipação térmica. Finalmente, definir o número de processadores é também tarefa da partição, visto que mais processadores (dedicados ou off-the-shelf) podem aumentar o paralelismo potencial, mas a comunicação pode ser um problema. Os objetivos acima descritos podem estar (e geralmente estão) interrelacionados durante o desenvolvimento do projeto do sistema.

O artigo de Vahid reflete uma tendência atual dos ambientes de pesquisa em síntese de alto nível (High Level Synthesis - HLS): a preocupação com síntese e agrupamento de computações, em oposição a meras operações. Já não se discute se o somador deve estar nesta ou naquela Parte Operativa, mas sim se este processador realiza esta ou aquela parte do algoritmo. Todo este processo contudo está ancorado no uso de HW ao invés de SW como meio de implementação, por ter sido este o caminho natural da realização de ASICs até então.

Em [JER 91] é apresentada a linguagem MV, ou Meta-VHDL. Nesta linguagem busca-se modelar sistemas eletrônicos como um conjunto de controladores e máquinas comunicantes. O modelo de sistema é um conjunto de processos concorrentes, cada processo sendo bastante complexo do ponto de vista das operações que executa.

Aplicações destes sistemas seriam micro-controladores, sistemas de tempo real e redes de telecomunicações.

Uma descrição MV é um conjunto de FSM cooperantes e hierárquicas. MV é feita sobre VHDL, tendo sido adicionadas primitivas de modo a que fosse facilitada a descrição de tais máquinas. Embora isto seja possível em VHDL diretamente, MV esconde certos detalhes de implementação.

Uma idéia semelhante é apresentada em [VAH 91], onde a linguagem é SpecCharts. A linguagem também é voltada para facilitar a entrada e modificação de controladores de alto nível, possuindo uma interface gráfica compatível com definições mais comuns de FSM. A linguagem preocupa-se em explicitar a comunicação entre os controladores complexos, definindo transições síncronas ou assíncronas, entre estados de máquinas internas e a comunicação entre as máquinas controladoras no alto nível.

É comum que HDLs usem a comunicação entre processos como dados comuns (sinais globais) ou como portas. Mas isto são as maneiras de comunicação; o comportamento da comunicação é deixado ao encargo do usuário. Vahid propõe o uso de duas novas construções, o canal e o protocolo. Um protocolo é um conjunto de portas e um comportamento definido destas portas. Define-se depois o canal e o protocolo ao qual ele deve obedecer.

O ponto comum entre as duas aproximações é que o conceito de sistema é o de um arranjo de máquinas complexas, cuja capacidade computacional individual seja dedicada a uma parte do problema, e a comunicação entre elas é que coordena as diferentes tarefas do sistema. Em nenhum momento nos trabalhos vistos discute-se possibilidades de deixar algumas tarefas ao encargo do SW, ou seja, um sistema é sempre um HW complexo, não uma associação HW-SW.

O trabalho de Wollan ([WOL 93]) apresenta um processo de projeto mais voltado para sistemas computacionais. Existe em biblioteca um core processor full custom rodeado de ROM, RAM e macroblocos configuráveis. Existe um ambiente de SW para desenvolvimento do processador, cujo modelo está disponível em VHDL. O micro tem 34 instruções lógicas e aritméticas, além de um número variável de portas de I/O e interrupções. O objetivo de Wollan é possibilitar a implementação rápida de sistemas microprocessados. Contudo, restringe-se o sistema a diferentes instâncias de um mesmo microprocessador.

Existe espaço nos ambientes de projeto atuais para novas definições e abordagens no que tange o projeto de sistemas computacionais. O modelo simples utilizado até agora em ambientes de HLS de que um sistema é composto de HW não resolve problemas complexos como, por exemplo, o projeto do controlador de uma planta industrial com uma série de interfaces com o operador. Não vale a pena descrever todo o processo em VHDL e sintetizá-lo, visto que seria mais econômico implementar muitas funções em SW.

Por outro lado, existem pesquisas na direção de sistemas formados por máquinas de HW comunicantes. Apesar de gerais no que tange a implementação de HW, estes modelos não abordam casos concretos de projeto conjugado de SW e HW e do compromisso entre ambos.

## 2.3 Trabalhos correlacionados

Recentemente, a preocupação no desenvolvimento integrado de HW e SW recebeu atenção do ponto de vista metodológico através de uma série de publicações. Um dos primeiros artigos a tratar do assunto foi certamente o de Gupta e de De Micheli ([GUP 92]), sobre síntese de sistemas utilizando HW reprogramável, isto é, microprocessadores e FPGAs.

A idéia central do artigo em questão é a de que um sistema é um misto de HW e SW. Certas funções de um ASIC podem não ser significativas em relação ao tempo total da tarefa, e portanto podem ser executadas em SW, sobre um HW reprogramável. Uma descrição inicial do sistema é particionada em SW e HW, para depois a parte de HW ser mapeada em FPGAs, GAs ou mesmo circuitos full-custom.

No trabalho de Gupta e De Micheli ressalta-se que a tarefa de implementação de sistemas pressupõe as seguintes atividades:

a)modelamento das funcionalidades do sistema;
b)determinação da fronteira entre as tarefas de um ASIC e aquelas de um microprocessador standard;
c)especificação e síntese da interface entre o HW e o SW, assim como dos mecanismos de sincronização;
d)implementação de rotinas de SW que respondam em tempo real às necessidades dos módulos de HW que executam concorrentemente.

Um sistema é descrito como um conjunto de processos paralelos na linguagem HardwareC ([DeM 90]). Estes processos são então analisados por uma ferramenta que monta um grafo representando os caminhos de dados dos processos. É então feita uma partição do grafo para alocação de tarefas em HW e SW.

Uma visão similar do conceito de sistema e de HW-SW co-design é apresentada em [MAR 93], por Marwedel e Schenk, que especificam os caminhos de projeto no ambiente Mimola. Neste caso, o ambiente de síntese de alto nível favorece estruturas de HW programáveis. Um sistema também aqui é visto como um HW que interpreta a especificação comportamental de um algoritmo. Uma descrição comportamental do HW (sistema) em uma linguagem tipo Pascal é a entrada do ambiente. Esta descrição pode ser implementada diretamente como uma máquina de estados finita de controle mais uma Parte Operativa, ou pode ser interpretada por uma máquina microprogramada.

Um ponto comum aos trabalhos acima discutidos é que o projetista do sistema deve descrevê-lo como um conjunto de processos. Desta descrição será obtido o HW e o SW a ser executado em alguma máquina programável, como um processador standard (Gupta, De Michele) ou um processador criado em função da descrição do sistema (Mimola). Ao se tentar ampliar as funções do sistema implementado, mais descrições em HardwareC ou Pascal-like serão necessárias, quando uma alternativa mais cômoda poderia ser descrevê-las em C e compilá-las para o processador alvo. Este ponto advém do fato que Gupta e De Michele preoucupam-se em implementar HW, que era a proposta original do sistema Olympus. O SW no caso é apenas uma maneira de economizar algumas funções no ASIC alvo. Consideração semelhante é válida para o sistema Mimola, onde o objetivo era a síntese de HW a partir de descrições alto nível.

A proposta mais voltada a um projeto de sistemas baseado em SW que é interpretado por um HW é feita por Athanas e Silverman em [ATH 91]. Neste artigo apresenta-se um protótipo de sistema de computação adaptativo (PRISM), baseado em um processador comercial e um array de FPGAs. A especificação do sistema é feita através de um programa C. Este programa é analisado e as partes críticas para o desempenho do algoritmo são candidatas a implementação em HW através de um conjunto de FPGAs. Um sintetizador é então chamado para realizar fisicamente a função. Considera-se crítica uma rotina que execeda um tempo limite para realização do algoritmo.

No trabalho em questão o compilador deve ter inteligência para decidir quais partes do algoritmo são críticas, visto que os processadores de uso geral não são feitos para terem ótimo desempenho para todos os tipos de aplicações. Assim, identificando-se as partes do SW para as quais o processador não teria bom desempenho pode-se transferir estas módulos para o HW. No momento atual, segundo reportado, as rotinas que devem ser implementadas em HW devem ser indicadas pelo usuário do ambiente.

Obviamente, este método tem problemas. A comunicação entre o processador e o HW dedicado é delicada, e limita o tipo de função em HW dedicado que pode ser implementada. Até onde se tem notícias de publicações ([ATH 93]), PRISM apenas implementa HW dedicados que ocupem um ciclo de relógio. Máquinas complexas não podem ser portanto transferidas do SW para o HW.

Em [LIN 91] o sistema Fastchart é apresentado. Nesta proposta um sistema de tempo real é implementado em HW. Para tal, a previsibilidade do tempo gasto em cada tarefa é importante. Assim, o processador principal não possui pipeline, nem cache nem outros recursos que aceleram a computação, mas são imprevisíveis no tempo de execução. A maneira com que Fastchart implementa sistemas de tempo real foi a transferência de funções da CPU para HW dedicados através de periféricos a esta. A CPU executa uma instrução por ciclo sem acesso a memória, e ocupa dois ciclos para instruções que acessam a memória, de modo a manter a previsibilidade de execução.

Neste caso, um sistema ainda é um SW mais HW, mas foi dada a devida importância à previsibilidade do HW em termos de execução das tarefas descritas em SW. Além disto, tem-se o conceito de transferência para HW externo ao processador do que não é fundamental para o desempenho do sistema em uma determinada aplicação de tempo real. Tem-se praticamente um sistema configurável segundo a aplicação.

Em [ERN 93] é apresentado o sistema COSYMA de síntese conjunta de HW e SW. Assim como no sistema PRISM, a estratégia é partir de uma descrição de um algoritmo e se chegar a uma função de HW dedicada. Diferentemente do PRISM, COSYMA remove a restrição da função em HW ocupar apenas um ciclo do processador que executa o SW. O algoritmo é analisado tendo em vista as rotinas críticas e, então, através de um algoritmo de simulated annealing, decide-se o que deve ser movido para HW.

O processador alvo é o Sparc, e o HW dedicado é sintetizado através do sistema Olympus ([DEM 90]). Uma limitação do sistema COSYMA advém do fato que a CPU é colocada em hold no instante em que a função em HW está trabalhando. Assim, a quantidade de paralelismo limita-se àquela encontrada no algoritmo movido para o HW.

A aceleração para os exemplos reportados é de 1.4 e 1.3, embora um exemplo tenha resultado em uma aceleração de 0.9, portanto mais lento que no processador original. As origens deste fraco desempenho encontram-se provavelmente na partição estática do código e no acesso à memória.

## 2.4 A proposta do ambiente de desenvolvimento de sistemas

Um sistema, pelo anteriormente exposto, será modelado como um conjunto de funções sendo executadas em um par HW e SW. Isto porque um processador nada mais é do que um interpretador de uma certa linguagem (do conjunto de rotinas de SW). Na maioria das aplicações supõe-se o uso de um processador comercial.

Existem razões de mercado para esta hipótese. A primeira diz respeito a facilidade com que se consegue obter processadores de alto desempenho por baixo custo, por exemplo, o Motorola 88000, o Texas C40 com unidade de ponto flutuante, o próprio Intel 486, Pentium e outros, mesmo microcontroladores com multiplicador. A segunda diz respeito ao tamanho dos sistemas atuais. Cada vez mais funções são implementadas, mas nem todas são críticas com respeito ao tempo de execução. Portanto, o HW padrão de um microprocessador interpretando as funções tem tempo de execução aceitável para muitas aplicações. O uso de ASICs dedicados justifica-se para partes críticas do sistema. Além disto, quanto menor o processador, maiores as possibilidades de seu uso como processor-núcleo integrado.

A ênfase dada para implementação em SW ou HW das partes que compõe um sistema é função da aplicação final do mesmo. Deve-se portanto fornecer um ambiente integrado para que o desenvolvimento concorrente de HW e SW aconteça. Além disto, procura-se otimizar e utilizar o parque industrial já existente, que faz o uso de microcontroladores, ferramentas de SW como montadores, sistemas de desenvolvimento e outras. Isto permitirá uma progressão contínua do engenheiro de sistemas para o uso das técnicas de microeletrônica, e uma efetiva difusão de técnicas modernas de projeto ao parque industrial instalado. Desta maneira, a vida útil de um projeto em uma empresa pode ser ampliada, visto que a adição de um HW dedicado permite uma atualização de funções sem necessidade de grandes retreinamentos do pessoal disponível.

É preciso também ressaltar que as ferramentas de síntese atual permitem pouca ou nenhuma cooperação entre HW e SW. Por exemplo, de nada adianta descrever-se um novo processador em VHDL e submetê-lo à síntese automática, se o SW para este novo processador não existir. Muito investimento durante o projeto de sistemas é feito para produção de SW.

Nesta seção será apresentado o Sistema ECSD, Environmet for Computational Systems Development. Discute-se a especificação do sistema e a entrada no ambiente de CAD, seguido de uma visão geral das técnicas para otimização de desempenho entre HW e SW. Este sistema foi pensado não somente tendo em vista a otimização dos sistemas computacionais atualmente disponíveis, mas também daqueles futuros, quando se espera que o acesso a tecnologias modernas de integração será ainda mais eficiente que atualmente.

### 2.4.1 Especificação do sistema computacional a ser implementado

Pela hipótese inicial, o sistema computacional deve estar descrito como um conjunto de rotinas de um processador. Algumas tarefas típicas de implementação de alto nível como particionamento de tarefas entre diversos micros, decisão sobre as partes analógicas e discretas já devem ter sido tomadas.

A entrada para o ambiente ECSD é um conjunto de rotinas em uma linguagem de programação. Já aqui deve-se aplicar algumas perguntas referentes à interpretação destes programas por algum HW. Por exemplo, pode-se fazer uma análise e descobrir qual o melhor processador, a partir de um conjunto de processadores, para resolver tal problema. No caso, particionar um problema entre HW e SW significa também descobrir qual o melhor processador para executar o conjunto de algoritmos propostos.

Dado um sistema, pode-se analisá-lo segundo dois métodos. No primeiro caso, busca-se analisar o código para verificar qual o HW que deve ser agregado a um HW padrão para melhora de desempenho. No segundo caso, pode-se analisar o algoritmo e verificar qual o processador que melhor se adaptaria ao conjunto de algoritmos.

Este segundo tipo de análise, embora interessante, não será objeto deste estudo. A opção por um certo processador é determinada não somente por fatores de desempenho (onde poderia haver muitos parâmetros idênticos, pesquisando-se diversos fabricantes), mas também por fatores como custo, disponibilidade no mercado, facilidade de migração de SW e treinamento dos projetistas. No caso, espera-se que a definição do processador a ser utilizado seja feita antes da aplicação dos métodos aqui explicados.

Se o processador escolhido é um dedicado, isto é, proprietário do engenheiro de sistema, tem-se mais graus de liberdade. Algumas escolhas possíveis são o número de bits, o conjunto de instruções, a própria arquitetura, o uso de co-processadores numéricos, etc. Se o processador é standard, algumas facilidades estão fixas, e deve-se portanto buscar HW dedicados na forma de periféricos ou de interfaces que se comuniquem via barramento principal ou através da memória (ou outros protocolos, como interrupção).

Uma linguagem de programação normal como C, Pascal ou outra tradicional para desenvolvimento de software não é necessariamente uma boa linguagem de especificação de sistemas computacionais. Por exemplo, descrições de processos paralelos são impossíveis, bem como detalhes sobre o I/O e pinagem da placa alvo não existem. No contexto deste trabalho, porém, a divisão de tarefas entre processadores já foi realizada, e existe um processador responsável pelas execuções das rotinas. As razões desta escolha advém do interesse em se estudar as relações entre o processador e o conjunto de programas que este deve executar. Imagina-se que este tipo de análise sempre estará presente no desenvolvimento de sistemas. Especificações de mais alto nível serão úteis quando houver um ambiente de CAD sobre o qual se basear para investigação de diferentes soluções.

Após a escolha do processador, deve-se compilar o código e otimizá-lo através da eliminação de subexpressões comuns, movimentação de invariantes para fora de laços, eliminação de variáveis induzidas, etc. Estas técnicas são utilizadas na maioria dos bons compiladores atuais ([AHO 88]). Portanto, o código assembler já está otimizado para uma certa arquitetura de máquina alvo, o processador em questão.

Pelo acima exposto, torna-se interessante otimizar o código assembler para um subconjunto de funções críticas a serem executadas em um dado processador. Isto porque, caso o programa assembler não execute na velocidade desejada, estará provado que o processador (custom ou standard) chegou ao seu limite e, portanto, é ali que se deve atuar para aumentar o HW do sistema. Dito de outra maneira, é naquele trecho específico que se deve aumentar a capacidade de interpretação do microprocessador. Ao aceitar-se o código assembler como entrada, aproveitam-se todas as transformações já existentes nos compiladores. O assembler é uma descrição comportamental de um certo algoritmo, com os recursos computacionais fixos (registradores, ULAs e barramentos já se encontram especificados no próprio código e no micro que interpreta este código).

### 2.4.2 O desenvolvimento de um projeto em ECSD

Esta sessão procura mostrar o desenvolvimento de um projeto dentro do ambiente ECSD (Environment for Computational Systems Development). A figura 2.2 mostra o caminho entre a especificação do sistema através de um programa C, apresentando a metodologia de projeto e um conjunto de ferramentas que suportam esta metodologia.

A linguagem C foi escolhida tendo em vista sua popularidade e grande disponibilidade de ferrramentas. Foi adaptado o compilador GNU ([STA 89]) para produzir código para o processador Risco ([JUN 93]). O Risco é um processador Risc de 32 bits, desenvolvido no CPGCC da UFRGS. Uma breve descrição de sua arquitetura e de seu conjunto de instruções encontra-se no anexo 1. Após a compilação, é feita uma seleção das rotinas candidatas a serem transportadas para HW. A rotina escolhida é então classificada para que sua síntese (automática, virtual ou manual) seja direcionada. cada uma destas tarefas será explicada nos próximos capítulos. Uma vez feita a reorganização das instruções, necessária pela inserção de HW ou por outra otimização, obtem-se o máximo paralelismo entre o HW dedicado e o processador, seguidas de simulações para validação do sistema como um todo. Se o sistema não atende os requisitos iniciais, pode-se retornar a diversos pontos para novas iterações de projeto.

### 2.4.3 Seleção da rotina crítica

No que tange a implementação do sistema, a necessidade de um HW dedicado surge quando o processador não mais responde no tempo necessário para uma determinada aplicação. A rotina que toma mais tempo de execução será considerada crítica, pois é a que mantém o processador mais ocupado. Atuando sobre a rotina crítica é provável que a a otimização do sistema seja mais significativa. Portanto, a descoberta da rotina crítica é parte fundamental do processo de otimização de sistemas computacionais.

A seleção das rotinas candidatas é feita através de uma análise de custos relativos ao número de vezes que a rotina é chamada ([PAR 91]). Do código assembly montado e da tabela de símbolos percorre-se o grafo do programa, montando-se uma árvore de chamadas de rotinas. Esta técnica será explicada em detalhes no capítulo 3 mas, basicamente, tenta-se através de uma análise estática do código de entrada determinar uma série de candidatas a rotina crítica do sistema.

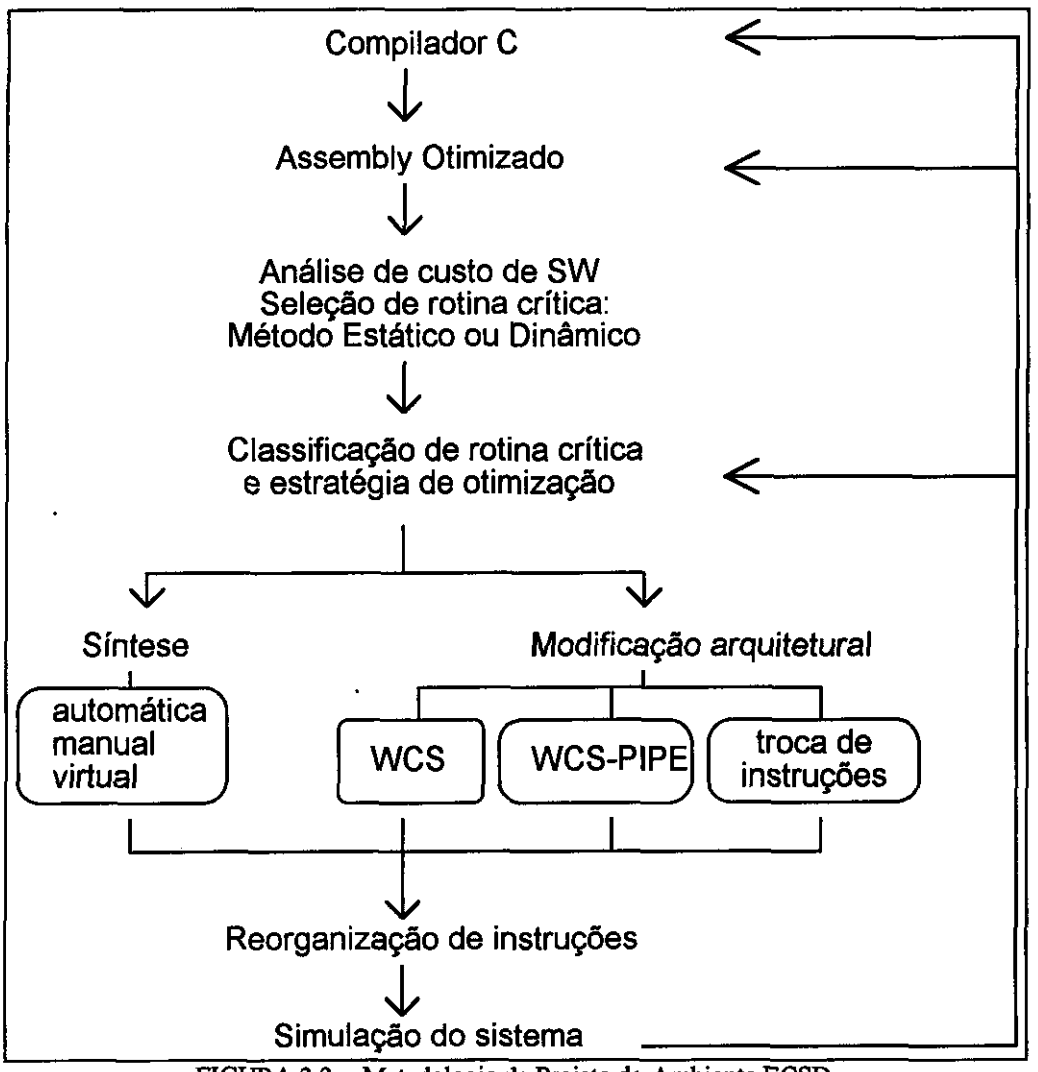



FIGURA 2.2 - Metodologia de Projeto do Ambiente ECSD

Este método é propenso a erros, pois a grande maioria de programas é dependente do conjunto de dados de entrada. A análise estática mesmo sendo complexa pode levar a resultados enganosos, como demonstrado em [ERN 93]. Neste trabalho iniciou-se com uma análise estática tendo em vista a simplicidade de implementação e a disponibilidade de ferramentas. Uma vez obtidas técnicas de análise de código mais refinadas podem-se substituir os métodos estáticos aqui apresentados sem prejuizo dos demais conceitos utilizados no trabalho.

Outra situação possível é permitir que o usuário execute o programa com uma série de testes. Para estes testes avaliam-se os custos de execução para o conjunto de rotinas dependentes de dados. Este tipo de técnica é chamado de profiling pela comunidade de HW-SW codesign. Apesar de criticável por ser largamente dependente do conjunto de dados utilizados, seu ponto forte é exatamente encontrar o conjunto de rotinas críticas para uma certa aplicação. Como o objetivo é a realização de um sistema dedicado à uma aplicação, a limitação conceitual do profiling praticamente não é significativa. Uma ferramenta para obtenção do profiling do processador Risco é apresentada no capítulo 6. Para outros processadores o uso desta técnica é dependente do conjunto de ferramentas encontrado no mercado. Para os processadores DLX e SPARC o profiling encontra-se disponível através de seus simuladores ([HEN 90], [SUN 91]).

De qualquer modo, não é descartada a hipótese de que será necessária uma intervenção do usuário para selecionar qual dentre as possíveis candidatas é aquela a ser tranformada em HW. Uma expansão futura na entrada do sistema seria permitir que a linguagem de especificação de entrada (assembly do processor alvo ou C) possuisse um conjunto extra de primitivas relativas ao tempo máximo ou mínimo de uma ação, como sugerido em [DAS 85]. Um conjunto de restrições temporais seria então um dado a mais para auxiliar na detecção das partes críticas de um programa. Na verdade, a especificação de restrições temporais é provavelmente anterior à escolha do algoritmo. A passagem destas restrições ao sistema de CAD deve ser levada em conta para que na partição HW SW este seja um item analisado. Além disto, pode-se atingir a otimização de sistemas tendo em vista as restrições temporais, como hoje se faz a síntese lógica tendo em vista requisitos de atraso do usuário.

### 2.4.4 Classificação das rotinas.

Uma vez selecionada a rotina crítica ou as candidatas, monta-se um flow-graph para se detectar a característica básica. O flow-graph é um grafo dirigido onde as dependências entre blocos básicos de um programa estão explicitadas. As rotinas são então classificadas em 3 categorias: memória, condicionais e computacionalmente intensivas. A idéa é localizar-se o aspecto mais crítico de uma rotina em respeito ao desempenho. Para realização desta análise, técnicas tradicionais usadas em compiladores como ordenação do flow-graph e levantamento de grafos de dependência de instruções são utilizadas ([AHO 88]). Para cada rotina o conjunto de blocos básicos que a constituem é determinado, e a partir da análise destes blocos procede-se aos demais passos, como será verificado no próximo capítulo.

Se o número de blocos básicos de uma rotina é grande, e cada bloco básico tem poucas instruções, então esta é provavelmente uma rotina do tipo condicional. No caso, poucas instruções não permitem uma otimização do grafo de dependências ou indicam um grafo com pouco paralelismo potencial disponível. No caso do processador Risco, 6 é o número de operações limite, visto que para um salto condicional são necessárias 3 instruções, restando apenas 3 instruções para computações.

Rotinas condicionais possuem muitos saltos, e cada salto provoca uma quebra (stall) do pipeline do processador. No caso do Risco, nops são inseridos, e portanto o desempenho global diminui. A simples transformação de instruções de jump em um HW dedicado não é um procedimento que assegure bom desempenho. Um salto condicional pode requerer que dados da parte operativa do processador estejam disponíveis para operaçõs na Unidade Lógica e Aritmética (subtração, soma ou mascaramento, por exemplo), de modo a decidir-se a direção do salto. A realização de *multi-way jumps* ([PAP 93]) só é possível com máquinas VLIW (Very large Instruction Word) ou microprogramas, mas no caso tem-se de mudar o firmware do processador. A transposição de toda a rotina para HW pode ficar prejudicada pela necessidade de acesso à memória, bloqueando o microprocessador. Uma solução que atua sobre as limitações acima descritas é proposta pela modificação da arquitetura do processador, e será explicada no decorrer do capítulo 4.

Para rotinas com muitos acessos à memória, o gargalo é o próprio acesso à memória. Uma solução possível é colocar-se a parte da memória em uso pela rotina dentro do próprio HW que executará a rotina. Esta é uma solução excelente para os

casos em que a memória a ser utilizada é pequena e possivelmente usada apenas naquela rotina. Um exemplo de tal comportamento seria a tabela de arcotangentes utilizada no algoritmo Cordic para cálculo de funções trigonométricas ([VOL 59]).

Uma rotina mais geral porém provavelmente será chamada uma vez com um conjunto de dados e outra com outro, e portanto o tamanho da memória aumenta. Quanto maior a memória, maior o custo do HW dedicado, e mais lenta e cara a memória local. Além disto, se a memória da função em HW é colocada no espaço de endereçamento do processsador (e o deve ser, para poder ser carregada), esta deverá ter possivelmente um duplo acesso, seja do processador seja da função de HW. No capítulo 4 será apresentada uma possível solução para o problema da otimização de rotinas com muito acesso à memória.

As rotinas computacionalmente intensivas são caracterizadas por um uso intensivo de operações na PO do processador, ou seja, existem poucos blocos básicos e muitas operações em cada bloco básico. Se a rotina deve ser transportada para HW, então deve-se ter um ganho nesta movimentação, ou seja, a rotina deve executar em menos ciclos que no processador ou deve-se ter paralelismo entre o processador e a rotina em HW. Para tal, é preciso sintetizá-la e avaliar seu novo custo em termos de ciclos de máquina do algoritmo.

### 2.4.5 Síntese da rotina candidata

Como já mencionado, uma vez selecionada uma rotina crítica pode-se sintetizá-la para verificar seu custo em HW. Planeja-se a síntese como automática, virtual ou feita pelo usuário. Na síntese automática deve-se gerar a partir do assembly da rotina uma descrição compatível com o sistema de síntese em uso (VHDL, para o sistema Synopsys [CAR 90], HardwareC, para o sistema Olympus [DeM 90], AHDL para o sistema da Altera [ALT 93], etc). Na descrição os protocolos de iteração entre o microprocessador e a função em HW estarão já especificados.

Nem sempre contudo deseja-se uma síntese completa. Pode-se por exemplo estimar o custo em ciclos de uma certa rotina, apenas para verificar a validade de sua passagem para HW ([SHA 93]). Além disto, o número de ciclos da rotina pode ser levado em conta depois, no instante em que se tenta obter paralelismo entre as operações em SW e em HW. Com a facilidade dos sistemas de síntese de alto nível atuais, este pode ser um passo altamente iterativo, onde se pode trocar o custo em HW por mais operações paralelas do microprocessador e do HW dedicado. Este tipo de atividade de estimativa considera-se síntese virtual. A rotina não é realmente sintetizada, tem-se apenas uma idéia do número máximo de ciclos a ser utilizado, sem considerações quanto ao custo de implementação. No caso, utiliza-se o algoritmo de alocação ASAP, As Soon As Possible.

O custo da função de HW tendo em vista o conjunto dos módulos utilizados presentemente não é levado em conta nos passos de síntese virtual. Como a especificação das operações da função de HW são derivadas do assembler do microprocessador, está claro que o custo do HW dedicado será função do paralelismo disponível nos blocos básicos do algoritmo a ser sintetizado.

Por fim, não se pode descartar a síntese manual, visto que somente ela pode ser criativa. Por exemplo, trocar um algoritmo de divisão de subtrações sucessivas por um de Newton pode ser feito somente pelo operador humano.

### 2.4.6 O ambiente de validação

Para o desenvolvimento de sistemas computacionais é comum o uso de sistemas de desenvolvimento, onde existe um emulador do processador a ser utilizado e de suas memórias, junto com compiladores de linguagem alto nível e assembly.

Um simples sistema de desenvolvimento não é suficiente para suporte do ambiente integrado aqui proposto. É preciso considerar que o HW do sistema será modificado pela introdução de funções especiais antes não existentes, que devem ser simuladas junto com o processador executando trechos de programa.

Existem presentemente ótimas ferramentas de desenvolvimento de ASICs, mas muito pouco tem-se publicado sobre a simulação de sistemas voltados para o projeto conjunto de HW e SW. Além deste problema, deve-se notar que o compilador-otimizador altera rotinas fonte do usuário. Portanto, a correção de erros do sistema pressupõe que o usuário esteja a par das modificações introduzidas pelo compilador, sob pena da rotina não ser mais compreensível.

A arquitetura do processador Risco foi validada através do simulador de código compilado SHC ([MAR 92], [CAR 93]). Sobre este simulador construiu-se um sistema de desenvolvimento capaz de aceitar código Risco como entrada, sendo a máquina de simulação a mesma presente no sistema SHC.

Para permitir a presença de funções dedicadas em HW na mesma descrição seria conveniente um simulador de barramento de Risco. No caso, este simulador estaria encarregado de coordenar diferentes máquinas de simulação através do barramento do processador. A idéia é que os diferentes eventos sejam reportados ao barramento. Como este é o meio de comunicação, os processos de simulação esperam uma atividade no barramento para verificar se devem ou não atuar. Assim, por exemplo, enquanto o processador é simulado em SHC, a função de HW poderia ser simulada com o simulador lógico SLX, ou com o elétrico STX ([CAR 93a]).

Outra alternativa de co-simulação e desenvolvimento seria a disponibilidade da descrição VHDL do processador e de seu HW dedicado. Contudo, a simulação de um algoritmo complexo no processador tomaria muitos ciclos da máquina de simulação VHDL, o que poderia inviabilizar grandes projetos, que são exatamente o alvo de HW/SW codesign.

O barramento foi escolhido como sendo o centralizador de atividades já na prevista migração do sistema virtual hoje existente para um sistema de desenvolvimento real, onde o código é diretamente executado no processador real. Neste caso, embora a execução do SW seja rápida, a simulação do HW dedicado pode ser lenta. No capítulo 6 detalhes desta família de simuladores serão abordados.

# 3 Algoritmos para implementação de sistemas

Neste capítulo algumas técnicas de otimização da implementação de um sistema através da cooperação entre o SW e o HW de um microprocessador serão apresentadas. Para tal, descrevem-se os passos de particionamento entre HW e SW através da descoberta de uma rotina crítica, a classificação desta rotina tendo vista sua otimização e a síntese da mesma para o caso desta ser computacionalmente intensiva. Além disto, os algoritmos capazes de explorar o paralelismo possível entre o SW e o HW dedicado são também apresentados. Todos os algoritmos aqui descritos têm aplicação em qualquer ambiente microprocessado atual. Embora apenas uma rotina crítica por vez seja analisada, o sistema é geral em termos da arquitetura alvo, isto é, em qualquer ambiente com microprocessador podem ser utilizadas as técnicas aqui descritas.

## 3.1 O particionamento HW/SW

Em algum instante do processo de projeto é preciso decidir a quantidade do sistema que será SW e o que será HW. Em ECSD parte-se do pressuposto que o sistema será implementado primeiro em SW, e as partes críticas do mesmo serão passadas para HW. Esta estratégia é similar àquelas utilizadas em [ATH 93] e [ERN 93]. A diferença fundamental entre ECSD e os trabalhos acima mencionados está no fato que se utilizam diferentes estratégias dependendo do tipo de rotina crítica, e a otimização de rotinas computacionalmente intensivas se dá não somente pelo uso de uma função de HW dedicada, mas pela obtenção do paralelismo entre o SW do microprocessador e a função de HW dedicada.

O particionamento entre HW e SW é feito pela detecção das rotinas críticas que devem ser otimizadas. A seleção das rotinas candidatas é feita através de uma análise de custos relativos ao número de vezes que a rotina é chamada ([PAR 91], [SMI 81]). Do código assembly montado e da tabela de símbolos percorre-se o grafo do programa, montando-se uma árvore de chamadas de rotinas. Por exemplo, imagine-se uma rotina que calcula uma integral digital que chama recursivamente as rotinas de multiplicação e divisão. Se o programa principal chama o filtro dentro de um comando "for", tem-se a árvore da figura 3.1.

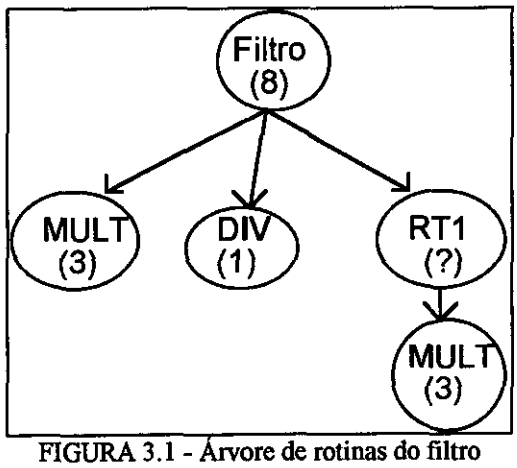



FIGURA 3.1 - Árvore de rotinas do filtro

Como o filtro é chamado 8 vezes e a multiplicação é chamada 2 vezes, o custo total da multiplicação é de 16. Isto foi possível avaliar porque o código possui constantes que permitem determinar o tempo de execução durante a compilação ou montagem.

Sobre a rotina RT1, nada se pode dizer. Embora se saiba que a rotina de multiplicação foi chamada 3 vezes em RT1 (rotina de ajuste de dados), o número de vezes que RT1 é chamada depende dos dados de entrada do programa, sendo desconhecidos no tempo de compilação. Portanto, deve-se fazer uma estimativa (heurística) do tipo de dado e do custo total da multiplicação. Por exemplo, pode-se grosseiramente dizer que a rotina dependente de dados tem peso 1, e portanto o custo total da multiplicação seria 16+3=19.

Quanto maior o custo de uma rotina, maior a chance que ao otimizá-la se chegue a um HW mais rápido (descontando-se aquelas rotinas que o próprio usuário indicou como críticas). É preciso tomar cuidado com a heurística porém. O número bruto soma os custos de rotinas avaliadas em tempo de compilação e aquelas avaliadas em tempo de execução e, portanto, pode-se ser enganado. É interessante fornecer-se ambos os custos. O usuário, em face ao tipo de dado, por exemplo, pode escolher qual rotina otimizar. Se for desejado o profiling, pode-se utilizar o simulador do Risco para tal, como explicado no capítulo 6. Existe também a situação onde bastaria otimizar uma rotina não necessariamente crítica, mas que com sua otimização o algoritmo como um todo já atingisse as necessidades de tempo do usuário. A otimização com menor custo deveria ser a escolhida.

Trabalhou-se apenas com a análise estática para se ter idéia das diferentes soluções de implementação possíveis para o sistema. O uso de outros algoritmos de análise, inclusive profiling, não alteram a necessidade de otimização ou as estratégias a serem utilizadas, apenas eventualmente o conjunto de rotinas consideradas críticas será diferente. A análise estática neste trabalho compreende os seguintes passos:

a)montagem dos blocos básicos por separação de saltos, como em [AHO 88];
b)descoberta dos caminhos de tempo fixo, determinados durante a compilação;
c)avaliação dos custos de cada caminho.

Uma vez descobertos os caminhos cujos tempos de execução podem ser avaliados durante a compilação procede-se à descoberta do tempo total de cada caminho. Para tal utiliza-se um algoritmo de propagação de constantes e de detecção de laços (conforme algoritmo exposto no anexo 13).

Este procedimento é feito para cada rotina, e após monta-se a árvore de rotinas. Com esta pode-se determinar o custo total de um programa, pela descoberta do custo de cada rotina sendo multiplicado pelo número de vezes que esta é chamada. Utiliza-se um procedimento depth-first para percorrer o grafo de rotinas até uma rotina folha. A cada visita aos nodos é calculado o custo da rotina multiplicado pelo custo da rotina pai, a rotina chamadora. Ao final da pesquisa todas as rotinas folhas terão seu custo comparado para determinar a rotina crítica. Para cada rotina é fornecido o tempo de execução fixo (determinável em tempo de compilação ou montagem) e o custo variável, determinado pelo conjunto de dados do sistema computacional. No caso de duas ou mais rotinas com o mesmo custo, escolhe-se aquela que foi chamada o maior número de vezes de maneira dependente de dados.

As técnicas acima descritas foram implementadas no montador universal para a família Risco ([LIM 94], [SUZ 90], [JUN 93]). Este montador, além de permitir a montagem de um programa para simulação realiza a análise das rotinas conforme explicado, não só para o Risco mas para toda a família ([SOU 95]), conforme será

mencionado nos capítulos 5 e 6. A figura 3.2 mostra a saída do montador para um programa que calcula o resultado de um filtro digital.

```
Rotina: mult
Chamada deterministicamente: 500 vezes
Basic blocks: 3
Maior caminho: 3
Caminho: 1 2 4
Total de computacoes: 27764
Total de acessos a memoria: 28257
Total de JMPs condicionais: 11076
Total de JMPs incondicionais: 8293
Speed up para SINTESE: 1.410708
Speed up para WCS: 1.203127
Speed up para WCS-PIPE: 2.455046
```

FIGURA 3.2 - Saida do montador para o exemplo de filtro digital.

Presentemente a entrada para a detecção de rotinas críticas é o assembler do Risco. Na verdade, esta detecção pode ser feita em níveis mais abstratos, possibilitando a decisão sobre como otimizar o sistema em estágios mais iniciais. Por exemplo, em [PAR 91] reporta-se um sistema que é capaz de analisar um subconjunto da linguagem C, prevendo o tempo de execução de um programa e de suas rotinas. Cada operação é mapeada para o assembler da máquina alvo, mesmo sem ser otimizado. Assim, estima-se o tempo mínimo e máximo de execução. Embora receba C como entrada, o sistema trabalha fortemente baseado no assembler do processador.

No caso do sistema proposto neste trabalho, decidiu-se pelo assembler, pois a parte de otimização do compilador é muito significativa em relação as arquiteturas Risc, tendo em vista a velocidade de execução do código. Assim, é sobre o código otimizado que se deve estudar as rotinas potencialmente críticas.

## 3.2 Classificação de rotinas

Uma vez determinada a rotina crítica, tem-se um ponto de partida para a otimização do sistema. Para tal, a estratégia será identificar na rotina algumas características marcantes, que poderão sinalizar para a estratégia mais significativa de otimização.

### 3.2.1 A classificação de rotinas tendo em vista sua otimização

Dividiu-se o conjunto de rotinas em três grandes grupos. Tendo em vista os tipos de operações mais comuns, ou de maior custo de processamento, as rotinas podem ser computacionalmente intensivas (muitas operações em um único Bloco Basico - BB), condicionais (muitos if-else, redundando em muitos BB em um flow-graph), intensivas em memória (muitas instruções de load/store dentro de um mesmo BB) ou mesmo uma combinação entre todas as acima.

Esta classificação é particularmente importante para direcionar o algoritmo de síntese automática da rotina em questão. Uma rotina computacionalmente intensiva pode ser acelerada pelo uso de técnicas que exploram ao máximo o paralelismo, como software pipeline ([LAM 88]), tree height reduction ([NIC 91]), percolation scheduling ([AIK 88]) e outras ([GAJ 92]). Para rotinas com outro tipo de característica somente a síntese de HW não é satisfatória, como será apresentado.

A tabela 3.1 apresenta um resumo das possibilidades de orimização tendo em vista a característica fundamental da rotina crítica.

TABELA 3.1 - Resumo da classificação de rotinas e estratégia de otimização

| Tipo de Rotina | Característica | Método de Otimização | Tipo de Processador |
|---|---|---|---|
| computacional | muitas operações | síntese da rotina | normal |
| condicional | muitos desvios | mudança na arquitetura | dedicado |
| memória | muito acesso a memória | mudança na arquitetura | dedicado |
| mista | indefinido | otimização de arquitetura | normal ou dedicado |

### 3.2.2 A classificação pela regra de Amdahl

Em um artigo já clássico na computação paralela, Amdahl estabeleceu que o ganho máximo que pode ser obtido utilizando-se um modo de execução mais rápido para uma máquina é limitado pela fração de tempo que este modo pode ser realmente executado ([AMD 67]). Uma vez que cada rotina crítica está identificada, é preciso otimizá-la. Para tal será utilizada a regra de Amdahl modificada. Basicamente, tenta-se estabelecer a quantidade de ganho que se obterá no algoritmo quando se remove o motivo principal da rotina ser crítica. A medida que o ganho com esta remoção aumenta, a probabilidade de se utilizar uma estratégia coerente com este ganho aumenta.

A fórmula de Amdahl estabelece que o fator de ganho será dado pela divisão entre a parcela que se quer acelerar e o percentual da parcela em relação ao conjunto. Contudo, a aceleração global é dada pelo percentual que foi acelerado em função do novo custo após aceleração, visto que dificilmente se consegue diminuir a influência de um certo fator. Dito de outra maneira, se a otimização se refere a apenas uma parcela de um programa, então somente esta parcela será acelerada, e portanto a aceleração global será menor que a aceleração da rotina crítica simplesmente.

Em processadores de arquitetura Risc sem operações de ponto flutuante o custo de cada instrução é facilmente calculado. No geral as instruções de computação tem custo um, enquanto que as de acesso a memória limitam-se a instruções load-store, cujo custo varia com o número de barramentos disponíveis. Dependendo do número de estágios do pipeline, o custo dos saltos condicionais e incondicionais será variável.

Para validação do método de partição baseado na fórmula de Amdahl foram utilizados dois procesadores Risc, o Risco ([JUN 93]) e o DLX ([HEN 90]). A tabela 3.2 apresenta um resumo das diferentes características de cada processador, assim como os custos para grupos de instruções utilizados neste trabalho.

Pela tabela 3.2 pode-se observar que o processador Risco é mais econômico que o DLX, devido ao menor número de pinos, tendo em vista o número de barramentos externos. O custo de instruções computacionais é o mesmo nos dois processadores.

Devido ao pipeline maior do DLX, um salto condicional também vai possuir um custo mais elevado. O custo com salto WCS refere-se a uma otimização a ser apresentada no capítulo 4.

TABELA 3.2 - Comparação de custo de intruções entre o Risco e o DLX

| Parâmetro | Risco | DLX |
|---|---|---|
| # de barramentos externos | 1 (dados e instruções) | 4 (dados e instruções) |
| # de registradores | 30 + PC + status | 32 + PC |
| profundidade do pipe | 3 estágios | 5 estágios |
| custo inst. computacional | 1 ciclo | 1 ciclo |
| custo jump incondicional | 2 ciclos | 2 ciclos |
| custo jump WCS | 0 ciclos | 0 ciclos |
| custo jump condicional | 3 ciclos | 4 ciclos |
| custo condicional WCS | 1 ciclo | 2 ciclos |
| custo acesso a memória | 1 ciclo | 0 ciclos |

Uma rotina computacionalmente intensiva possui muitas operações por bloco básico. Quanto mais operações existirem em um bloco básico, maior a possibilidade de paralelismo no fluxo de operações, contando-se com restrições devido à dependência de dados entre instruções. Possivelmente se esta rotina for sintetizada com aproveitamento deste paralelismo o ganho em termos de desempenho será alto. Considera-se que a rotina manterá seu flow graph intacto, mas cada BB será reduzido a apenas uma computação. Isto é o mesmo que dizer que dentro de um mesmo BB todas as operações podem ser realizadas em paralelo, o que certamente nem sempre acontece. Esta avaliação é extremamente otimista, mas o objetivo é identificar o potencial desta rotina quanto à síntese. Uma rotina que receba baixo ganho segundo esta fórmula certamente quando de sua síntese real estará muito abaixo do esperado.

No caso de rotinas computacionalmente intensivas, o ganho será dado pela substituição de todos as computações por blocos básicos. Como o flow-graph é mantido, os saltos são considerados como transições de estado, e possuem custo 1. Seja C o custo de instruções computacionais (número de ocorrências destas vezes o custo de uma instrução tipo computacional), M o custo das instruções de memória, UJ o custo das instruções de saltos incondicionais enquanto que CJ é o custo dos saltos condicionais. A fórmula do fator de ganho é dada por

$$FG = \frac{C}{C+M+UJ+CJ} \quad (3.1),$$

enquanto a aceleração global é dada por

$$AG = \frac{1}{1-FG+\frac{FG}{\frac{C}{BB}}} \quad (3.2).$$

Na equação 3.1 computa-se o custo das instruções computacionais em relação a todos os outros custos de instruções da rotina. A equação 3.2 mostra a situação onde as

instruções computacionais são substituidas pelo número de blocos básicos presentes. A expressão *C/BB* significa qual será o ganho quando se perfaz esta substituição.

Para rotinas intensivas em saltos procede-se da mesma maneira. Calcula-se o efeito das instruções de salto no programa e a aceleração que seria obtida se estas instruções fossem substituidas por outras mais rápidas. No caso da equação 3.3 tem-se o efeito das instruções de salto, e portanto escreve-se

$$FG = \frac{UJ + CJ}{C + M + UJ + CJ} \quad (3.3),$$

enquanto que a equação 3.4 mostra a aceleração no caso da substituição destas instruções por outras de menor custo,

$$AG = \frac{1}{1 - FG + \frac{FG}{\frac{UJ + CJ}{UJn + CJn}}} \quad (3.4).$$

Para determinar se a rotina sob análise é caracterizada por muitos acessos a memória procede-se de maneira análoga, considerando-se contudo que somente acessos a memória são inúteis. Algo deve ser computado utilizando-se o dado acessado. Portanto, deve-se identificar a quantidade de operações com dados de memória em relação às operações internas ao processador. Isto significa que na fórmula de Fator de Ganho o termo C+M é considerado como sendo o termo a ser otimizado. Na verdade, esta equação já pressupõe a estratégia de otimização a ser utilizada, qual seja, aumentar o número de acessos à memória para mantê-la constantemente ocupada em fornecer novos dados ou armazenar resultados. Assim, escreve-se

$$FG = \frac{C + M}{C + M + UJ + CJ} \quad (3.5),$$

e

$$AG = \frac{1}{1 - FG + \frac{FG}{\frac{C + M}{D * 2}}} \quad (3.6).$$

O termo D*2 na equação 3.6 refere-se ao fato de que a cada acesso a memória duas operações podem ser executadas por um processador atuando em paralelo, visto que o acesso à memória no Risco tem custo de 2 ciclos (no DLX e outros microprocessadores esta fórmula deve ser corretamente ajustada). A variável D armazena o máximo entre o número de operações computacionais e o número de acessos à memória. A equação 3.6 portanto pressupõe uma otimização tendo em vista um HW paralelo, como será explicado no capítulo 4.

TABELA 3.3 - Resultado de classificações para Risco e DLX

| | RISCO | | | DLX | | | |
|---|---|---|---|---|---|---|---|
| **rotina** | **comp.** | **WCS** | **WCS PIPE** | **comp.** | **WCS** | **WCS PIPE** | **diff** |
| isqrt | 1.66 | 1.2 | 1.00 | 2.00 | 1.25 | 0.88 | |
| gaussian A/G mean | 1.28 | 1.25 | 1.00 | 1.41 | 1.33 | 1.00 | |
| divisão fracionária | 1.16 | 1.28 | 1.00 | 1.25 | 1.40 | 1.00 | |
| divisão inteira | 1.27 | 1.26 | 1.00 | 1.41 | 1.35 | 1.00 | |
| Hamming code | 2.20 | 1.18 | 1.00 | 3.00 | 1.22 | 1.00 | |
| equação diferencial | 2.14 | 1.13 | 1.15 | 2.14 | 1.13 | 1.36 | |
| Cordic | 1.69 | 1.18 | 1.04 | 1.90 | 1.21 | 1.05 | |
| LL1 | 1.60 | 1.12 | 1.33 | 1.50 | 1.11 | 1.63 | * |
| LL3 | 1.50 | 1.11 | 1.50 | 1.37 | 1.09 | 2.00 | |
| LL5 | 1.30 | 1.15 | 1.44 | 1.25 | 1.13 | 2.14 | |
| PID | 1.20 | 1.16 | 1.50 | 1.16 | 1.14 | 2.00 | |
| filtro digital | 1.11 | 1.20 | 1.22 | 1.10 | 1.19 | 1.75 | |
| produto vetorial | 1.57 | 1.18 | 1.22 | 1.57 | 1.18 | 1.57 | * |
| quick sort | 1.20 | 1.14 | 1.20 | 1.21 | 1.15 | 1.54 | |
| pesquisa binária | 1.10 | 1.27 | 1.20 | 1.12 | 1.33 | 1.20 | |
| procura de string | 1.12 | 1.25 | 1.12 | 1.15 | 1.30 | 1.28 | |
| concatenação listas | 1.20 | 1.24 | 1.14 | 1.25 | 1.29 | 1.41 | * |
| procura padrão | 1.07 | 1.21 | 1.16 | 1.09 | 1.25 | 1.50 | * |

A tabela 3.3 apresenta alguns resultados de aceleração global (AG), obtidos a partir da aplicação das equações acima descritas em um conjunto de exemplos, seja para o processador Risco seja para o DLX. As linhas marcadas com um * indicam quando houve diferenças na escolha da estratégia de otimização entre os processadores para uma mesma rotina. As diferenças advém basicamente das diferentes maneiras que cada processador utiliza memória. Como no pipeline do DLX está sempre previsto um acesso para leitura/escrita na memória de dados, operações como procura de padrões podem ser consideradas como rotinas condicionais, enquanto que para o Risco o tempo de acesso à memória é significativo, resultando em uma otimização WCS-PIPE.

Na primeira parte da tabela encontram-se algoritmos que exigem muitas computações, como a raiz quadrada inteira, duas rotinas de divisão e resolução de uma equação diferencial. Os programas LL1, LL3 e LL5 são retirados dos benchmarks de Livermore Loops. O PID e o filtro digital combinam funções de computação com acessos à memória. Os exemplos finais tratam de código escalar, com manipulação de caracteres e strings.

### 3.3 Otimização de rotinas computacionalmente intensivas

Uma vez que a estratégia vencedora seja a síntese da rotina crítica, deve-se implementá-la buscando a máxima aceleração em relação a execução no processador. Portanto, o máximo paralelismo de execução deve ser buscado. Este paralelismo existe não somente dentro dos blocos básicos da rotina a ser sintetizada, mas também entre a rotina transformada em HW dedicado e o processador que disparou a execução em HW da função. Diferentemente de outros sistemas, ECSD busca exatamente expor e aproveitar ao máximo este paralelismo potencial ([CAR 94a], [CAR 94b]).

### 3.3.1 Relação entre as rotinas chamadoras e chamadas

Quando da passagem de uma rotina de SW para HW deve-se analisar não somente a rotina a ser passada, mas sua chamadora também. Este tipo de análise trará benefícios significativos no que tange a operação em paralelo do SW e do HW, conforme será demonstrado adiante.

Se depois de uma instrução de chamada de função em HW existissem operações cujos dados não fossem dependentes do resultado da computação em HW, então estas operações poderiam ser movidas para uma posição próxima da chamada da função de HW, de modo a ocupar o espaço de espera do microprocessador pelos resultados do HW dedicado. Se este não for o caso, pode-se tentar explorar aquelas instruções acima da chamada de HW, mas estas possivelmente calculavam dados a serem utilizados pela rotina em HW. A análise do grafo de precedência identificará estas possibilidades.

Uma outra estratégia é a movimentação de operações através da execução de loop-unrolling. Existem muitas possibilidades da rotina crítica ser chamada muitas vezes durante a execução de um programa, visto que se assim não fosse, provavelmente não seria crítica. A possibilidade de que a chamada da rotina esteja em loops também é grande. Portanto, é interessante estudar a otimização sobre loops.

Uma rotina pode ser considerada de tempo de execução previsível ou não previsível. No primeiro caso, sabe-se o número de ciclos de máquina a serem tomados pela rotina durante sua execução. No segundo caso, o número de ciclos é dependente dos dados que chegam à rotina. Um exemplo para a linguagem assembly do Risco de rotina de tempo de execução previsível encontra-se na figura 3.3, quando os limites do array são conhecidos em tempo de compilação, e existe apenas um caminho possível.

```
                                        add         i, r0,r0
                                        add         prod, r0,r0
                                w_1:    sub.aps r0, i,K1
                                        jmp.GE      end_p
prod = 0;
i = 0;                                  ld          ai, a,i
while (i<Array_Size) {                  ld          bi, b,i
        prod = prod+a[i]*b[i];          push        ai, stack
        i = i+1;                        push        bi, stack
        }                               sr          MULT
                                        pop         res
                                        add         prod, prod,res
                                        jmp         w_1
                                        add         i ,i,1
                                end_p:  ret         res
```

FIGURA 3.3 - Produto escalar de 2 vetores

Rotinas de tempo de execução não previsível podem ser encontradas em laços cujos limites sejam dependentes de dados, ou mesmo quando os limites são conhecidos, o fluxo do programa é dependente dos dados que a ele chegam. Um exemplo deste tipo

de rotina encontra-se no próximo capítulo, figura 4.1, o clássico algoritmo de pesquisa binária.

A previsibilidade do tempo de execução na rotina chamada é importante visto que, se o número de ciclos de máquina é conhecido a priori, sabe-se o total de operações que deve ser movimentado na rotina chamadora de modo a manter o máximo paralelismo entre a função de HW e o microprocessador. Se esta informação por outro lado é desconhecida, um loop de espera deve ser inserido no código original (espera-se por uma interrupção ou realiza-se pooling em um conjunto de portas), de modo a que os resultados possam ser lidos. O número total de ciclos que uma rotina deve ocupar é importante para calcular-se o ganho obtido ao movê-la para HW.

Na rotina chamadora, ao determinar-se o número exato de ciclos de execução a análise de dados é facilitada. Neste caso, basta a aplicação de loop-unrolling para obtenção de algum ganho [CHA 92]. No caso de rotinas cujo número de ciclos não é determinável em tempo de compilação é preciso utilizar uma técnica que desenrole o laço junto com seu teste, ou ainda técnicas mais refinadas como trace-scheduling ([FIS 81]). Note-se que a otimização de loops em programas é um problema clássico da computação paralela ([LIL 94]). Contudo, muitas das técnicas propostas dependem obviamente de um HW de processador que suporte multiplas execuções, ou ao menos diversos operadores na parte operativa como as máquinas VLIW. Como não se pode alterar a arquitetura do processador hospedeiro, técnicas mais simples como loop-unrolling devem ser utilizadas.

### 3.3.2 Exemplo de paralelismo entre SW e HW

Quando uma rotina é crítica para a implementação de um sistema computacional, movê-la para HW pode ser uma estratégia vencedora. Como exemplo, podem-se citar rotinas de extração e rotação de bits, extração de raíz quadrada, soma em ponto flutuante, etc. O sistema PRISM por exemplo permite esta passagem, com limitações no número de ciclos que a rotina em HW pode ocupar. Como já mencionado, somente 1 ciclo de processamento poderia ser gasto.

Em princípio nada impede a utilização de mais de um ciclo do processador para realização de uma computação em HW, a não ser o fato do microprocessador ficar parado (executando nops) à espera dos dados que viriam das HW função de HW, como em Cosyma.

Embora a execução de uma rotina em HW seja possivelmente mais rápida do que em SW, o algoritmo como um todo que utiliza a rotina pode não ser beneficiado pelo mesmo fator multiplicativo. Por exemplo, referindo-se a figura 3.3 para o produto escalar de dois vetores. Como no processador Risco original não existe a instrução de multiplicação, deve-se fazer uma rotina em SW. Utilizando-se um simples algoritmo desloca-soma, o custo desta rotina é de 2+Array_Size*322 ciclos de máquina.

Se um HW dedicado estivesse disponível para realizar a operação de multiplicação, seria necessário transferir os dados para a função em HW, esperar pelo processamento e finalmente ler o resultado. Supondo que a multiplicação ocupasse 4 ciclos de máquina ([SOU 94]), o compilador poderia inserir automaticamente nops para que o microprocessador esperasse a disponibilidade do resultado.

A figura 3.4 mostra o algoritmo resultante com os ciclos de espera. O novo custo agora é de 2+Array_Size*23, um ganho significativo, mas que pode ser ainda ampliado. Uma simples inspeção mostra que o processador está parado enquanto espera a computação da função em HW, e portanto não se está aproveitando o conceito de duas máquinas executando em paralelo. Para explorar ao máximo o HW dedicado, deve-se levar este conceito em consideração.

Pode-se movimentar operações do algoritmo original de modo que a função em HW seja iniciada o quanto antes, evitando que o microprocessador deva esperar pelo seu término. Este conceito encontra-se representado na figura 3.5, onde após aplicação de loop-unrolling tem-se possibilidades de mover mais operações. O novo custo da rotina é de 2+Array_Size*16 ciclos de máquina.

```
            add       i, r0,r0              ;zera i
            add       prod, r0,r0
      w_1:  sub.aps   r0, i, K1             ;teste do loop
            jmp.GE    end_p
            nop
            ld        ai, a,i               ;carrega array
            ld        bi, b,i
            stpoi     ai, addHW, pHW_i      ;escreve na HWf
            stpod     bi, addHW, pHW_i
            MULT                            ;dispara execução
            nop                             ;espera término
            nop
            nop
            nop
            ld        res, addHW,pHW_o      ;lê resultado
            add       prod, prod,res
            add       i, i,1                ;nova iteração
            jmp       w_1
            nop
    end_p:  ret       prod
```

*FIGURA 3.4 - Produto escalar com multiplicação em HW*

### 3.3.3 Limitações quanto a movimentação de operações

A figura 3.5 mostra uma rotina de multiplicação de dois vetores depois de executado um loop-unroll e de movimentação de operações. Uma possível desvantagem do uso desta técnica encontra-se na figura 3.6. No caso, existe uma dependência de dados entre os resultados de uma iteração do laço e a seguinte. Nestes casos, loop-unrolling não produz efeitos significativos, e como o número de operações no laço não é suficiente para preencher os espaços de espera, nops devem ser inseridos. A aceleração então pode ser conseguida apenas pelo melhor desempenho do próprio HW que realiza a rotina, pela troca do algoritmo de cálculo ou por um aumento da freqüência de relógio. No exemplo da figura 3.6, trocando-se o número de ciclos de relógio da multiplicação de 4 para 2 tornaria o microprocessador livre de qualquer ciclo de espera.

```
            add         i,r0,r0
            add         prod,r0,r0
      w_1:  sub.aps     r0,i,k1
            jmp.GE      end_p
            nop
            ld          ai,a,i
            ld          bi,b,i
            stpoi       ai,addHW,pHW_i
            stpod       bi,addHW,pHW_i
            MULT
            add         i,i,1
            ld          ai,a,i
            ld          bi,b,i
            ld          res,addHW,pHW_o
            stpoi       ai,addHW,pHW_i
            stpod       bi,addHW,pHW_i
            MULT
            add         prod,prod,res
            add         i,i,1
            nop
            nop
            jmp         w_1
            nop
     end_p:ret          prod
```

FIGURA 3.5 - Produto escalar depois do loop-unroll

```
                                          add       i,r0,1
                                    w_1:  sub.aps   r0,i,997
                                          jmp.GE    end_W
                                          nop
                                          sub       auxi,i,1
                                          ld        xa,x,auxi
i=1;                                      ld        yi,y,i
while(i<997) {                            sub       aux,yi,xa
        x[i]=z[i]*(y[i]-x[i-1]);          ld        zi,z,i
        i=i+1;                            stpoi     zi,addHW,pHW_i
        }                                 stpoi     aux,addHW,pHW_i
                                          MULT
                                          nop
                                          nop
                                          nop
                                          nop
                                          ld        res,addHW,pHW_o
                                          stpoi     res,x,i
                                          jmp       w_1
                                          nop
                                   end_W:ret
```

FIGURA 3.6 - Laço com dependência de dados

Evidentemente, podem existir loops onde a função de HW é chamada mais de uma vez (na verdade, durante o loop-unrolling provoca-se artificialmente esta situação). Esta situação favorece a movimentação de operações, visto que os dados de uma chamada podem ser produzidos enquanto o HW dedicado está ocupado.

Um algoritmo de visita aos nodos de um grafo pode indicar o número de operações que separam uma chamada à função de HW de outra no caminho direto, ou se um HW dedicado está no caminho direto de outro ou não. O grafo de controle e dados da figura 3.7 mostra este fato. Os arcos contínuos indicam precedência de dados, enquanto que aqueles tracejados introduzem dependência entre operações que envolvem à chamada de HW. A chamada de HW da função multiplicação é feita através de uma instrução STORE, dai a referência a HW_st no diagrama.

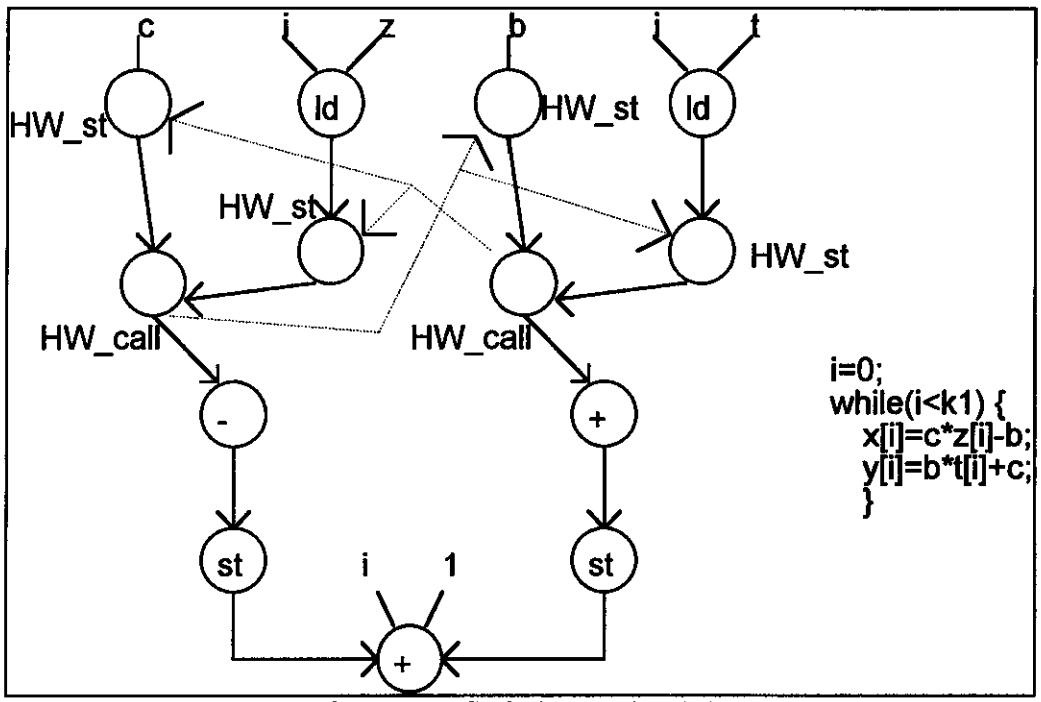



FIGURA 3.7 - Grafo de controle e dados

Poder-se-ia imaginar que a presença de duas funções de HW aceleraríam ainda mais a execução da rotina. A tabela 3.4 mostra o tempo relativo usando-se uma ou duas funções de HW. Contudo, uma simples análise do grafo mostra que se a função de HW utilizasse 3 ao invés de 4 ciclos de máquina, obter-se-ia uma aceleração igual ao uso de duas funções de HW, mas com custo associado possivelmente mais baixo. Este fato demonstra a importância de se realizar uma síntese virtual em primeiro lugar, de modo a realmente otimizar o HW dedicado em conjunto com a rotina chamadora. Somente depois de estabelecidas as melhores possibilidades de ganho deve-se proceder a uma síntese real.

TABELA 3.4 - Resultados com mais de uma função em HW

| Número de funções | ciclos do processador | custo |
|---|---|---|
| 1 | 4 | inicial + tamanho laço*24 |
| 2 | 4 | inicial + tamanho laço*21 |
| 1 | 3 | inicial + tamanho laço*21 |

Para execução do algoritmo de movimentação de operações tendo em vista o paralelismo entre duas máquinas utiliza-se um grafo de restrições como o da figura 3.7. Note-se que existem arcos entre as diferentes chamadas da função MULT. Isto porque não são permitidos acessos aos registradores da função de HW enquando esta está computando. Esta limitação pode ser removida no futuro, embora aparentemente não seja significativa.

```
                              lc      aux,x,i
                      w_1:    sub.aps r0,aux,r0
                              jmp.ne  fim
                              nop
                              ld      zi,z,i
                              stpoi   a,addHW, pHW_i
while (x[i] < 0) {            stpod   b, addHW, pHW_i
        x[i+1] = z[i]*a;      MULT
        i = i+1;              nop
        }                     nop
                              nop
                              nop
                              ld      aux, addHW, pHW_o
                              add     i,i,1
                              st      aux,x,i
                              jmp     w_1
                              nop
                      fim:    ret
```

FIGURA 3.8 - Laço dependente de dados

```
                              lc      aux,x,i
                      w_1:    sub.aps r0,aux,r0
                              jmp.ne  fim
                              nop
                              ld      zi,z,i
                              stpoi   a,addHW, pHW_i
                              stpod   b, addHW, pHW_i
                              MULT
                              nop
                              nop
                              nop
                              add     i,i,1
                              ld      aux, addHW, pHW_o
                              ld      zi,z,i
                              stpoi   a, addHW, pHW_i
                              stpod   zi, addHW, pHW_i
                              MULT
                              st      aux,x,i
                              sub.aps r0,aux,r0
                              jmp.ne  fim
                              ld      aux, addHW, pHW_o
                              stpoi   aux,x,i
                              jmp     w_1
                              nop
                      fim:    ret
```

FIGURA 3.9 - Resultado do loop unwinding

Para o caso de rotinas chamadoras cuja duração não pode ser determinada em tempo de montagem, ou ao menos, para laços de tempo não definido chamando a função de HW, loop-unrolling não pode ser aplicado. Nestes casos, realiza-se loop-unwinding, como no exemplo da figura 3.8. Embora tenha-se de repetir o teste de laço, mais operações estarão provavelmente a disposição para movimentação. Na figura 3.9 foi realizado o loop-unwinding, e o resultado é que todas as 4 instruções de espera da segunda multiplicação foram eliminadas por instruções úteis.

A movimentação é realizada pelo algoritmo da figura 3.10. Se mesmo após a aplicação de loop-unwinding em rotinas com tempo de execução não determinado não houver ganho, então algoritmos mais complexos como trace scheduling ([FIS 81]) poderiam ser utilizados para aproveitamento dos diferentes caminhos da rotina chamadora.

```
loop-unroll um bloco básico
enquanto existirem operações não agendadas
        pega operação a ser agendada
        se é uma chamada de HW
                move operações independentes para cima
                se mais de uma possível
                        escolha aquela cuja dependência é a outra chamada de HW
        elimina operação da lista
```

FIGURA 3.10 - Algoritmo de movimentação de operações

### 3.3.4 Protocolos de comunicação entre a função de HW e o microprocessador

Basicamente, pode-se embutir qualquer protocolo de comunicação entre a rotina chamadora e a função de HW, já que o ambiente ECSD estará no controle das ferramentas de síntese. Por exemplo, a espera do término de uma computação no HW dedicado poderia ser feita com um test-and-jump, significando que a comunicação é feita com um registrador set-reset permitindo acesso a portas de I/O.

O uso de interrupções seria recomendável caso o número de ciclos do HW dedicado fosse grande o suficiente e ao mesmo tempo o microprocessador pudesse executar outras rotinas sem dependência de dados. Provavelmente, a interrupção seria mais útil no caso de processos paralelos, e não de execução paralela de código.

Presentemente, utilizam-se os casos descritos a seguir para comunicação entre o microprocessador e a função de HW. No HW dedicado um conjunto de registradores de entrada é definido. Cada registrador ocupa um endereço no espaço de endereçamento do microprocessador. Os parâmetros são passados à função de HW de maneira sequencial através do uso de um ponteiro para a primeira posição onde estão os registradores do HW dedicado.

O retorno dos resultados computados também é feito através do espaço de endereçamento da memória. Uma posição de memória é lida e seus dados passados para um dos registradores do microprocessador.

Uma chamada para uma função de HW significa uma escrita em uma certa posição de memória gerando o evento de início para a máquina de controle da função. No caso em que o tempo de processamento é conhecido em tempo de compilação,

nenhum aviso é dado quando do final da computação. Visto que a sincronização entre o microprocessador e o HW dedicado pode ser descoberta em tempo de compilação, os dados estão simplesmente disponíveis. No caso de não se determinar o número de ciclos antes da execução, um outro endereço é utilizado como flip-flop sinalizador de fim.

O protocolo acima descrito é facilmente implementável, e eficiente se o número de parâmetros a serem passados não é muito grande. Para muitos parâmetros, provavelmente a rotina será intensiva em memória, e portanto deve ser tratada como tal. Atualmente, quando a função de HW está trabalhando, nenhum acesso a seus registradores é permitido. Isto foi escolhido para evitar a necessidade de duplicação de variáveis dentro da função, de modo a que os dados iniciais estejam disponíveis e os próprios registradores de comunicação possam ser usados como registradores de cálculo pelo HW dedicado.

No caso em que a função de HW e o microprocessador dividissem o mesmo meio físico de implementação (mesmo chip ou FPGA), as possibilidades de comunicação são mais interessantes. Por exemplo, a passagem de parâmetros poderia ser feita pelo próprio conjunto de registradores do microprocessador, ao custo de registradores de duplo acesso e uma diminuição dos registradores disponíveis para computação na PO do microprocessador. Esta é ainda uma área aberta na pesquisa.

### 3.4 Resultados práticos para o Risco

Foram implementados dentro do sistema ECSD as ferramentas de inserção automática de nops e movimentação de operações visando ao paralelismo descritas neste capítulo para o processador Risco. A partir de um conjunto de exemplos foram organizados resultados comparativos na tabela 3.5. No anexo 9 encontra-se um exemplo de uma sessão em ECSD com aplicação dos conceitos do capítulo.

Nota-se na tabela 3.5 que o ganho obtido com a passagem de uma operação complexa para HW é grande, e é ainda ampliado com os conceitos expressos neste capítulo. O ganho sem e com movimentação varia entre 7 e 20% (Equação diferencial e LL5, respectivamente). O único exemplo dependente do número de iterações da tabela 3.5 é o da Equação Diferencial, onde a significa o número de iterações. É importante ressaltar que embora o ganho seja percentualmente pequeno, o custo da movimentação de operações é ainda menor, e portanto obtem-se mais desempenho quase sem custos.

TABELA 3.5 - Conjunto de resultados para movimentação de operações

| Rotina | função em HW | número de ciclos HW | número de chamadas | custo Risco | custo c/HW | custo com movimentação |
|---|---|---|---|---|---|---|
| LL1 | Mult | 4 | 3 | 54800 | 18800 | 17000 |
| LL1 | Mult | 2 | 3 | 54800 | 17200 | 15800 |
| LL5 | Mult | 4 | 1 | 54837 | 26921 | 22434 |
| LL5 | Mult | 2 | 1 | 54837 | 24927 | 20939 |
| Eq. Diferencial | Mult | 4 | 5 | a*81 | a*58 | a*54 |
| Eq. Diferencial | Mult | 2 | 5 | a*81 | a*50 | a*47 |
| Gaussian Arithmetic Geometric Mean | Divisão | 16 | 1 | 4096 | 512 | 448 |
| Gaussian Arithmetic Geometric Mean | Divisão | 4 | 1 | 4096 | 320 | 264 |
| CORDIC | SIN/COS | 16 | 1 | 178 | 128 | 115 |

# 4 Otimizações com mudanças arquiteturais

Como já mencionado, as técnicas descritas no capítulo 3 servem bem a rotinas computacionalmente intensivas. Rotinas com muitos saltos ou com muitos acessos à memória principal podem não ter uma boa aceleração se forem simplesmente movidas para HW. Assim, devem-se buscar técnicas que permitam a otimização destas rotinas.

O problema da quebra do pipeline dos processadores modernos na execução de rotinas com muitos saltos tem sido largamente estudado, como na revisão apresentada por Lilja em [LIL 88] ou em [POL 88]. A estratégia de se organizar as operações a serem executadas de modo a aumentar o número de instruções por bloco básico em máquinas com arquitetura super-pipeline ou VLIW provocou o desenvolvimento de diversos algoritmos de ordenamento de execução de instruções entre saltos. Contudo, para rotinas com muitos saltos os algoritmos para aproveitamento do paralelismo de HW como software pipeline ou multiway branching podem ser aplicados a um elevado custo do HW dedicado e ainda bloqueando o acesso do microprocessador à memoria ([LAM 88], [AIK 88], [HUA 92], [ALL 92] e [PAP 93]). Pode-se imaginar porém que existam outras maneiras de se explorar o HW já disponível na PO do microprocessador.

As técnicas a serem apresentadas neste capítulo exigem contudo que o sistema seja composto por um processador custom, isto é, por um processador proprietário ou licensiado ao projetista de sistema, visto que são necessárias modificações na arquitetura da máquina global. Para mostrar a generalidade das soluções propostas, ao menos para as arquiteturas Risc, serão utilizados dois processadores, o Risco e o DLX ([HEN 90]).

## 4.1 Risco-WCS

Observe-se por exemplo o algoritmo de pesquisa binária apresentado na figura 4.1. O flow-graph do algoritmo encontra-se na figura 4.2. Por um algoritmo de inspeção e tradução este flow-graph pode ser convertido na FSM da figura 4.3. Basicamente, basta transformar cada operação do microprocessador em um estado da máquina destino. As instruções de salto são substituidas por um estado de teste. A nomeação de estados é feita simbolicamente, graças aos recursos disponíveis nos programas de síntese como o Syf ou Altera ([GRE 92] e [ALT 93]).

As operações onde existe a dependência de dados são colocadas em mais de um estado. Embora fosse possível explorar o paralelismo entre operações de blocos básicos, por motivos de compatibilidade com a Parte Operativa do microprocessador este paralelismo não é explorado. Esta aparente limitação restará clara a seguir, tendo em vista que a parte opertiva desta máquina de controle será o próprio processador.

Como o flow-graph foi construído a partir do assembly, é facil deduzir que todas as operações da FSM podem ser executadas pela PO do micro, enquanto que a tranformação do assembly em uma FSM Moore de controle é trivial, bastando colocar cada operação em um estado e renomear os estados sequencialmente tendo em vista a ordem do programa. O problema da codificação ótima de estados não existe, já que será utilizada uma memória de controle.

O problema passa a ser então a implementação da máquina de estados da figura 4.3. Note-se que a síntese em HW desta máquina não nos interessa, visto que, por hipótese, esta é uma rotina muito intensiva em saltos. Esta situação identifica pouco paralelismo disponível em blocos básicos e muitos desvios de controle. Como cada desvio de controle significa uma perturbação no pipeline do processador, tenta-se então diminuir os estágios de pipeline de modo a que as instruções de salto sejam executadas mais rapidamente.

```
struct {
        int key;
        int value;
        } d_arr[MAX];

bin_search(x) {
int val, mid, up, low=0;
up = MAX-1;
val = -1;
while (low <= up) {
        mid=(low+up)>>1;
        if (data[mid].key==x) {
                up=low-1;
                val=data[mid].value;
                }
        else    if (data[mid].key>x)
                        up=mid-1;
                else low=mid+1;
        }
return(val);
}
```

```
                add     low,r0,r0
                sub     up,max,1
                sub     val,r0,1
l1:             sub.aps r0,low,up
                jmp.GT  end_w
                nop
                add     mid,low,up
                shr     mid,mid,1
                ld      key,d_arr,mid
                sub.aps r0,key,x
                jmp.NE  else_1
                nop
                sub     up,low,1
                ld      val, d_arr,key
                jmp     end_if
                nop
else_1:         sub.aps r0,key,x
                jmp.LE  else2
                nop
                sub     up,mid,1
                jmp     enf_if
                nop
else_2:         add     low,mid,1
end_if:         jmp     l1
                nop
end_w:          ret     val
```

FIGURA 4.1 - Algoritmo de pesquisa binária

Já em 1985 reportaram-se arquiteturas que suportavam multi-way jumps e pre-fetches, de modo a diminuir o custo de saltos no pipeline do processador ([KAR 85]). A colocação de uma memória local para instruções, apesar de ser um conceito bastante difundido em microprogramação, também foi objeto de estudo tendo em vista a otimização de largos trechos de programas, como em [CHA 88]. O conceito de uma memória RAM interna utilizada como origem de instruções também veio da microprogramação, com alguns avanços como a possibilidade de conversão de uma parte da memória em controle ou armazenamento de dados, como em [MAL 88].

Em 1987 Ditzel apresentava uma proposta de diminuição do custo de saltos a zero para processadores com pipeline ([DIT 87]). A CPU Crisp é constituida por uma unidade de execução, uma memória cache de decodificação de instruções e uma unidade de fetch. Basicamente corta-se o pipeline a metade, pois a unidade executora originalmente com pipeline de 6 estágios é dividida em duas através da unidade de cache de decodificação. O acesso às instruções é feito por um barramento de 16 bits. A cache de decodificação transforma os 16 bits em uma microinstrução de 192 bits. A cache

armazena o endereço de próxima instrução, eliminando a instrução de salto. Uma instrução normal seguida de uma instrução de salto é transformada em uma única instrução pela cache, dai o nome da técnica de branch-folding.

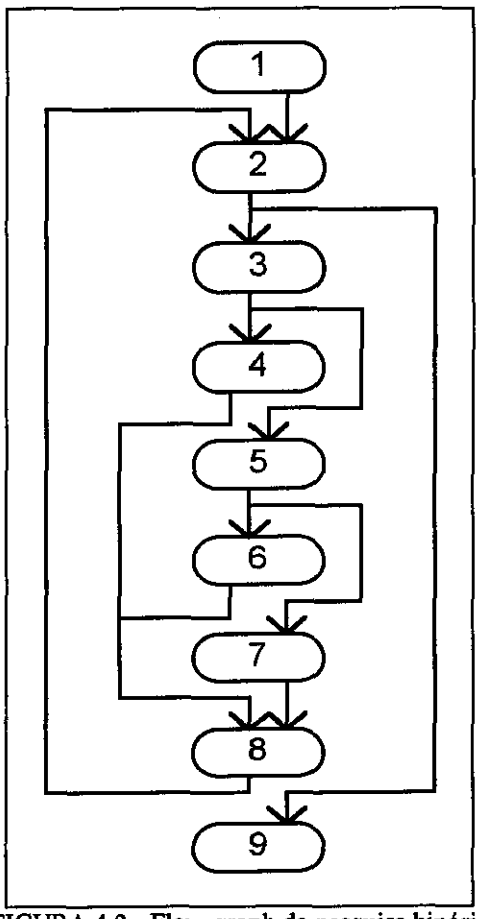



FIGURA 4.2 - Flow-graph da pesquisa binária

É preciso notar porém que para saltos condicionais é mantido um endereço de destino extra. Como a unidade executora não tem como informar a cache o destino antes de 3 ciclos, se o endereço assumido for o errado o pipeline é esvaziado, perdendo-se as instruções executadas erroneamente.

Seguindo-se a idéia de diminuir o número de estágios do pipe para máquinas com muitos saltos, para implementar a FSM da figura 4.3, uma memória de controle como a da figura 4.4 poderia ser utilizada. A Parte Operativa desta FSM de controle é o próprio processador Risco. A modificação real é que não é realizada a busca de instruções na memória externa, somente na local. Diferentemente de uma cache, e próxima à idéia de Ditzel, na memória de controle já se encontra disponível o próximo endereço de acesso. Portanto, a instrução de jump não mais existe, nem a quebra do pipe devido a ela.

A proposta do Risco WCS contudo difere daquela do processador Crisp no fato que a Parte Operativa desta nova máquina de controle é o próprio processador. Dito de outra maneira, enquanto que no Risco a memória interna armazena endereços e instruções como aparecem na memória principal, em Crisp ocorre uma decodificação dentro da cache, que transforma instruções de 16 bits em 192 bits, comandando diretamente a Parte Operativa. Isto impede que a unidade executora de Crisp seja utilizada como um processador normal se a rotina crítica não contiver muitos saltos.

A nova penalidade para saltos no Risco-WCS é de zero para saltos incondicionais, e um ciclo para saltos condicionais. Ao contrário de Crisp, não é necessário o esvaziamento do pipe no caso de saltos condicionais. Potencialmente, a solução de Crisp é melhor para execução de programas condicionais, tendo em vista que o tamanho da cache pode ser maior que a memória WCS do Risco. Contudo, Crisp está mais longe do conceito de ASIP, no sentido que uma rotina computacionalmente crítica ainda deveria ter a cache, enquanto que no Risco basta eliminar a WCS com economia de área.

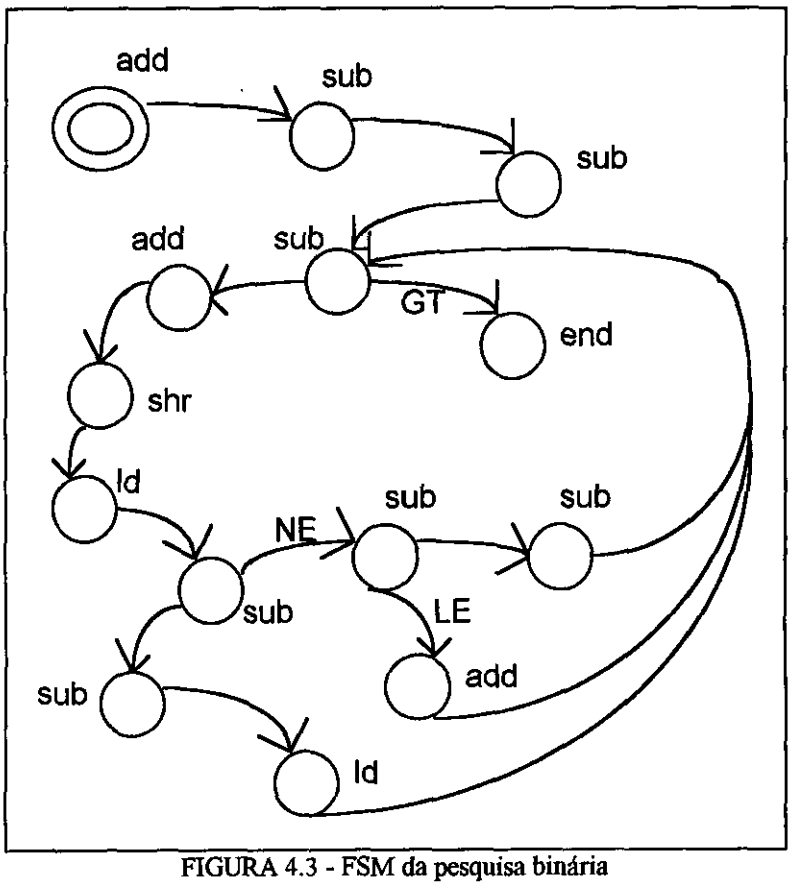



FIGURA 4.3 - FSM da pesquisa binária

O simples uso da memória de controle permite a aceleração do algoritmo de pesquisa binária de 2 a 2.35 vezes, dependendo do conjunto de dados e do resultado da pesquisa (primeiro ou último possível). Se a utilização de uma função em HW fosse tentada, a aceleração somente seria possível com o uso de uma memória local ao HW para rápidos acessos, além de duplicação de HW para multiway jumping.

A tabela 4.1 mostra o ganho obtido pela inclusão da memória de controle quando da execução de vários algoritmos condicionais, para o caso do processador Risco. Pode-se notar que o ganho converge para 100%, ou seja, as rotinas executam com o dobro da velocidade no Risco-WCS.

Note-se que o uso da WCS torna o Risco um ASIP, adaptado ao tipo de problema. Na verdade, o termo WCS (Writable Control Store) é oriundo da microprogramação ([HEN 90]), e atualmente esta técnica não é utilizada pelos fabricantes de processadores, por uma série de razões. Ressalte-se que a escrita de microcódigo era muito sujeita a erros, e de difícil manutenção. Em computadores que

deveriam executar sistemas operacionais com time-share, a carga da WCS com um novo contexto não poderia ser realizada com velocidade suficiente para que a execução do programa valesse a pena.

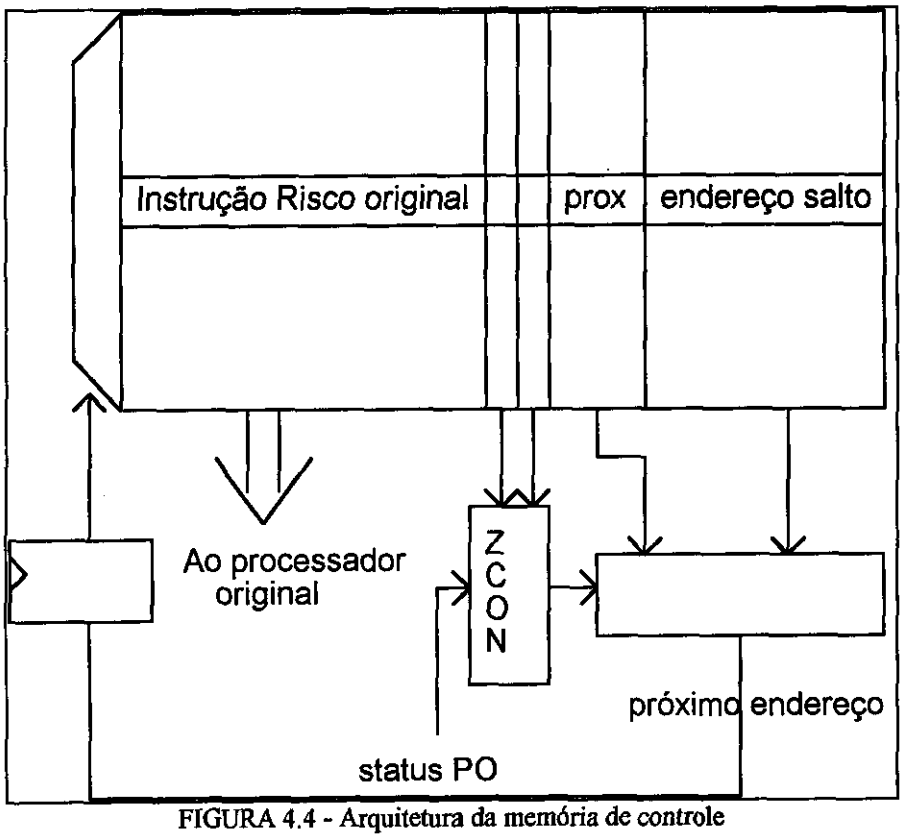



FIGURA 4.4 - Arquitetura da memória de controle

TABELA 4.1 - Desempenho do Risco-WCS para vários exemplos

| Rotina | Risco | Risco-WCS | Ganho | Condições |
|---|---|---|---|---|
| pesquisa binária | 26 | 13 | 2 | primeira |
| pesquisa binária | 6+log(S)*17 | 4+log(S)*7 | 2.37 | última, S=1024 |
| procura de string | 15+S*14 | 12+S*7 | 1.96 | primeiro, S=32 |
| procura de string | 8+(\|A-S\|)*18+S*14 | 7+(\|A-S\|)*10+S*7 | 1.88 | último, S=32,A=8 |
| concatenação listas | 17+L*18 | 7+L*9 | 2.00 | igual tamanho, L=32 |
| procura de caracter | 13 | 10 | 1.3 | primeiro |
| procura de caracter | 5+S*11 | 4+S*5 | 2.16 | último, S=32 |
| divisão inteira | 27 | 14 | 1.92 | um ciclo |
| filtro FIR | N*17 | N*14 | 1.21 | N = ciclos |
| PID array | N*17 | N*13 | 1.30 | N = ciclos |
| LL1 | N*20 | N*17 | 1.17 | N = ciclos |
| LL3 | N*17 | N*14 | 1.21 | N = ciclos |

Estas limitações não se aplicam aos processadores dedicados em ambiente de HW-SW codesign como aqui proposto. Não é necessária a microprogramação, já que na memória WCS encontra-se a mesma instrução que estaria na memória principal. Em outras palavras, utiliza-se a WCS como uma memória local, e não como uma memória de microprograma. Não existe problema de time-sharing (chaveamento de contexto entre vários usuários de um processador), já que o processador será utilizado para aplicações dedicadas e específicas. O uso fundamental da WCS no Risco e no DLX é a diminuição do número de estágios de pipeline quando de situações com programas com muitos desvios.

## 4.2 Características físicas do RISCO-WCS

Embora a simulação comprove ganhos, é importante verificar-se a factibilidade do uso de uma memória extra de programas. Esta seção aborda este fato e reporta as atividades de desenvolvimento do circuito do Risco-WCS.

### 4.2.1 Tamanho da memória

Obviamente, a memória de controle não tem custo zero. Para alguns sistemas ela poderia ser uma ROM, EPROM ou EEPROM, para outros uma RAM permitiria mais flexibilidade, visto que o programa que reside na memória poderia ser alterado em tempo de execução. Neste caso haveria instruções específicas para carga da WCS. A decisão do tipo de memória a ser utilizado depende do tipo de problema. Quanto mais específico o processador, mais interessante o uso de uma ROM ou EPROM. Para um caso mais geral ou para uso em mais de um projeto, deve-se utilizar uma RAM.

Como a memória está dentro do chip do processador, ela tende a ser pequena, tendo em vista a minimização da área. Quanto menor a memória, mais rápida, e portanto mais facilmente integrável ao pipeline do processador. Contudo, quanto maior a memória WCS, mais rotinas podem ser ali colocadas para execução.

Nesta primeira fase experimental, a memória de controle é constituida de 32 palavras de 46 bits. Destes, 32 são as próprias instruções que anteriormente residiriam na memória externa, 10 bits são utilizados para armazenar o endereço de destino e 4 bits indicam quais das 16 possíveis funções de comparação deve ser utilizada para decidir o endereço destino de salto na WCS.

Comparativamente, o tamanho proposto da WCS é apenas um pouco maior do que o banco de registradores do próprio Risco. Com 32 palavras, podem-se armazenar duas rotinas na WCS, por exemplo, a pesquisa binária e a divisão inteira. Para a versão em Standard Cells do processador Risco a WCS consumiu 32% da área total do circuito ([BUR 95]). No caso, o processador sem a WCS é realizado com 3199 gates em uma área de 5483x6207 micra$^2$. Com a inclusão da WCS e do roteamento tem-se a área final do projeto em 6130x7320 micra$^2$, o que permite concluir pela factibilidade do projeto. Note-se que embora o aumento em área seja de 32%, a memória em si ocupa apenas 3300x0.6 micra$^2$. A área extra é explicada pelo roteamento em Standard Cells, tendo em vista o grande número de conexões entre a memória e a Parte de Controle do processador. Possivelmente esta área possa ser muito diminuída pelo uso do Risco full-custom e da memória. O Risco full-custom possui área de 3000x2680 micra$^2$ , e portanto seria possível a realização do circuito WCS em uma área estimada de 6000x4000 micra$^2$, ou 24 mm$^2$ .

Tanto o Risco normal quanto o Risco-WCS foram submetidos à fabricação pelo PMU nacional e espera-se seu retorno para testes. A figura 4.5 mostra o layout do Risco-WCS, onde a grande área é a memória RAM de controle.

### 4.2.2 Temporização

A figura 4.6 mostra detalhes da memória e das fases que ativam cada registrador local. As fases foram distribuídas para que o pipeline do Risco não sofresse alteração, ou seja, quando o controle é passado para a WCS, a instrução seguinte àquela de transferência é ainda executada. Depois, como não é realizada uma busca de instruções na memória externa, a fase de decodificação é utilizada para acesso à WCS. Portanto, o acesso à WCS deve ser muito rápido, para que a fase de decodificação não sofra atrasos. Durante todo o tempo em que a WCS encontra-se em uso o circuito de fetch é congelado, e o contador de programa aponta para a próxima instrução válida depois do comando de ativação da WCS, WCSON.

Existem duas outras soluções que permitem o uso de memórias WCS mais lentas. A primeira seria construir-se a WCS com dupla porta, de modo a que ambas as instruções possíveis de destino fossem lidas. Esta técnica contudo exige o uso de uma memória WCS dupla porta, e portanto mais cara.

Outra possibilidade seria o fato de se remover a parte de decodificação do caminho dos sinais, como feito em Crisp. Na WCS armazenar-se-ia, ao invés da instrução como em memória, a lista de comandos da PO. Ou seja, transformar-se-ia o Risco em uma máquina VLIW de ULA única. Esta solução contudo significa memórias WCS maiores (mais bits de controle devem ser armazenados) e desperdício de recursos, pois toda a parte de decodificação do processador não é utilizada, mas continua sendo necessária para execução normal do processador quando da busca de instruções na memória principal.

Como o Risco tem arquitetura Risc, o atraso de decodificação não é grande. Na verdade, o caminho crítico de decodificação é constituído de uma porta NAND de 5 entradas em série com uma OR de 3 entradas. Assim sendo, uma memória com tempo de acesso de 20ns é suficiente para que na implementação da primeira versão do Risco-WCS não se altere o relógio do processador. Esta foi a opção final enviada para fabricação.

### 4.3 Risco-WCS e rotinas intensivas em memória

Quando o acesso à memória é dominante, tem-se a limitação do tempo de acesso à memória e do uso do barramento. Em teoria, apenas a substituição do processador e do sistema de memória poderia otimizar tal programa. Contudo, para alguns casos podem-se aplicar algoritmos de paralelização de loops como SW-pipeline ou agrupamento de laços ([POL 88]) para obtenção do máximo paralelismo, reduzindo-se depois a implementação conforme os recursos disponíveis.

O conceito acima exposto foi implementado como a técnica de WCS-pipe, onde o processador e um HW dedicado compartilham o acesso à memória para execução de laços. Por exemplo, seja o filtro digital da figura 4.7. No caso, utiliza-se o microprocessador com multiplicador embutido da família Risco ([CAR 94], [SOU 94], [SOU 95]). Na figura 4.8 o uso de recursos do processador é disposto no tempo, ao mesmo instante em que se realizou possível pipeline entre sucessivas iterações do loop. Note-se que a memória está sendo utilizada a maior parte do tempo. O custo total no caso sem WCS-pipeline é de 10*Array_Size, enquanto que com a aplicação de WCS-pipeline tem-se um novo custo de 6*Array_Size. O agrupamento de instruções é realizado tendo em vista o número de ciclos de acesso à memória. Como no Risco um

acesso à memória custa 2 ciclos de máquina, agrupam-se as instruções duas a duas para efeito de máximo acesso a memória.

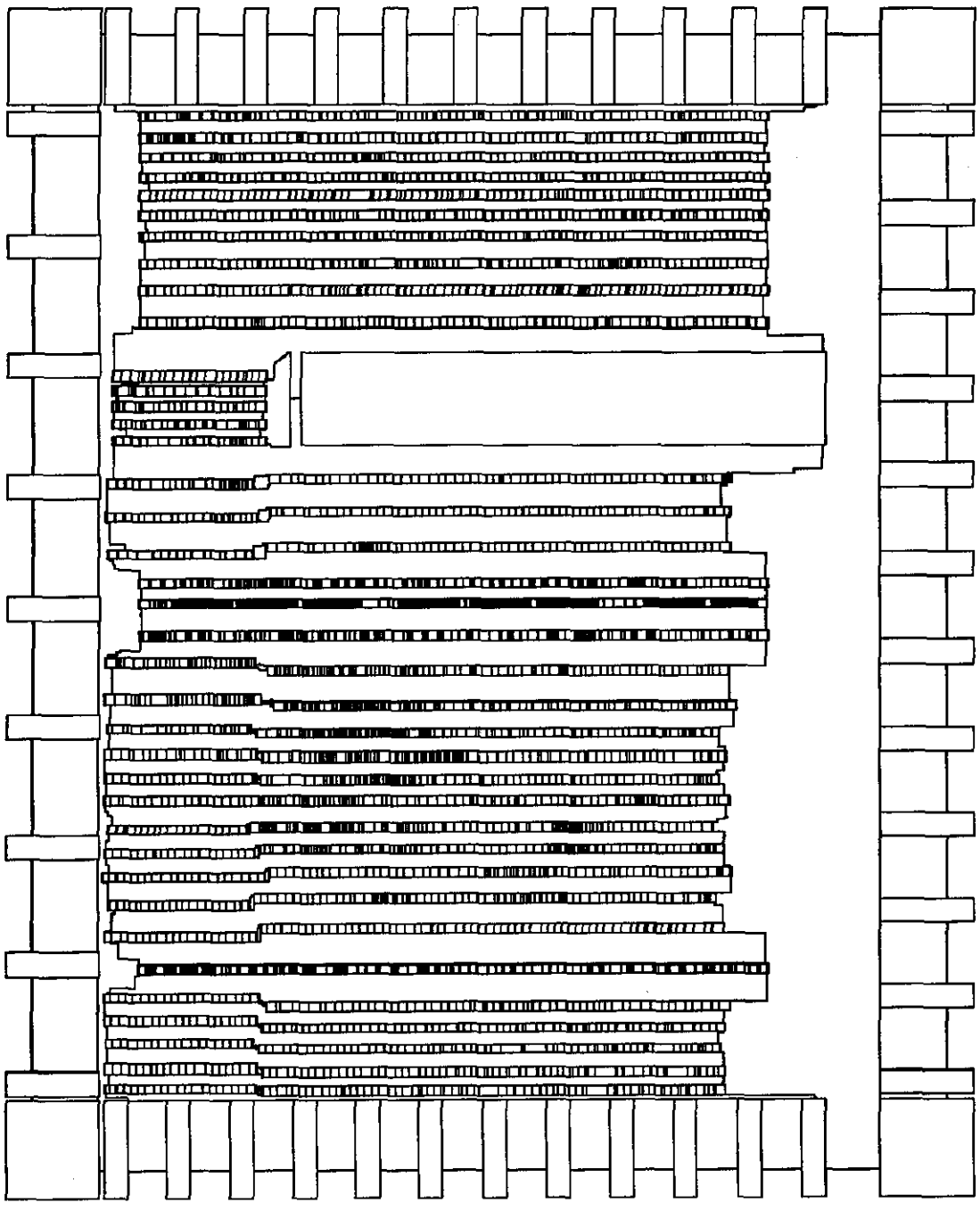


FIGURA 4.5 - Layout do Risco-WCS

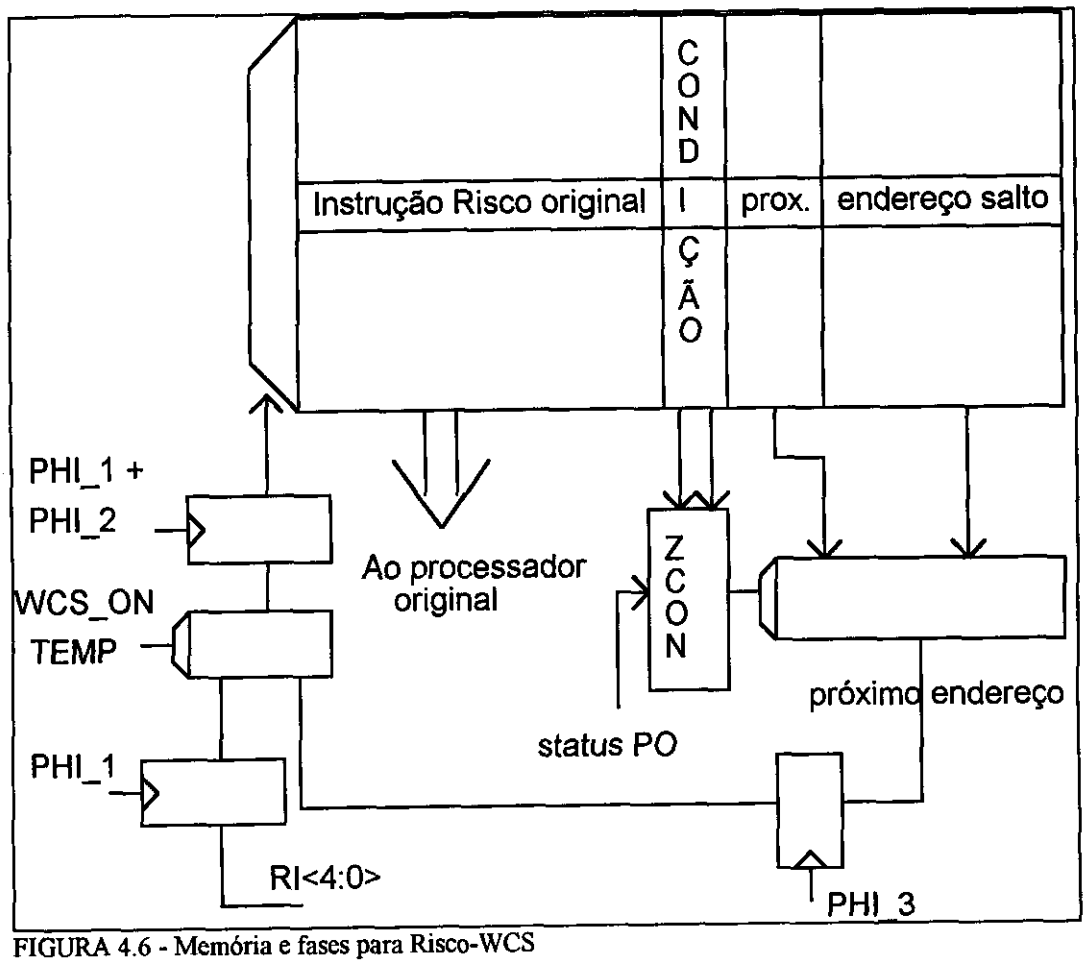



FIGURA 4.6 - Memória e fases para Risco-WCS

```
                                        add     i,r0,r0
                                w_1:    sub.aps r0,i,I
                                        jmp.GE  fim
                                        nop
/* yn = yn-1 + K*(x-yn-1) */           ld      xi,x,i
i=0;                                    ld      yi,y,i
while(i < I) {                          sub     aux,xi,yi
        aux = x[i] - y[i];              mult    aux,aux,K
        aux = aux * K;                  add     aux,aux,i
        y[i+1] = y[i] + aux;            add     i,i,1
        i = i + 1;                      st      aux,y,i
        }                               jmp     w_1
                                        nop
                                fim:    ret
```

FIGURA 4.7 - Filtro Digital

A idéia acima proposta exige suporte do sistema de CAD para execução. A sincronização entre as duas máquinas só é possível porque os limites de cálculo são conhecidos a priori. No caso em que os limites fossem conhecidos apenas em tempo de execução, a condição de parada exigiria que tanto o micro quanto o HW dedicado

possuíssem acesso à variável de controle do loop. Por enquanto esta possibilidade foi evitada nos exemplos estudados, tendo em vista a necessidade de controle e comunicação extra para se ter a variável de controle dividida entre duas máquinas. Uma estratégia intermediária poderia ser a liberação da função em HW para realizar computações, mas sem poder gravá-las na memória. Portanto, no caso mais prático onde somente o microprocessador teria acesso à variável de controle, o armazenamento de um resultado na memória deveria ser condicional, e poderia ser bloqueado a tempo pelo mecanismo de controle de acesso desabilitando a função de HW.

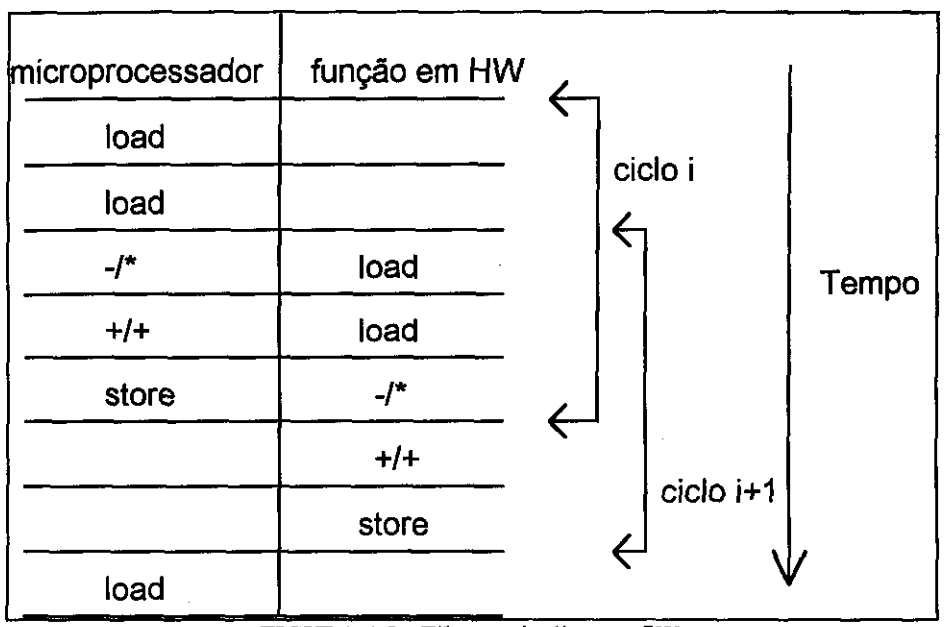



FIGURA 4.8 - Filtro e pipeline em SW

Além do problema de sincronização, existe aquele de dependência de dados entre os loops. Quando houver dependência real, nada poderá ser feito. Contudo, nos outros casos, técnicas de agrupamento de loops como as descritas em [POL 88] podem ser utilizadas com sucesso.

É importante notar que o HW dedicado deixa de ser passivo, (apenas recebia dados do micro), e passa a ter de buscar dados na memória global. Isto somente é possível se a memória de programa for separada da memória de dados, ou no caso de arquiteturas não harvard, quando o microprocessador não esteja fazendo acesso à memória. A segunda opção só é realizável com o uso do Risco-WCS, que não tem de ir à memória para buscar instruções. Aliás, microcontroladores também podem fazer uso desta técnica, visto que possuem muitas palavras de memória disponíveis internamente (8051, por exemplo).

A tabela 4.2 apresenta resultados para execução do WCS-PIPE para uma série de rotinas. Note-se que o ganho é sempre significativo, pois tem-se máquinas paralelas trabalhando para a computação. Este conceito de paralelismo poderia ser explorado por um HW dedicado que bloqueia o microprocessador, como no sistema Cosyma. A diferença provável são os custos envolvidos. Enquanto o uso da WCS abre espaço para o uso do WCS-PIPE e portanto otimiza dois problemas, o HW dedicado atua somente sobre um. A comparação final deve ser feita em termos de desempenho e custos. Apesar do custo da WCS, o HW dedicado no caso do WCS-PIPE tende a ser mais barato, pois

repete a PO do processador, sendo o paralelismo buscado em nível de código para duas máquinas idênticas.

TABELA 4.2 - Resultados para o WCS-PIPE

| Rotina | Risco | WCS-PIPE | Ganho | Condições |
|---|---|---|---|---|
| filtro FIR | N*17 | N*9 | 1.88 | N = ciclos |
| PID array | N*17 | N*10 | 1.70 | N = ciclos |
| LL1 | N*20 | N*10 | 2.00 | N = ciclos |
| LL3 | N*17 | N*11 | 1.50 | N = ciclos |

## 4.4 O DLX com WCS e WCS-PIPE

Para comprovar que as técnicas da WCS e do WCS-PIPE são gerais, ao menos para processadores com arquitetura RISC, implementou-se uma memória WCS no processador DLX ([HEN 90]). A descrição do processador DLX em VHDL foi obtida junto a universidade de Paris 6, Laboratório Masi, graças aos professores Pirouz Barzaghan e Alain Greiner.

O DLX possui um pipeline diferente do Risco, com 5 estágios, acesso à memória de dados em todos os ciclos e memória de dados separada de memória de endereços. A conseqüência é que o uso da WCS não acelera tanto a execução quanto no Risco. Isto porque o número de estágios do DLX passa de 5 para 4 (ganho de 20%) contra a passagem de 3 para 2 no Risco (ganho de 33%). Contudo, como pode ser observado comparando-se as tabelas 4.2 e 4.3, o DLX com WCS-PIPE possui ganhos superiores ao do Risco com WCS-PIPE. A própria diferença arquitetural é que provê a explicação para este fato.

No DLX o pipeline de 5 estágios pode ser visto como uma associação de 5 máquinas paralelas, contra apenas 3 do Risco. O uso de um HW dedicado que explore os acessos à memória na prática multiplica o efeito de paralelismo no pipeline. Logo, quanto maior a profundidade do pipeline, maior o ganho no uso da técnica de WCS pipe. Por isto o DLX apresenta ganhos mais significativos que o Risco. Nos casos em que o ganho é maior que 2 tem-se o efeito da presença da WCS, que diminui o custo de saltos durante o processamento.

TABELA 4.3 - Resultados para o DLX-WCS e WCS-PIPE

| Rotina | DLX | DLX-WCS | WCS-PIPE | Condição | Ganho |
|---|---|---|---|---|---|
| pesquisa binária | 18 | 16 | - | primeiro | 1.125 |
| pesquisa binária | 3+log(S)*18 | 3+log(S)*14 | - | último, S=1024 | 1.279 |
| procura de string | 15+S*12 | 12+S*8 | - | primeiro | 1.279 |
| procura de string | 20+(A-S)*22+S*10 | 18+(A-S)*16+S*12 | - | último, A=32 | 1.248 |
| filtro FIR | N*15 | N*13 | N*13/2+2 | N=1024 | 2.307 |
| LL1 | N*24+1 | N*21+1 | N*21/2+2 | N=1024 | 2.285 |
| LL3 | N*14+1 | N*12+1 | N*12/2+2 | N=1024 | 2.333 |

O anexo 4 apresenta a descrição VHDL do DLX-WCS conforme implementado, enquanto que o anexo 5 mostra a descrição VHDL do Risco-WCS.

## 4.5 Resultados para um exemplo completo de controle de motor

A fim de validar o uso das soluções propostas neste capítulo foi implementado um algoritmo de controle vetorial de motores de indução ([REG 95]). O programa utiliza multiplicações em ponto fixo, divisão e operações trigonométricas. O algoritmo foi colocado no ambiente e uma série de otimizações voltadas ao Risco foram implementadas. As operações trigonométricas foram colocadas como um HW dedicado, e a função de multiplicação na WCS. Como a multiplicação é feita em ponto fixo, uma série de testes deve ser feito a cada passo de controle para verificação dos limites do resultado, caracterizando uma rotina dominada por saltos.

A rotina de controle do motor encontra-se no anexo 8, seguida dos diferentes passos do ambiente ECSD relativos a análise das rotinas e simulações com a WCS. A tabela 4.4 resume os resultados obtidos.

TABELA 4.4 - Resultados para o controle do motor de indução

| Versão | Aceleração |
|---|---|
| Risco Padrão | 1 |
| Risco e função de HW | 1.367 |
| Risco e WCS | 1.485 |
| Risco com ambas | 1.864 |

# 5 Otimização de sistemas independentes da arquitetura

Nos primeiros capítulos deste trabalho foram apresentadas técnicas de realização e otimização de sistemas computacionais tendo em vista uma arquitetura alvo baseada em processadores RISC, já que modificações arquiteturais são mais facilmente realizáveis em arquiteturas do tipo Risc. A movimentação de operações é aplicável em qualquer caso de arquitetura, mas para microcontroladores, por exemplo, nem sempre é vantajosa, tendo em vista que a função de HW pode ser várias ordens de grandeza mais rápida do que a execução de um programa em um microcontrolador.

Na realidade, processadores com arquitetura CISC são bastante presentes no mercado nacional e internacional. Para aplicações industriais e automotivas podem-se citar os já clássicos 8051 ou o 6811. Além disto, um sistema baseado em um microprocessador como o 386 ou 486 teria à disposição um sistema de desenvolvimento já pronto, qual seja, o próprio Computador Pessoal onde atuam estes processadores. No caso da indústria eletrônica nacional, um grande número de sistemas é equipado com arquiteturas como o TMS320C25-31 ou o conceitualmente antigo mas eficiente MCS8051 e outros membros da família.

Para aplicações que utilizem microcontroladores, contudo, são necessárias técnicas que sejam independentes da arquitetura, válidas tanto para microcontroladores quanto para processadores Risc, explorando o espaço de atuação permitido pelos recursos tecnológicos e de CAD atuais. Um sistema microcontrolado geralmente tem como um de seus objetivos a economia de componentes e espaço em placa, tendo em vista o próprio uso do microcontrolador, um sistema digital programável com memória dentro do próprio CI. Um dos objetivos de uma otimização seria a transformação do processador em outro mais econômico, do ponto de vista de área. Isto feito, a nova área disponível poderia ser utilizada para inclusão em uma mesma pastilha de periféricos do microcontrolador ou de circuitos analógicos auxiliares. Este é o conceito da realização de sistemas em silício, ou ASIS (Application Specific Integrated Systems).

Este capítulo apresenta as possibilidades de otimização de processadores em relação à realização integrada de um circuito que implemente um sistema para aplicação específica.

## 5.1 Estratégia de economia de área

Para que um sistema computacional baseado em processadores-padrão possa estar encapsulado em um mesmo integrado deve-se ter grande disponibilidade de área, e uma interação forte com a empresa de semicondutores fabricante do processador. As disponibilidades tecnológicas atuais, seja em termos de tecnologia, seja em termos de CAD, permitem a realização de processadores compativeis em SW e temporização com os processadores clássicos. Nada mais natural que proceder à implementação de sistemas computacionais com o uso também de processadores off-the-shelf.

Apesar das técnicas de movimentação de rotinas críticas para funções em hardware e movimentação de operações para paralelismo como apresentadas no capítulo 3 serem aplicáveis a qualquer sistema microprocessado, as modificações arquiteturais propostas no capítulo 4 não são tão efetivas devido às arquiteturas alvo diferentes. O

8051 por exemplo não possui pipeline, o P6 possui uma arquitetura superescalar, enquanto que o 88000 é baseado em um processador Risc, mas a unidade de ponto flutuante encontra-se integrada na mesma pastilha do processador principal.

Um dos motivos de ser difícil a aplicação de mudanças arquiteturais em um processador Cisc como os microcontroladores mais populares advém do fato que uma nova instrução ou uma mudança no ciclo de instruções provoca a necessidade de mudanças no compilador deste processador. Para processadores com arquitetura Risc reescrever-se o compilador é uma tarefa factível, enquanto que para processadores Cisc, devido ao grande número de instruções e às diferentes otimizações necessárias, tem-se potencialmente mais trabalho. Além disto, a compatibilidade de SW é fundamental, pela larga difusão destes processadores pelo investimento existente no treinamento de engenheiros.

As modificações que se podem realizar na arquitetura do processador são portanto limitadas em software, mas pode-se tranquilamente explorar o espaço de projeto no que tange a área do processador. Como os microcontroladores mais usados são de arquitetura conceitualmente simples, isto é, sem pipeline e com parte de controle tomando uma área significativa, a otimização desta arquitetura é factivel. Por exemplo, a colocação de pipeline no 8051 seria plenamente razoável nos dias de hoje. O novo processador, apesar de mais rápido, não possuiria a mesma temporização do processador original.

Outra possibilidade a ser explorada é o fato que a arquitetura do microcontrolador é Cisc. Isto significa que, provavelmente, nem todas as instruções são usadas eficientemente pelo compilador, conforme estudos já clássicos de [PAT 80] e [HEN 84]. Portanto, se o conceito da diferença Risc-Cisc pode aqui também ser aplicado, a área que resta será utilizada para nova funções, sejam elas o HW dedicado que acelerará um certo programa ou uma interface analógica, um sistema de comunicação de redes, etc.

Por se manter o conjunto de instruções original e a temporização original, o novo processador é completamente compatível com o originário, e portanto toda a base de SW instalada e eventuais placas protótipo também o são. Pode-se pensar nesta estratégia como o uso de ASIPs em sistemas já estabelecidos, onde será otimizado o sistema completo, e não somente o processador. Além disto, pode-se imaginar que este nova implementação do sistema tem fácil acesso à prototipação: é o próprio processador origem em uma placa executando a aplicação alvo.

O ambiente necessário para gerar este novo processador é aquele já disponível atualmente, ou seja, programas de síntese lógica ou de alto nível e analisadores de rotinas críticas. Para realização de sistemas completos envolvendo circuitos analógicos ainda devem ser realizados mais estudos (vide capítulo 7). É importante observar-se que a síntese pura e simples de um processador com um conjunto de instruções reduzido em relação ao original nunca será mais vantajosa que a simples compra do processador original, que é vendido aos milhões para o mundo todo. O conceito a ser buscado é que o novo processador provoca uma economia de área que favorece a integração de outros módulos, implementando um sistema completo ou grande parte dele em um único circuito integrado, aumentando seu valor agregado.

## 5.2 A análise de programas alvo

Pelo raciocínio acima exposto, o ASIP será implementado tendo como objetivo um ASIS, ou Application Specific Integrated System. Os passos para tal partem da liberação de área no processador original e do uso desta área para circuitos dedicados descobertos por técnicas de profiling (capítulo 6) ou por análise estática (como no capítulo 3).

O microcontrolador escolhido para otimização foi o MCS8051. As razões desta escolha devem-se a sua larga popularidade na íindustria local e internacional, assim como a disponibilidade abundante de compíladores e programas já desenvolvidos para esta arquitetura.

A primeira tarefa é verificar se os compiladores disponíveis para estes microcontroladores são tão ineficientes em relação ao uso do conjunto de instruções quanto se imagina. Para tal verificação foram realizados diversos ensaios com vários compiladores C para o 8051. Contruiu-se um analisador de assembly para o 8051 capaz de realizar estatísticas quanto ao número de ocorrências de uma instrução de maneira estática, isto é, em tempo de compilação de um programa.

Durante este trabalho foram quase sempre utilizadas ferramentas comerciais, de modo a reproduzir o ambiente que um projetista de sistemas encontra à disposição no mercado. Para todos os exemplos foi utilizada a biblioteca padrão disponível nos compiladores C, com as funções sin, cos, abs e outras.

Foram estudadas três aplicações específicas: o controle vetorial de motor de indução [REG 95], um controle de válvulas [LOR 93] e o protocolo para um *barramento de chão-de-fábrica* [LOR 93a]. Cada programa representa uma pesquisa em andamento no Departamento de Engenharia Elétrica da UFRGS, utilizando microcontroladores. Os programas foram escolhidos não somente pela sua disponibilidade, mas também por sua especificidade. Desejava-se avaliar problemas específicos, visto que o caso geral já é coberto pelo processador comercial padrão. O código fonte C de cada programa encontra-se no anexo 11, enquanto que o assembler analisado está no anexo 10.

Cada programa foi analisado estática e dinamicamente. A análise estática apenas registra quais instruções são utilizadas e o número de ocorrências de uma instrução em tempo de compilação. Com este ensaio pode-se verificar quais instruções são realmente utilizadas do conjunto total. Uma instrução que não apareça nesta lista certamente nunca será chamada, e portanto pode ser removida do código VHDL que gerará o processador.

Durante a análise dinâmica verificam-se quais instruções são realmente executadas para uma certa aplicação. Este tipo de estudo mostra a freqüência de ocorrência de uma certa instrução, e portanto revela chaves de otimização quando se desejar melhorar o desempenho do processador e for permitida uma mudança na temporização do processador. Para realização da análise dinâmica deve-se ter acesso ao conjunto de instruções realmente executadas pelo processsador, o que exige ou um simulador com trace ou um sistema de desenvolvimento com tal recurso. No estudo a ser apresentado foi utilizado um emulador com um trace e desassembly, de modo que o

analisador estático e as ferramentas já desenvolvidas para a análise estática pudessem ser acopladas ao sistema.

Para este estudo foram utilizadas as ferramentas de trace da Metalink, o disassembler *DIS8051 Cross Disassembler v.2.1* , o compilador *Archimedes 8051 C Compiler v.4.23* and *C-51 C Compiler v.2.30* (Keil-C), além de utilitários escritos em C para o MS-DOS.

Para todos os exemplos estudados somente uma fração do conjunto de instruções era efetivamente utilizado durante a execução do código. Além disto, um grande número de instruções nunca foi utilizado. Estas instruções não sendo necessárias podem ser eliminadas completamente com melhoria de área. A tabela 5.1 mostra a lista de grupos de instruções nunca utilizadas para os três exemplos. Note-se que se consideram grupos de instruções devido aos diferentes modos de endereçamento e operação. Assim, a ausência de JMPs refere-se a duas instruções de salto absoluto, JMP endereço e JMP @A+DPTR.

TABELA 5.1 - Lista de grupos de instruções jamais utilizadas na análise estática.

| **Controle de Motor** | **Controle de Válvula** | **Protocolo Profibus** |
|---|---|---|
| ACALL | ACALL | ACALL |
| CALL | CALL | AJMP |
| DA | DA | CALL |
| DIV | DIV | DA |
| JBC | MUL | DIV |
| JMP | RL | JMP |
| RETI | RRC | MOVC |
| RR | SWAP | RLC |
| SWAP | | RR |
| | | RRC |

A figura 5.1 apresenta um gráfico comparativo da análise estática dos grupos de instruções mais utilizadas para o programa do Controle de motor, enquanto que na figura 5.2 apresenta-se a análise dinâmica para o mesmo problema. Como a teoria Risc previa, mais de 80% do código é coberto por uma fração mínima de instruções. A tabela 5.2 mostra os grupos de instruções mais utilizados.

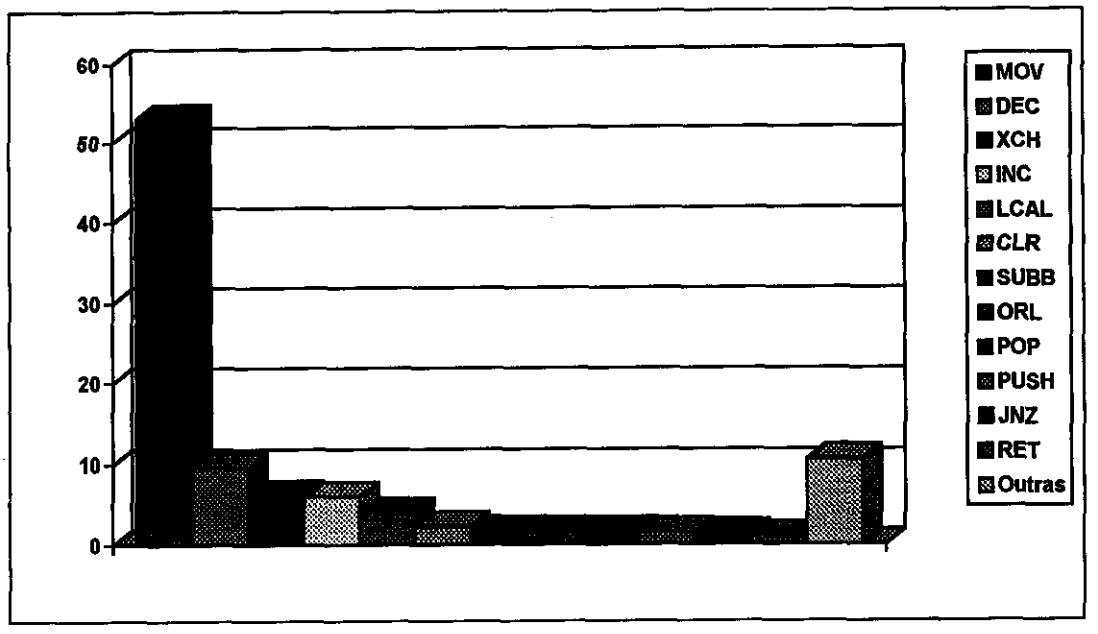



FIGURA 5.1 - Grupos de instruções mais utilizados na análise estática do programa para controle do motor

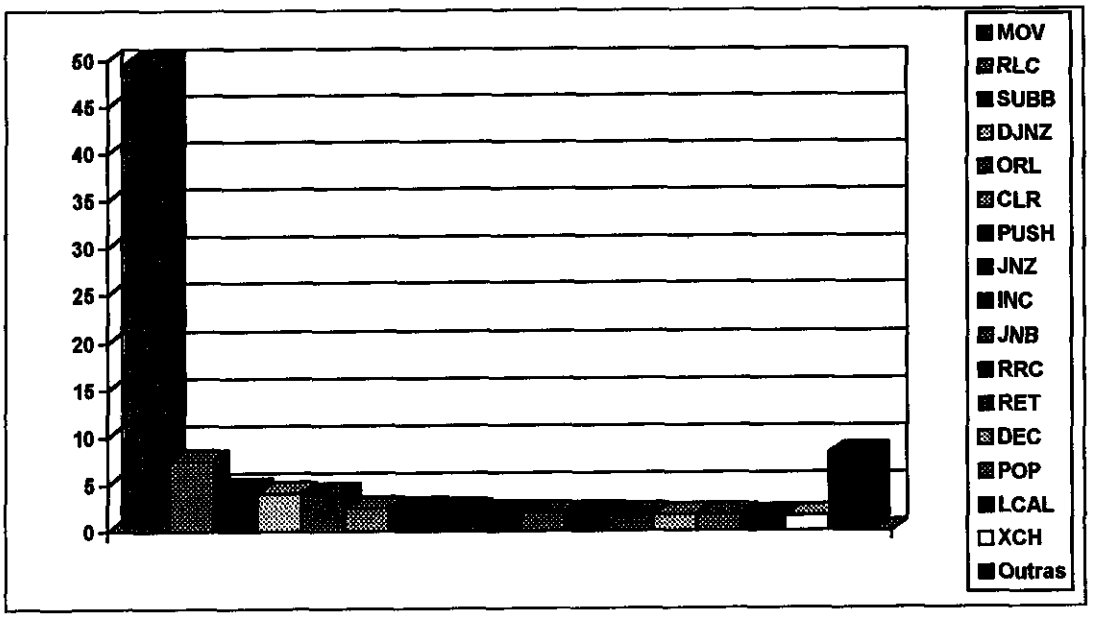



FIGURA 5.2 - Grupos de instruções mais utilizados na análise dinâmica do controle do motor

TABELA 5.2 - Grupos de instruções mais utilizados no programa para controle do motor

| **Grupos de instruções mais utilizadas (análise estática):** | **(%)** | **Grupos de instruções mais utilizados (análise dinâmica):** | **(%)** |
|---|---|---|---|
| MOV | 52.918 | MOV | 49.730 |
| DEC | 9.596 | RLC | 7.499 |
| XCH | 6.117 | SUBB | 4.444 |
| INC | 5.976 | DJNZ | 4.120 |
| LCALL | 3.984 | ORL | 3.965 |
| CLR | 2.273 | CLR | 2.546 |
| - | - | PUSH | 2.469 |
| - | - | JNZ | 2.330 |
| - | - | INC | 2.037 |
| **TOTAL:** *6 grupos* | *80.864 %* | **TOTAL:** *9 grupos* | *79.140 %* |

Procedimentos idênticos foram feitos para os outros exemplos, e os resultados encontram-se nas figuras 5.3 e 5.4.

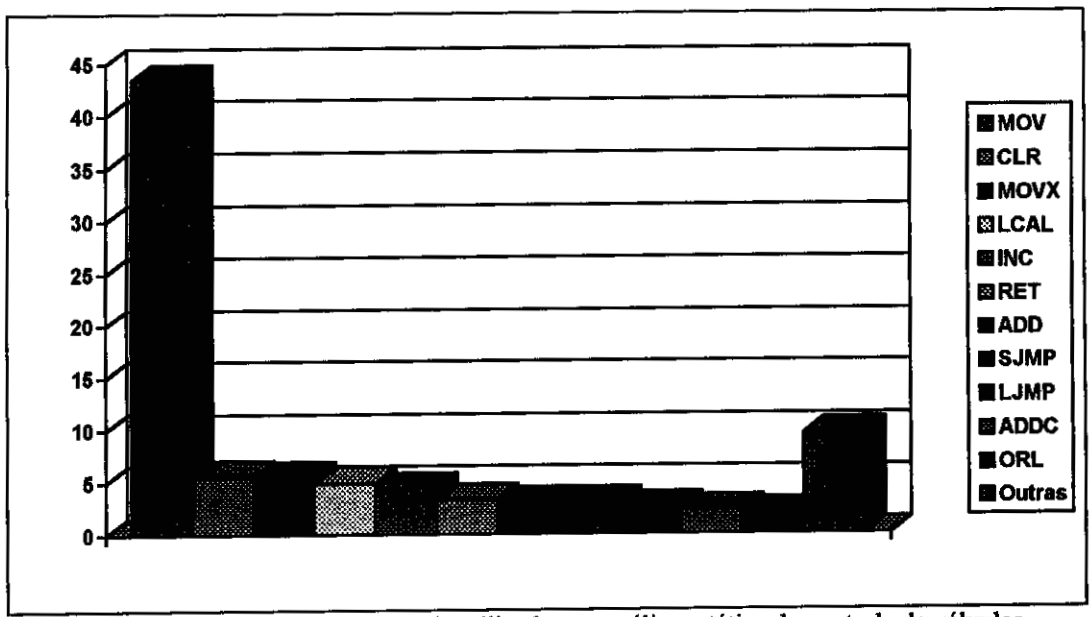



FIGURA 5.3 - Grupos de instruções mais utilizados na análise estática do controle de válvulas

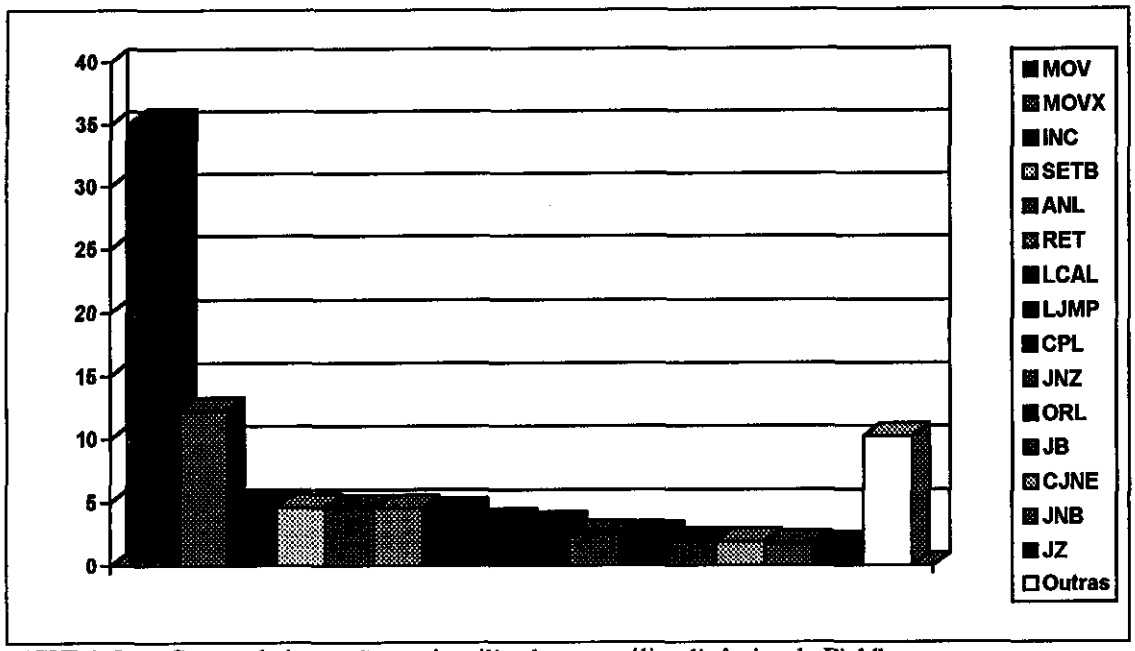



FIGURA 5.4 - Grupos de instruções mais utilizados na análise dinâmica do Fieldbus

TABELA 5.4 - Estatísticas de uso dos grupos de instruções do programa de controle das válvulas

| **Grupos de instruções mais utilizados (análise estática):** | **(%):** |
|---|---|
| MOV | 43.460 |
| CLR | 5.586 |
| MOVX | 5.450 |
| LCAL | 4.905 |
| INC | 4.360 |
| RET | 3.270 |
| ADD | 2.997 |
| SJMP | 2.997 |
| LJMP | 2.452 |
| ADDC | 2.180 |
| ORL | 1.907 |
| **TOTAL:** *11 grupos* | *79.564 %* |

TABELA 5.6 - Estatísticas dos grupos de instruções encontrados no sistema profibus

| **Grupos de instruções mais utilizados (análise dinâmica):** | **(%)** |
|---|---|
| MOV | 35.191 |
| MOVX | 12.290 |
| INC | 4.936 |
| SETB | 4.682 |
| ANL | 4.580 |
| RET | 4.580 |
| LCAL | 4.351 |
| LJMP | 3.511 |
| CPL | 3.232 |
| JNZ | 2.545 |
| ORL | 2.494 |
| JB | 1.985 |
| CJNE | 1.934 |
| JNB | 1.781 |
| JZ | 1.679 |
| ADDC | 1.476 |
| CLR | 1.018 |
| **TOTAL:** *17 grupos* | *89.771%* |

## 5.3 Resultados de economia de área

Para avaliar o ganho em área relativo à retirada das instruções nunca utilizadas foram feitas diversas versões do 8051. A única diferença entre cada versão era a parte de controle do processador, onde as instruções eram retiradas ou inseridas conforme a necessidade do algoritmo. O processador foi descrito em VHDL através de sua divisão clássica entre Parte de Controle, Parte Operativa e Parte de Validação. Tomou-se cuidado durante a descrição de tornar a inserção ou retirada de instruções o mais prática possível, tendo em vista que esta operação, no futuro, deverá ser feita de maneira automática.

A estratégia acima mencionada resultou em uma Parte de Controle simplificada, na prática apenas um gerador de 12 estados seqüenciais dependentes do tipo de instrução, variável com o número de bytes ou ciclos por instrução. A Parte Operativa não foi nunca alterada, tendo em vista que ela não é a responsável pela maior área do processador. Na Parte de Validação foram inseridos comandos de atuação de instruções dependentes dos estados da parte de controle. Assim, a retirada de instruções significa apenas a retirada de uma equação lógica relativa a aquela instrução na Parte de Validação. A tabela 5.7 resume os resultados obtidos.

TABELA 5.7 - Resultados de economia em área

| **Programa aplicativo** | **numero de instruções jamais utilizadas** | **portas lógicas economizadas** | **% de economia** |
|---|---|---|---|
| Controle de Motor | 51 | 250 | 43% |
| Profibus | 48 | 224 | 39% |

Como se pode observar, a economia em área da Parte de Controle é significativa. Em termos de gates economizados, tem-se um ganho real. No espaço economizado podem ser instaladas três 8255 que fazem parte da aplicação. Isto significa a eliminação de 3 circuitos além do processador, com conseqüente diminuição de área de placa.

Foi feita também uma tentativa de implementação de um subconjunto de 50 instruções do 8051 em FPGA, visando a prototipação do processador. A tabela 5.8 apresenta os resultados atingidos. É preciso ressaltar que o circuito completo não foi suportado pelo ambiente de CAD Altera, causando um erro proveniente da falta de recursos (memória física). A alternativa encontrada foi sintetizar cada módulo separadamente. Mesmo assim, a parte de validação somente foi implementada com 17 instruções, devido a limitações do sistema de síntese automática. Para comparações com o Risco de 16 bits, foram utilizados somente 16 endereços de ROM e 16 endereços de RAM interna, o que equivale ao banco de registradores do Risco 16 bits a ser descrito a seguir.

TABELA 5.8 - Resultados para o 8051 em FPGA

| Circuito | FPGA | Total de LEs | Total de portas | LEs usadas | Portas utilizadas |
|---|---|---|---|---|---|
| Controle | 8282PLCC84 | 208 | 2500 | 42 | 504 |
| Temporizador | 8282PLCC84 | 208 | 2500 | 93 | 1116 |
| Operativa | 8636PQFP160 | 504 | 6000 | 428 | 5136 |
| Validação | 8282PLCC84 | 208 | 2500 | 178 | 2136 |
| ROM | 8282PLCC84 | 208 | 2500 | 42 | 504 |
| RAM | 8452PLCC84 | 336 | 4000 | 277 | 3324 |

Note-se que embora a Parte Operativa ocupe a maior área, é preciso considerar que somente as 17 instruções mais importantes foram implementadas (82% do código do controle de motor). Isto significa que realmente a parte de Validação exige uma intensa área. Portanto, a idéia de realização de um processador com menos instruções pode realmente significar economia de área. Como 82% do código é coberto pelas instruções propostas, pode-se imaginar que, embora sem ser completamente compatível em termos de temporização, a nova arquitetura possa substituir o 8051 original na maioria das aplicações. Além disto, com poucas instruções presentes, e removendo-se a limitação de compatibilidade em nível de ciclos de máquina, pode-se modificar a Parte Operativa para inserção de uma fila de instruções e pipeline, acelerando o relógio do processador.

O custo em portas lógicas a que se refere a tabela 5.8 é um custo para FPGAs. Em Standard Cells este custo seria menor, conforme visto na tabela 5.7. Isto se deve não propriamente ao estilo de síntese, mas à arquitetura alvo do projeto. Como cada Logic Element (LE) de um FPGA da família Altera 8000 contém duas Look-Up Tables, o gasto de portas é proporcional ao gasto de LE, que por sua vez depende do número de FFs de um circuito e da programabilidade. Para um projeto fortemente combinacional, como a parte de validação, a arquitetura dos FPGAs utilizados não é ótima. Por exemplo, sintetizando-se menos instruções na parte de validação e comparando resultados, pode-se estimar em 108 portas o custo de cada nova instrução. Em Standard Cells este número varia entre 4 e 6 portas por instrução.

Como se verá adiante, o 8051 completo não pode ser implementado no mesmo FPGA do Risco (81500), ainda que o Risco tenha 16 bits. Na versão do 8051 com número reduzido de instruções espera-se que a ocupação em área seja de 1060 LCs, ou 12720 portas. Este circuito poderia ser implementado em um 81188 (12000 portas utilizáveis) ou em um 81500 no limite. Apesar do preço elevado (mais de US$100), considera-se para efeitos de prototipação a implementação com FPGA válida. Em termos de competição com o 8051 original, note-se que uma implementação de 20000 peças em standard cells poderia ser obtida por aproximadamente US$11, mas com os periféricos como a 8255 incluidos, o que torna o circuito mais conveniente que o processador original e periféricos em termos de espaço em placa e mesmo preço final.

## 5.4 A família de processadores Risco

A disponibilidade crescente de FPGAs e outros circuitos de lógica programável de baixo custo com alta capacidade de integração tornam possível a prototipação e mesmo a produção de circuitos dedicados digitais complexos, mesmo para pequenas quantidades. Embora alguns passos atrás de tecnologias full-custom ou mesmo standard cells no número de componentes para integração, as vantagens da lógica programável em termos de reusabilidade, facilidade de programação e teste tornam este tipo de componente uma opção viável para a implementação de sistemas eletrônicos computacionais.

Com a tecnologia hoje disponível é possível a implementação de processadores dedicados com arquitetura Risc em EPLDs, com um conjunto de instruções específico para uma certa aplicação. A idéia central é se realizar um processador dedicado que, através de seu conjunto de intruções dedicado, tenha desempenho superior a um processador tradicional. Com as metodologias de projeto e os recursos tecnológicos hoje disponíveis, a chave para produção de ASIPs em pequeno volume pode ser o PFGA.

### 5.4.1 O Risco EPLD

Para validação deste conceito foi realizada uma versão EPLD da família do processador Risco ([CAR 94], [SOU 95]). Uma versão 16 bits do processador foi utilizada como arquitetura alvo, pela disponibilidade de CAD e componentes durante o trabalho. A plataforma de CAD foi a linguagem AHDL do sistema Altera [ALT 93]. Basicamente fez-se uma transformação da descrição VHDL do Risco para AHDL, mudando-se o número de bits. O número de estágios do pipeline foi mantido, assim como os latches sensíveis a nível da versão CMOS full-custom.

Os primeiros resultados foram insatisfatórios. A família de EPLDs a disposição não era voltada para o uso de buffers de alta impedância para uso no barramento interno do microprocessador, devendo-se modificar o projeto para uso de multiplexadores. Além disto, a temporização baseada em latches também não era bem suportada pela família disponível. Sendo assim, modificou-se o sistema de barreiras temporais para uso de Flip-Flops sensíveis a borda, mas gerando-se as 3 fases originais do processador para manter a temporização compatível com a do Risco original.

Além destas modificações, a partição do sistema foi um trabalho difícil. O sistema de CAD e o próprio tipo de componente privilegiavam a síntese de FSMs, e não de máquinas complexas com sua Parte Operativa embutida. Isto significou uma grande

dificuldade na realização de circuitos típicos de microprocessadores, como a ULA e o banco de registradores. A solução encontrada foi a partição manual do banco de registradores e dos demais componentes do microprocessador. Como a arquitetura é Risc, basicamente o custo dos chips refere-se à Parte Operativa e ao banco de registradores.

No final do processo obteve-se um roteamento completo em dois integrados MAX7259, com a freqüência máxima de relógio de 4.7MHz sem otimização para velocidade ([SOU 95]). Apesar de aparentemente lento, é preciso considerar que na taxa de 1 instrução por ciclo este processador já é três vezes mais rápido do que um 8051 com um relógio de 12Mhz, executando instruções de apenas um byte e um ciclo. Embora o custo do processador seja elevado em comparação com o 8051, deve-se observar que se obteve um processador dedicado, o que significa que a idéia de um ASIP em EPLD/FPGA é válida. Além disto, com o aumento do valor agregado da aplicação, o custo do processador pode ser diluído, assim como se espera uma diminuição significativa dos custos dos FPGAs, maior que a redução de custos dos microprocessadores.

Recentemente, com a disponibilidade de famílias de FPGAs com barramentos de conexão interna dedicados à propagação de carrys, obteve-se um Risco 16 bits em dois chips 8820, ou em apenas um 81500, com 54% de ocupação. Isto significa que cada vez mais os ASIPs poderão ser prototipados e implementados definitivamente em tempos cada vez mais curtos. A figura 5.5 apresenta uma tela de simulação do Risco EPLD.

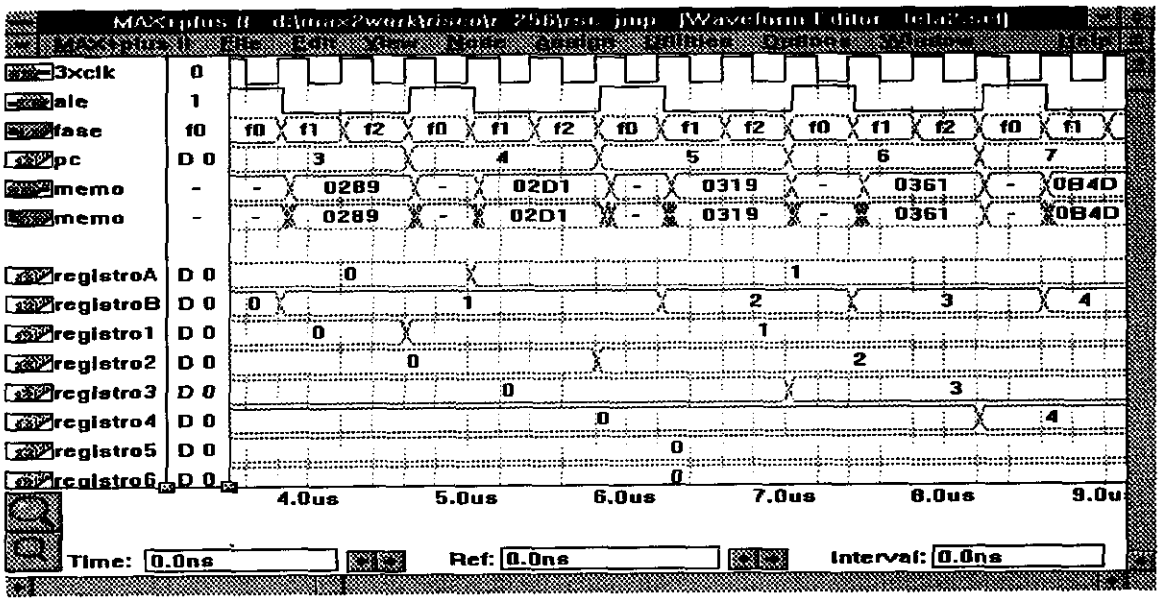



FIGURA 5.5 - Tela de simulação do Risco EPLD

## 5.4.2 O suporte de SW

Embora a realização fisica de processadores possa ser feita através de EPLDs ou FPGAs, não existe vantagem alguma em se ter um novo processador sem o suporte conveniente de SW. Muitas ferramentas são necessárias para completar a idéia de ASIPs. A figura 5.6 apresenta um diagrama de suporte de SW que está sendo desenvolvido para o processador Risco.

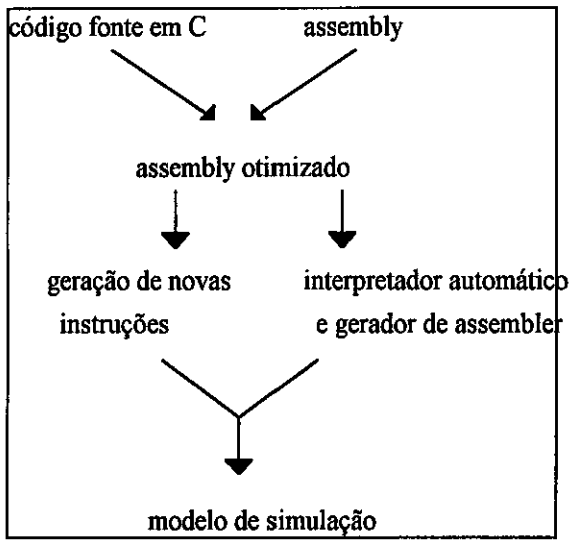



FIGURA 5.6 - Arquitetura do ambiente de SW para o processador Risco

Muitas das ferramentas da figura 5.6 já existem correntemente no ambiente ECSD. Notadamente, a análise de rotinas críticas e a estatística de utilização de instruções já se encontram disponíveis, assim como o Montador Universal ([LIM 94]). Este programa aceita um arquivo de configuração com as instruções disponíveis do processador, e monta uma saída binária a partir do fonte assembly dedicada ao processador alvo. Basicamente, podem-se alterar o número de registradores, o tamanho da palavra (número de bits) e as instruções do processador. Assim, pode-se incluir uma nova instrução no repertório de instruções do Risco original (MULT, para multiplicação, por exemplo) e, colocando-se seu código no arquivo de configuração, o montador gerará corretamente o SW.

As verificações de endereçamentos inconsistentes, acesso a registradores não suportados e outros problemas do fonte em relação às disponibilidades arquiteturais são também realizadas no ambiente do Montador Universal. A figura 5.7 apresenta o arquivo de configuração do Montador Universal, com inclusão da intrução MULT.

## 5.5 Limitações de processadores ASIPs baseados em famílias

As modificações na arquitetura da família de Riscos são apenas relativas a detalhes como o número de registradores, o número de bits da palavra de dados e instrução e a presença ou não de certas instruções. O pipeline do processador original foi mantido, assim como a arquitetura do barramento de acesso a memória. Modificações maiores poderiam produzir efeitos colaterias não previstos neste trabalho. Por exemplo, o uso de uma arquitetura Harvard com a separação do barramento de dados e endereços como o DLX poderia acelerar a execução de alguns programas, ao custo de mais 96 fios de conexão. O pipeline do processador deveria também ser modificado, o que poderia prejudicar programas em que a rotina crítica fosse condicional, como visto no capítulo 4.

```
ARQUIVO DE CONFIGURACAO

NUMERO DE BITS DO PROCESSADOR
#BITS = 16

NUMERO DE OPERANDOS
#OPERANDOS = 3

NUMERO DE REGISTRADORES
#REGISTRADORES = 8

TAMANHO MAXIMO PARA AS CONSTANTES
#CONSTANTES = 5

TAMANHO DO OPCODE
#OPCODE = 5

DEFINICAO DAS INSTR. SEUS OPCODES, NUMERO E TIPO DE OPERANDOS

TIPOS DE OPERANDOS

3 OPERANDOS

REG3 = 3 REGISTRADORES
REG2_CONST = 2 REGISTRADORES E 1 CONSTANTE
REGFB_CONST = 1 REGISTRADOR DESTINO,1 FB E 1 CONSTANTE
FB = REGISTRADOR DESTINO OU REGISTRADOR ZERO

2 OPERANDOS

REG2 = 2 REGISTRADORES
REG_CONST = 1 REGISTRADOR E 1 CONSTANTE

#INSTRUCOES
ADD    0 0 0 0 0  3  REG3 REG2_CONST REGFB_CONST
MULT   0 0 0 0 1  3  REG3 REG2
ADDC   0 0 0 1 0  3  REG3 REG2_CONST REGFB_CONST
SUB    0 0 1 0 1  3  REG3 REG2_CONST REGFB_CONST
SUBC   0 0 1 1 1  3  REG3 REG2_CONST REGFB_CONST
SUBR   0 1 0 0 1  3  REG3 REG2_CONST REGFB_CONST
SUBRC  0 1 0 1 1  3  REG3 REG2_CONST REGFB_CONST
AND    0 1 1 1 1  3  REG3 REG2_CONST REGFB_CONST
OR     0 1 1 1 0  3  REG3 REG2_CONST REGFB_CONST
XOR    0 1 1 0 1  3  REG3 REG2_CONST REGFB_CONST
RLL    1 0 0 0 0  3  REG3 REG2_CONST REGFB_CONST
RLLC   1 0 0 0 1  3  REG3 REG2_CONST REGFB_CONST
RLA    1 0 0 1 0  3  REG3 REG2_CONST REGFB_CONST
```

FIGURA 5.7 - Arquivo de configuração do Montador Universal

Um ponto importantíssimo concerne a disponibilidade de SW de uso do processador ASIP gerado. Embora a geração do processador possa ser realizada com um mínimo de intervenção manual (confiando-se em um programa de partição mais refinado do que os que hoje existem), o compilador C capaz de gerar código otimizado para o novo processador é também um desafio. Presentemente, o Montador Universal apenas garante que o programador do processador terá a sua disposição um assembly.

Contudo, o Compilador C ainda gera código para máquinas de 32 bits, e não otimiza o código para máquinas de 16 bits com outro subconjunto de instruções, por exemplo. Recentemente nota-se uma preoucupação exatamente com este item na comunidade internacional, como verificado em [WIL 94] e [LEU 95].

Embora a sintonia de um processador com uma aplicação traga significativas economia de recursos, resultados melhores podem ser obtidos quando se elimina a restrição de manter a estrutura de temporização do processador ASIP idêntica a do processador original. Esta restrição foi considerada no início do trabalho no instante em que era necessário manter a compatibilidade entre o processador original e sua versão integrada no sistema. Isto porque alguns programas poderiam fazer uso de interrupções cujas respostas fossem sincronizadas com a temporização original do processador.

Experimentos realizados com o processador Risco com modificações mais radicais de arquitetura tem demonstrado que, com área constante ou até menor que a do processador original, podem-se integrar funções de HW e memórias locais de programas, tornando o Risco um microcontrolador com arquitetura Risc ([ALB 95]).

Mesmo com limitações quanto a temporização, deve-se ressaltar o fato que as técnicas propostas neste capítulo são úteis não somente para a reengenharia de placas, tendo em vista a integração completa de um sistema, mas para prototipação de sistemas de baixo custo. Uma aplicação que possa ser executada no processador original pode depois migrar para um processador específico, com diminuição do espaço em placa, menor potência dissipada e aumento do valor agregado do integrado, que implementa um sistema, e não apenas um circuito.

# 6 Ferramentas de validação do ambiente ECSD

Este capítulo apresenta as diferentes ferramentas de validação que compõem o estado atual do sistema ECSD. Embora os conceitos expressos ao longo do trabalho exijam um ambiente integrado, no estado atual de desenvolvimento ainda é necessária alguma intervenção manual. Mesmo em seu estado corrente o sistema pode ser utilizado para auxílio no projeto de sistemas e na otimização dos mesmos.

## 6.1 Simulação e profiling

Como mencionado no capítulo 3, a partição entre SW e HW pode-se dar pela análise do programa em tempo de compilação ou em tempo de execução, determinando-se neste último caso um conjunto de rotinas críticas que podem ser depois otimizadas pelos conceitos expressos neste trabalho. Contudo, a simulação de um sistema HW e SW completo pode ser extremamente custosa em termos de tempo, dependendo da complexidade do projeto. A simulação para obtenção de dados de desempenho é portanto aplicável em alguns casos, e sob certas condições.

Uma primeira idéia é o uso de uma descrição VHDL do processador alvo do sistema e um compilador capaz de gerar código para este processador. Pode-se utilizar a máquina de simulação VHDL para executar diversos programas de teste do sistema em questão.

Embora factível, esta estratégia não é muito rápida em termos de execução, visto que, por mais elevado que seja o nível de descrição do processador, a máquina VHDL de simulação será sempre um gargalo, por ser a responsável pela velocidade através da gerência de eventos. Quando se deve simular um grande número de instruções para validar a proposta de sistema tem-se uma situação em que se pode colocar no limite as máquinas de simulação lógica hoje disponíveis. O problema fundamental, portanto, é o tempo de simulação quando se deseja trabalhar conjuntamente com HW e SW, dai a dificuldade de uso dos simuladores lógicos multiníveis atuais.

Outra solução possível seria o uso de um emulador da máquina alvo, como realizado no estudo do número de operações do 8051. A emulação é a execução do programa pelo próprio processador em uma placa onde se carrega o programa. Já foram propostos emuladores universais, aptos a emular qualquer processador através da microprogramação. Pelo acentuado tempo de execução provocado pela interpretação de conjuntos de instruções muito diferentes, esta idéia foi abandonada, ou o menos existem poucas referências a este respeito. A desvantagem da emulação para processadores ASIP reside no fato que, embora eficiente para profiling, depois da otimização não se tem onde executar a simulação das modificações arquiteturais de processadores propostas.

No caso do processador Risco, decidiu-se realizar a implementação de um simulador dedicado em C para a fase de profiling. Este simulador pode depois ser substituído por um emulador quando da montagem da placa do micro. Sobre este simulador podem ser realizadas estatísticas relativas a chamadas de funções e sobre o número de instruções efetivamente executadas.

O simulador foi pensado para ser posteriormente utilizado também na fase de validação do novo processador. Além disto, o simulador aceita como entrada um programa assembly do Risco. Entre os comandos tem-se breakpoints, inicialização de variáveis e registradores, execução passo a passo, etc, como um sistema de desenvolvimento normal.

A figura 6.1 apresenta uma saída parcial da tela de exibição do simulador do Risco, para o algoritmo de divisão inteira. Como o simulador é escrito em C sem modelos de atrasos de portas, sua velocidade de execução é elevada, permitindo a simulação de muitas instruções em um curto espaço de tempo. Comandos especiais para profiling foram inseridos, de modo a se detectar as rotinas críticas.

```
   Simulacao Passo a Passo (sbs) do arquivo div_int.ext

MAIN lab 0
 reg[3]:4   ADD r3 r0 4
 reg[2]:62   ADD r2 r0 62
 reg[4]:4   ADD r4 r3 r0
WHILE1 lab 0
 reg[1]:-58   SUB r1 r4 r2
 0 JMP PROX1
 0 NOP 0
 reg[4]:8   SLL r4 r4 1
 0 JMP WHILE1
WHILE1 lab 0
 reg[1]:-54   SUB r1 r4 r2
 0 JMP PROX1
 0 NOP 0
 reg[4]:16   SLL r4 r4 1
 0 JMP WHILE1
WHILE1 lab 0
 reg[1]:-46   SUB r1 r4 r2
 0 JMP PROX1
 0 NOP 0
 reg[4]:32   SLL r4 r4 1
 0 JMP WHILE1
WHILE1 lab 0
 reg[1]:-30   SUB r1 r4 r2
 0 JMP PROX1
 0 NOP 0
```

FIGURA 6.1 - Tela de simulação do algoritmo de divisão inteira

Note-se que o profiling é dependente do simulador disponível. Este deve ser capaz de identificar as chamadas de rotinas e calcular os tempos de execução isoladamente, tendo em vista que está-se interessado no tempo individual de cada rotina, e não no tempo global de uma rotina que chama outras.

## 6.2 Simulação e validação

Um sistema computacional raramente existe isolado, deve-se estudá-lo também junto ao meio onde ele irá atuar. A validação do sistema computacional passa portanto pela validação do sistema em um certo ambiente. Isto significa a modelagem do ambiente de atuação do sistema e a simulação conjunta deste com os algoritmos em questão.

O sistema ECSD modifica uma descrição do usuário, além de gerar novos componentes no sistema original. Portanto, a validação do novo sistema passa pela execução dos programas modificados e pela execução concorrente da simulação da função em HW.

### 6.2.1 O problema da simulação conjunta de HW e SW

O problema da simulação conjunta de HW e SW recebeu atenções mais recentemente devido aos desenvolvimentos em HW-SW codesign, visto que os tempos de execução de um e outro diferem por algumas ordens de grandeza.

Em [BEC 92] é utilizado um simulador comercial (Verilog) sobre o qual foi estabelecida uma camada de SW em Unix para comunicação entre os processos que procedem à simulação. Segundo reportado, quando da chegada do HW real, bastou a troca de algumas rotinas de comunicação para que todo o sistema pudesse ser emulado. A solução encontrada por Becker et al. contudo não contempla a análise de tempos relativos de execução entre SW e HW.

Gupta, em [GUP 92a], utiliza um simulador de eventos que gerencia a simulação da descrição do microprocessador e do HW dedicado. O gerenciador não é uma entidade física (como um barramento, por exemplo), mas um gerente de comunicações virtuais dos protocolos previamente declarados entre o micro e a função de HW.

Para a simulação de uma central telefônica, o trabalho de Loucks et al. utilizou um conjunto de workstations [LOU 93]. No caso, o emulador de processador não existe fisicamente, pois o SW que este deveria executar é compilado e executado na própria workstation. Apesar de rápido, uma conversão de código é depois necessária, e portanto requer-se uma nova fase de verificação. A maior vantagem de tal abordagem é que simulações de interrupções em tempo real podem ser obtidas. É reportado inclusive que o sistema de desenvolvimento respondeu mais rápido do que o sistema real, o que comprova uma necessidade de verificação posterior.

Em [RET 93] apresenta-se um modelo mais simples das possibilidades de simulação. Uma biblioteca de módulos de HW descrita em VHDL é compilada junto a um conjunto de programas em ADA, e uma simulação do sistema global é levada a cabo pela máquina de simulação do sistema de suporte de VHDL. Este é provavelmente o mais lento dos métodos conhecidos.

A solução mais adequada ao desenvolvimento conjunto de SW e HW parece ser o Mpsas ([SUN 91], [SUN 91a]), da SUN Corporation. Através deste conjunto de programas pode-se simular um núcleo Sparc junto com qualquer HW dedicado que se queira. Basicamente, o simulador é constituído por um modelo de barramento e processador. Todas as conexões são realizadas através do barramento, pelo uso de portas especificadas pelo usuário. O simulador é de código compilado, e aceita como entradas seja uma topologia diferente de conexões ao barramento seja um programa C que simula uma função em HW.

Note-se que através de Mpsas a velocidade de simulação é a maior possível. Isto porque a máquina hospedeira (uma estação de trabalho SUN) possui o mesmo processador para o qual se está realizando um projeto conjunto de SW e HW. Portanto,

a máquina alvo é praticamente emulada, e não simulada. A única dificuldade aparente é quando da definição de uma modificação no processador, pois então novas instruções ou outras modificações deverão ser novamente simuladas, e não mais emuladas. Em seu estado atual Mpsas não prevê modificações na arquitetura do processador como inclusão de instruções, por exemplo.

Pelo fato do computador hospedeiro possuir o mesmo processador do sistema alvo, tem-se a possibilidade do uso de programas já disponíveis no sistema operacional SUNOS para realização da partição HW/SW. Por exemplo, uma idéia razoavelmente próxima do real das rotinas críticas de uma aplicação pode ser fornecida pelo programa **prof**. Contudo, para uso deste programa deve-se compilar o fonte sem otimizações, o que pode levar a resultados errôneos.

### 6.2.2 O ambiente de validação em HDC

O processador Risco possui uma série de ferramentas de SW de suporte, como montador ([LIM 94]) e simulação através da linguagem HDC ([MAR 92]). Sobre a descrição HDC do Risco foi criada uma interface para permitir um ambiente mais confortável para o desenvolvimento de SW. No caso, não é necessária a visualização de sinais internos do microprocessador pelo seu programador, mas apenas o conteúdo de registradores e memória. Para observação de nós internos pode utilizar a descrição do Risco em [JUN 93].

A figura 6.2 mostra uma tela de uma simulação SHC (Simulador de HDC) do Risco, enquanto a figura 6.3 mostra a tela do simulador em ECSD. Sob a interface gráfica da figura 6.3 existe ainda a máquina de simulação SHC, porém limitada ([CAR 93]). Não foram utilizados todos os modos de atraso do SHC, mas apenas o atraso unitário, já que não se quer apenas validar o processador, mas também o SW que se deseja executar.

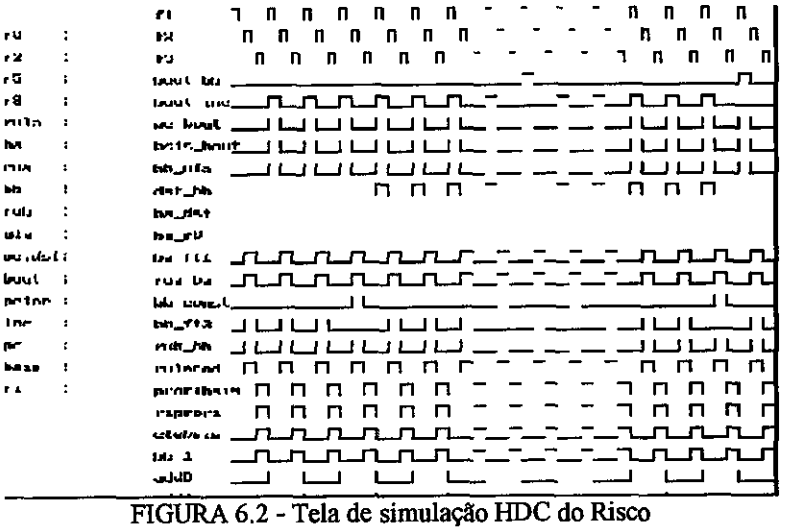


FIGURA 6.2 - Tela de simulação HDC do Risco

Como o simulador SHC é de código compilado, foram feitas modificações para que se pudessem carregar memórias distintas, de modo a simular diferentes programas fontes sem recompilação do código de simulação.

Um problema interessante foi a colocação de break-points em código de processadores que usam pipeline. Neste caso, adotou-se uma política compatível com um futuro sistema de desenvolvimento físico. No breakpoint, a rotina controladora de simulação quebra o pipeline com inserção de operações vazias (nops), e entra em espera. Ao avançar-se além do breakpoint, o pipeline estará repleto de nops, mas o programa prosseguirá com os dados corretos. Esta é a mesma técnica de atendimento de interrupção no Risco ([JUN 93]).

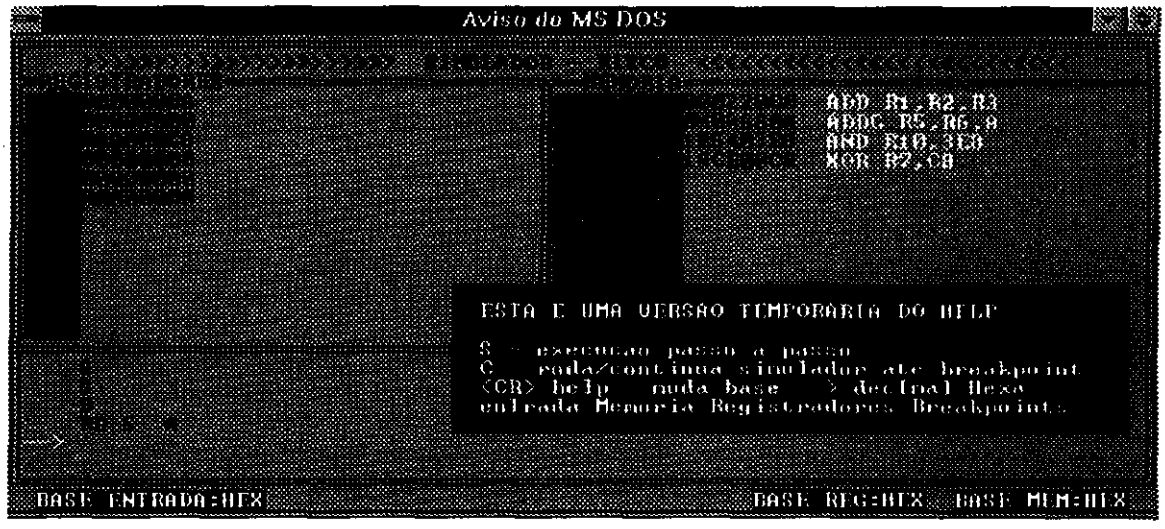



FIGURA 6.3 - Tela do sistema de desenvolvimento

Resta porém em aberto o problema da simulação do HW dedicado. Neste caso, a descrição comportamental da função de HW foi gerada para a ferramenta de síntese. A saída da ferramenta de síntese possivelmente em nível lógico deve ser utilizada como entrada para uma descrição HDC da função em HW sintetizada. Contudo, sobre esta descrição ainda podem existir vários níveis de interesse.

Por exemplo, do ponto de vista da execução do SW, bastaria saber quantos ciclos a função leva para executar. Portanto, uma vez obtida a síntese, não necessariamente deve-se simular com o netlist de nível lógico, mas pode-se simular o grafo de dependências modificado pela síntese e criar os eventos corretos de sincronização com o microprocessador (se o número de ciclos para a comunicação é conhecido em tempo de compilação).

O HW dedicado pode ser visto também em diversos níveis de abstração. A validação deste HW pode requerer que sinais analógicos sejam produzidos na entrada de um subcircuito, enquanto que o microprocessador executa o SW pertinente ao sinal analógico de entrada (filtros digitais, por exemplo).

Mesmo uma netlist lógica pode requerer muito mais tempo para validação do que o simulador do microprocessador, além de trazer problemas do ponto de vista de ambiente de simulação. Um evento em uma rede lógica pode ser produzido a cada nanosegundo ou menos, enquanto que a simulação do SW produz eventos apenas síncronos com o relógio do micro, em dezenas ou centenas de nanosegundos. A simulação do HW dedicado e do microprocessador em conjunto envolve portanto um gerenciador de eventos global.

Este gerenciador existe na realidade, e é o próprio barramento do sistema. Qualquer sinal espúrio ali presente pode provocar comportamentos não esperados do micro, da memória, de periféricos e do HW dedicado. Esta idéia é uma extensão do sistema Mpsas, onde cada subsistema tem a sua máquina dedicada de simulação, que se comunica globalmente via o barramento, que é o subsistema utilizado no sistema físico real. O que se propõe é que cada máquina de simulação possa atuar separadamente dependendo do nível de refinamento exigido pelo projetista.

Todo o ambiente de simulação aqui proposto baseia-se na possibilidade de substituição futura do modelo do processador por uma placa com o Risco-ASIP onde seria feita a emulação de programas.

Foi implementado um simulador de Risco baseado na máquina SHC com o barramento servindo de mestre da simulação, como acima proposto. Os resultados de tal arquitetura de simulação, porém, não foram significativos quanto à velocidade de simulação. Embora o simulador SHC suportasse diversos níveis de descrição e a ligação através do barramento do micro, a velocidade de execução do código era extremamente lenta. Consequentemente, a execução de um programa para profiling era impossível na prática. Em termos de validação do sistema projetado com um eventual HW dedicado o simulador é útil, mas somente para execução de poucas rotinas que efetivamente chamam a função de HW.

### 6.2.3 O Ambiente de validação e simulação em C do Risco

Tendo em vista as dificuldades de simulação conjunta de SW e HW no ambiente SHC, e na especificidade de um simulador C apenas para o Risco, decidiu-se expandir o simulador do processador para aceitar a memória WCS e eventuais funções de HW. Esta modificação permite a validação de circuitos com a WCS, WCS-PIPE, e avaliação da otimização, ao mesmo tempo que garante uma velocidade de execução razoável da simulação.

O simulador foi modificado para aceitar um código que é carregado em tempo de execução para a WCS. Uma vez encontrado o comando de ativação da WCS no assembler a função de simulação passa ao controle da WCS, que executa sem acessar a área da memória principal, como no micro real.

Para simulação do WCS-pipe teve-se de incluir uma função de HW. Neste caso, como por definição o HW dedicado terá a mesma Parte Operativa do processador, o simulador apenas reproduz a Parte Operativa do microprocessador no HW dedicado. A simulação procede normalmente, e sempre que a função de HW é ativada o simulador executa instruções em ambas as Partes Operativas para depois atualizar os ciclos de processamento. Um exemplo de simulação com a WCS é mostrado na figura 6.4, para o algoritmo de divisão inteira. Resultados de outras simulações podem ser encontrados no anexo 6.

### 6.2.4 O Ambiente de validação e simulação em VHDL do Risco

Existem versões do processador Risco e Risco WCS em VHDL. Portanto, após a movimentação de operações ou após a escolha de uma estratégia de otimização podem-se realizar simulações com a máquina de VHDL para validação da movimentação de operações.

A dificuldade aqui reside no fato que a máquina de simulação VHDL é lenta, por ser baseada em eventos. A vantagem de se usar tal sistema é a mesma do SHC, ou seja, pode-se ter em um mesmo ambiente o processador, seu conjunto de programas e eventuais funções em HW em vários níveis de descrição e detalhe. Contudo, devido ao tempo de simulação, este tipo de ambiente serve apenas para validação do HW dedicado, e não para profiling ou validação do sistema como um todo.

```
   Simulacao Passo a Passo (sbs) do arquivo div_int.w

 MAIN lab 0
Wcs On
reg[3]:4   ADD r3 r0 4
reg[2]:62   ADD r2 r0 62
reg[4]:4   ADD r4 r3 r0
  WHILE1 lab 0
reg[1]:-58   SUB r1 r4 r2
reg[4]:8   SLL r4 r4 1

...

  WHILE1 lab 0
reg[1]:2   SUB r1 r4 r2
  PROX1 lab 0
reg[5]:62   ADD r5 r2 r0
reg[6]:0   ADD r6 r0 r0
  WHILE2 lab 0
reg[1]:60   SUB r1 r4 r3
reg[6]:0   SLL r6 r6 1
reg[4]:32   SRL r4 r4 1
reg[1]:30   SUB r1 r5 r4
reg[6]:1   ADD r6 r6 1
reg[5]:30   SUB r5 r5 r4
  WHILE2 lab 0
reg[1]:0   SUB r1 r4 r3
  FINAL lab 0
reg[1]:0   nop 0 n 0
Wcs Off
0 NOP 0
```

FIGURA 6.4 - Tela com simulação WCS

## 6.3 O Sistema de Desenvolvimento moderno

Dos conceitos anteriormente expostos, é importante verificar que, no que tange o projeto conjunto de HW e SW, tem-se a necessidade de definir a versão atualizada dos sistemas de desenvolvimento utilizados correntemente para trabalhos em circuitos microprocessados. A possibilidade do uso de FPGAs para prototipação rápida e execução de instruções de um processador modificado levam à factibilidade da construção de uma placa universal de processador, em que a arquitetura escolhida do processador seja a decidida pelo sistema de CAD ([SOU 95]).

Mesmo nos casos em que não se realizem modificações arquiteturais, pode-se necessitar de um ambiente de validação prevendo o uso de funções dedicadas. A validação destas funções dedicadas é tarefa suportada pelo ambiente de desenvolvimento. A função pode ser implementada em um FPGA ou avaliada por uma máquina de simulação diferente daquela do processador ([BEN 95]).

A questão da validação de um projeto conjunto de HW e SW está ainda aberta. Das soluções citadas muitas são específicas, e não atendem casos gerais. Outras são independentes do processador alvo, mas lentas na execução do SW. É possível que a solução para estes sistemas esteja na prototipação rápida de processadores para emulação, através de placas dedicadas, como as da Aptix ([GUO 92]) ou através da realização do processador em um único FPGA, como os recentemente disponíveis na família FLEX10K da Altera, com 100000 gates, ou 62000 gates e 24576 bits de RAM. ([ALT 95]). Neste caso, a descrição do novo procesador e de sua função de HW seria passada a uma placa de FPGAs que emulariam o processador alvo e seu HW dedicado. O SW a ser executado seria carregado em memória RAM local à placa, que poderia se comunicar com um computador hospedeiro como os sistemas de desenvolvimento atuais.

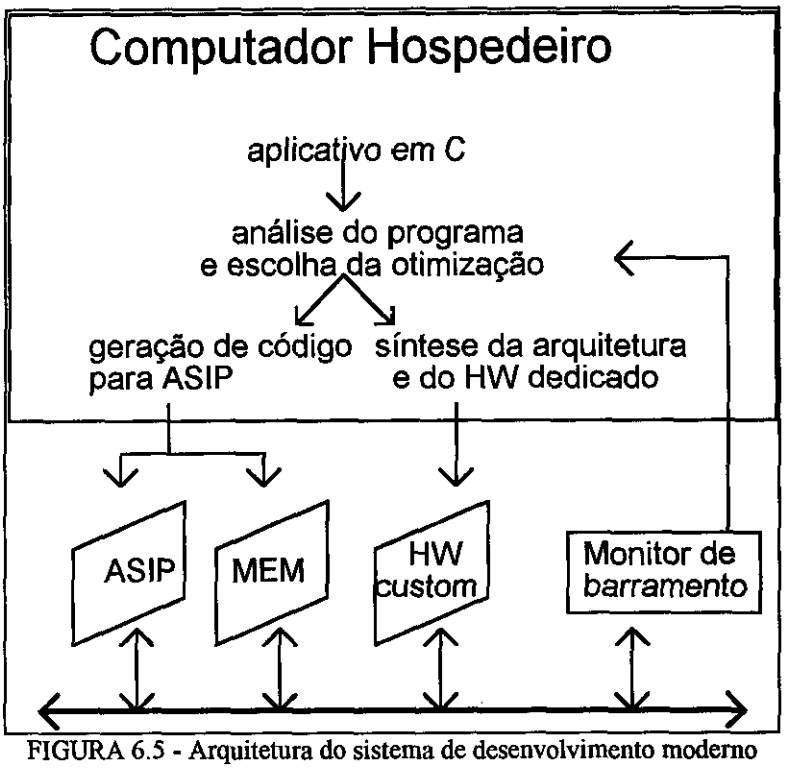



FIGURA 6.5 - Arquitetura do sistema de desenvolvimento moderno

A figura 6.5 apresenta uma proposta para o Sistema de Desenvolvimento moderno. Note-se que seu princípo básico é o mesmo dos sistemas de desenvolvimento microprocessados atuais. A diferença fundamental encontra-se na disponibilidade da definição de uma função de HW e de um processador dedicado. Este grau de liberdade a mais é importante, na verdade fundamental, pois poderá cobrir aplicações que utilizem processadores tão distintos quanto o 8051, Risco ou DLX

# 7 Trabalhos futuros e conclusão

Este trabalho procurou estudar diversas soluções para o problema da implementação de sistemas computacionais. Evidentemente, pelo escopo do trabalho ser abrangente, muito ainda resta a ser feito. Neste capítulo apresentam-se futuros trabalhos e sugestões para modificações no sistema de CAD tendo em vista os problemas encontrados nesta primeira versão.

## 7.1 Limitações de ECSD

Presentemente, apenas uma rotina crítica é analisada por vez. Evidentemente, para grandes sistemas, é possível que sob certas situações dinâmicas a ênfase do sistema encontre-se em duas ou três rotinas diferentes. Presentemente, é missão do usuário trabalhar com mais de uma rotina crítica, instruindo o sistema o que fazer em cada caso. Embora múltiplas rotinas críticas sejam possíveis, seu número será forçosamente baixo, pela hipótese inicial de se estar trabalhando para realização de um sistema computacional dedicado, visto que o caso geral já é devidamente coberto pelos microprocessadores de uso comum.

Uma vez que o HW dedicado foi gerado, este deverá ter um custo associado. Eventualmente, esta função de HW poderia atender a todas as rotinas críticas candidatas, não só a algumas. Neste caso, o HW pode ser caro. Contudo, se as funções de HW forem chamadas em tempos distintos, somente uma estará ativa em um dado instante de tempo, podendo-se agrupar operações de HW comuns a mais de uma função, de modo a se obter um HW dedicado mais barato. Na sua versão atual ECSD prevê apenas um HW dedicado, mas esta extensão de implementação é factível e desejável. Uma possível realização seria a produção de uma descrição VHDL em que as máquinas seriam agrupadas sob diferentes comandos. A ferramenta de síntese estaria naturalmente encarregada da otimização do HW, tendo em vista a exclusividade das operações no tempo.

## 7.2 Modificações a serem feitas no trabalho

Durante o desenvolvimento de ECSD foram abertas novas portas para pesquisas em CAD e arquitetura. Existem modificações que podem ser rapidamente executadas, outras que exigirão novos estudos e praticamente dão um rumo sobre a continuidade do trabalho.

### 7.2.1 Modificações quanto a linguagem de entrada

Um dos primeiros problemas relativos a ECSD é a falta de restrições temporais em sua linguagem de entrada. Por exemplo, o projetista hoje não pode especificar que um trecho de código deve responder em no máximo X nanosegundos. Contudo, apesar de existirem trabalhos em que a linguagem C foi modificada ([ERN 93]), na verdade é possível que se deseje um nível de abstração ainda mais alto, e neste nível colocar as restrições temporais. Por exemplo, a linguagem SDL tem sido usada para descrições de sistemas de telecomunicação, tendo em vista especificações do CCITT ([SAR 87]). Programas que traduzem de SDL para módulos em C existem comercialmente, e

recentemente trabalhos que traduzem de SDL diretamente para VHDL também estão diponíveis ([ISM 94]).

Outra possiblidade de entrada de mais alto nível seria o uso de uma linguagem voltada a objetos, que depois seriam mapeados para HW ou SW. Os objetos são descritos por uma linguagem suficientemente abstrata para que não haja uma polarização entre HW e SW desde o início da descrição. Um exemplo desta metodologia pode ser encontrado em [WOO 94], onde a linguagem de entrada é baseada em conjuntos de funções e classes em C++.

A especificação de sistemas computacionais ainda é um campo fértil de pesquisa. Isto porque presentemente nenhuma das propostas contempla funções abstratas o suficiente para não serem dependentes da aplicação alvo. Por outro lado, linguagens de entrada como C, mesmo com restrições temporais, dificultam a descrição ou a partição do sistema quando componentes analógicos entram em jogo.

A opção feita neste trabalho por uma linguagem assembler ou C justifica-se para o tipo de sistema alvo proposto, visto que se deseja atender a uma aplicação específica, e portanto busca-se uma versão ligeiramente alterada do sistema original. Além disto, sem um ambiente de CAD onde, depois de feita a partição HW-SW, o projetista pudesse testar ou emular a estratégia escolhida, nada poderia ser feito. Assim, entradas de níveis superiores utilizarão ECSD como ambiente de realização, avaliação e validação, assim como presentemente os ambientes de High Level Synthesis utilizam programas de síntese lógica como um de seus passos intermediários.

### 7.2.2 Modificações quanto a arquitetura alvo

Neste trabalho foram explorados somente alguns aspectos da contrução de sistemas computacionais. Dado um sistema, pode-se analisá-lo segundo dois métodos. No primeiro caso, busca-se analisar o código para verificar qual o HW que deve ser agregado a um processador standard para melhora de desempenho. No segundo caso, pode-se analisar o algoritmo e verificar qual o melhor processador que se adaptaria ao conjunto de algoritmos. Somente o primeiro método foi objeto de estudo neste trabalho. Claramente há espaço para exploração do segundo método, pois a realização de microprocessadores dedicados é perfeitamente viável nos dias de hoje.

Nada impede ao projetista que assim o desejar a criação de outras instruções além daquelas presentes no microprocessador escolhido. As primeiras candidatas seriam seqüências de operações muito utilizadas sugeridas como novas instruções.

Um exemplo desta técnica encontra-se na figura 7.1. Já que a seqüência de instruções and e add é utilizada muitas vezes, se fosse possível realizá-las uma única vez sem atrasos significativos no relógio original ter-se-ia um ganho no desempenho do processador para a rotina em questão.

Esta solução só é possível com o uso de processadores dedicados, onde o HW da Parte Operativa pode ser configurado, assim como a Parte de Controle deve decodificar novas instruções. Além disto, requer-se do ambiente de SW o suporte do novo conjunto de instruções de maneira compatível, ou seja, a seqüência de instruções deve executar como se fosse apenas uma no ambiente de simulação.

```
                                        add     res,r0,r0
                                        add     i,r0,r0
res = 0;                                xor     xor,i1,i2
i = 0;                          w_1:    sub.aps r0,i,16
xor = i1 ^ i2;                          jmp.GE  fim
while( i< 16) {                         nop
        res = res + (xor & 1);          and     aux,xor,1
        xor = xor >> 1;                 add     res,res,aux
        i = i + 1;                      sra     xor,xor,1
        }                               add     i,i,1
return (res)                            jmp     w_1
                                        nop
                                fim:    ret     res
```

FIGURA 7.1 - Algoritmo de Hamming code

Apesar de ser possível requerer do usuário a decisão se vale a pena dispor da nova instrução ou não, a busca da seqüência de operações que resultará em uma nova instrução pode ser complexa. Um algoritmo capaz de localizar padrões em todo o código do programa poderia avaliar o efeito que a nova instrução provocaria na aceleração ou não de diversas partes do código. A estratégia básica é a comparação de padrões de sub-grafos em grafos de controle de algoritmos. Apesar deste procedimento ser custoso, algumas técnicas heurísticas como as propostas em [SRE 93] podem ser utilizadas para verificação das possibilidades. Já existem trabalhos, como em [ONI 95], onde o compilador da linguagem de entrada é utilizado para ajudar na decisão da arquitetura do processador. Assim, um padrão de instrução que é repetido muitas vezes tende a tornar-se uma instrução do processador ASIP.

Se o processador for produzido full-custom, qualquer modificação acarretará custos de fabricação relativamente significativos. Contudo, se o meio de implementação for um GA ou FPGA, pode-se ter um menor custo de inserção de modificações. O custo do processador final dependerá não somente dos custos de desenvolvimento, mas do volume de produção também. Quanto maior a modificação necessária, maior o custo. O usuário tem controle então do custo do sistema conforme a necessidade de desempenho do SW e conforme o volume desejado.

Novas tecnologias de SW como retargetable compilers generators ([LEU 95]) podem ajudar a resolver a questão da geração do compilador para tais processadores. Atualmente os geradores de retargetable compilers utilizam como entrada um descrição estrutural do processador e sobre esta deduzem uma série de regras para geração de código. Propõe-se seguir o caminho contrário, ou seja, tendo em vista que o processador será gerado para uma aplicação sob algum critério de especificidade (para uma certa rotina crítica, por exemplo), então pode-se gerar o compilador C ao mesmo tempo em que se gera o processador, e não depois.

A expansão do presente trabalho tendo em vista a geração de ASIPs dedicados já foi iniciada, e resultados preliminares encontram-se em [ALB 95]. Basicamente, o compilador C de entrada do sistema está sendo modificado para escolher a arquitetura alvo do processador que melhor se adapta ao sistema em questão. As arquiteturas disponíveis são aquelas versões do Risco apresentadas neste trabalho, além de modificações arquiteturais mais significativas, como alteração de pipeline, de número de barramentos e do conjunto de instruções. A tabela 7.1 apresenta resultados para 2 exemplos, o controle do motor e o controle de válvulas. Embora estas arquiteturas

tenham sido geradas manualmente, está-se presentemente realizando as ferramentas que fazem parte do sistema completo. Note-se que para ambas as aplicações o uso de memórias locais torna o Risco um microcontrolador-ASIP.

TABELA 7.1 - Resultados do Risco como microcontrolador-ASIP

| | **Controle do Motor** | **Profibus** |
|---|---|---|
| ROM necessária | 693 palavras | 879 palavras |
| RAM necessária | 71 palavras | 81 palavras |
| ROM gerada | 1024 palavras, 256p-128b | 1024 palavras, 256p-128b |
| RAM gerada | 128 palavras, 128p-32b | 128 palavras, 128p-32b |
| tamanho da ROM | 1496x1900, 2.84mm$^2$ | 1496x1900, 2.84mm$^2$ |
| tamanho da RAM | 2314x1322, 3.06mm$^2$ | 2314x1322, 3.06mm$^2$ |

### 7.2.3 Modificações de ordem mais geral

Uma outra modificação necessária é a expansão do sistema ECSD para implementações que transcendam os sistemas computacionais digitais. A inclusão de circuitos analógicos é uma chave fundamental para a real integração de sistemas. Apesar do crescimento dos circuitos integrados digitais, ainda hoje existem aplicações analógicas. Por exemplo, em CD players, circuitos de comunicação e telefonia celular, etc. Embora a parte analógica de um sistema tenda a ficar entre 10 a 20% do sistema final, ela provelmente sempre existirá, pois as grandezas físicas do mundo em geral possuem comportamento analógico, e como tal quase sempre algum tipo de conversão é necessária.

Cada vez mais métodos de projeto analógico deixam de ser um segredo conhecido por poucos e tornam-se disponíveis a projetistas de circuitos através de CADs eficientes. Conseqüentemente, a possibilidade de realizar certos circuitos de forma analógica é interessante, e esta possibilidade certamente deverá ser levada em conta quando do particionamento do sistema. Por exemplo, em um ambiente industrial, um controle PID de um processo pode ser facilmente realizado de maneira analógica com um conjunto de amplificadores operacionais, enquanto que o microprocessador pode se situar como um veículo de comunicação entre diferentes PIDs através de um protocolo padrão. A especificação, particionamento e realização física de tal sistema é ainda um desafio para os projetistas de arquiteturas e CAD.

Esta expansão de ECSD para abranger a implementação de circuitos analógicos baseia-se fortemente no conceito de ASIPs. A potencial economia de área dos processadores dedicados (Risco ou 8051), aliada à especialização de funções do HW abrem espaço para integração em uma mesma pastilha de circuitos mistos, tornando viável o desenvolvimento de sistemas complexos completos.

## 7.3 Conclusão

Neste trabalho procurou-se lançar as bases para uma pesquisa continuada na área de projeto de sistemas. Algums problemas foram devidamente estudados, outros foram apenas mencionados para trabalhos futuros. Devido ao amplo espectro do trabalho, algumas ferramentas encontram-se prontas ao uso, outras ainda não estão concluídas, e ainda não é possível o uso do sistema de CAD como um todo sem intervenção do projetista em certos momentos. Contudo, apesar destas limitações, acredita-se que os

objetivos propostos no início do trabalho foram atingidos. Realizaram-se estudos sobre arquiteturas e a validação das mesmas para o projeto de sistemas computacionais, assim como estabeleceu-se um caminho para a protipação e reaproveitamento de HW já existente. Espera-se que este trabalho tenha continuidade para ampliação de seus limites, seja em termos de CAD para otimização de sistemas, seja quanto a novas maneiras de se implementar sistemas integrados.

Com as tecnologias atuais e com os recursos hoje disponíveis de CAD existe um amplo espaço de projeto que pode ser explorado por projetistas de sistemas. A crise da eletrônica será vencida pelo aumento do valor agregado de seus produtos, e esta atitude passa necessariamente pela integração de sistemas complexos em um único CI, ou em um conjunto de CIs dedicados. A partir deste trabalho abre-se uma janela de pesquisas para muitos anos, pesquisa esta que pode perfeitamente estar sintonizada com a realidade do parque industrial nacional.

# Anexo 1 Conjunto de instruções do Risco

O processador Risco foi desenvolvido no CPGCC da UFRGS. Sua arquitetura foi realizada durante um trabalho de mestrado ([JUN 93]), e o processador teve diversas implementações, em full-custom ([CAR 94a]), EPLD ([SOU 95]) e Standard Cell ([BUR 95]). Neste anexo apresentam-se apenas algumas características arquiteturais importantes para que o eleitor possa ter um bom acompanhamento deste trabalho.

1)Características básicas

O Risco em sua versão básica é um processador de 32 bits, com capacidade de endereçamento a palavra. O processador possui 32 registradores de 32 bits, sendo que alguns registradores tem uma função especial. Por exemplo, o registrador R0 é a contante zero, e ele não pode ser escrito. O registrador R1 armazena a palavra de status nos seus 5 bit menos significativos.

O processador foi pensado para ser de muito baixo custo, e portanto optou-se por realizá-lo em um encapsulamento de 48 pinos. Deste modo, como o endereçamento é feito a palavra, 32 pinos já são gastos no endereço, e então implementou-se um barramento multiplexado de dados e endereços. Existe assim a necessidade de se implementar um latch externo ao processador para manter o endereço estável quando do acesso à memória.

2)O conjunto de intruções

Todas as instruções do Risco são no formato operação, destino, fonte 1 e fonte 2. Os operadores fonte são no geral registradores, sendo que fonte 2 pode ser uma constante de 11 bits em complemento de 2. Este é um resumo, contendo as instruções utilizadas ao longo dos exemplos do trabalho. Maiores detalhes podem ser encontrados em [JUN 93].

| | |
|---|---|
| and | dst = ft1 & ft2 |
| add | dst = ft1 + ft2 |
| addc | dst = ft1 + ft2 + carry |
| jmp | r31 = ft1 + ft2 |
| ld | dst = M[ft1 + ft2] |
| ldpod | dst = M[ft1 + ft2], ft2 = ft2 - 1 |
| ldpoi | dst = M[ft1 + ft2], ft2 = ft2 + 1 |
| or | dst = ft1 \| ft2 |
| sla | dst = ft1 << ft2, bit 31 preservado |
| sll | dst = ft1 << ft2 |
| sr | M[dst] = r31, r31= ft1 + ft2, dst = dst -1 |
| sra | dst = ft1 >> ft2, bit 31 prservado |
| srl | dst = ft1 >> ft2 |
| st | M[ft1 + ft2] = dst |
| stpod | M[ft1 + ft2] = dst, ft2 = ft2 - 1 |
| stpoi | M[ft1 + ft2] = dst, ft2 = ft2 +1 |
| xor | dst = ft1 ^ ft2 |
| sub | dst = ft1 - ft2 |

Uma operação seguida de um sufixo aps força a atualização da palavra de status do processador (r01). O registrador r0 é a constante zero (0), enquanto que o registrador R31 é o contador de programa. O apontador de pilha pode ser qualquer registrador, no compilador escolheu-se r30. Quando r0 é o destino o resultado é perdido.

3)O pipeline do processador

O Risco possui um pipeline de 3 estágios de máquina. Internamente, o processador necessita de 3 fases para realização de operações. A figura A1.1 abaixo retrata o pipeline do Risco. Note-se que duas fases são perdidas para espera do dado que vem da memória. Isto foi feito para permitir o uso de memórias lentas para diminuição do custo total do sistema.

Como o barramento de dados e endereços é único, a cada instrução de load ou store é preciso primeiro endereçar a memória, para somente depois ler o dado. A memória de instrução é a mesma de dados, e portanto deve-se interromper o fetch da próxima instrução para realização do load/store. Esta situação também encontra-se representada na figura A1.1.

Os estágios do pipeline são: E: coloca o endereço para fetch; I1 e I2: espera acesso à memória: D: decodifica instrução; R: lê operandos do banco de registradores; O: opera registradores na ULA; W: escreve o resultado da operação no banco de registradores. Nos casos em que há dependência de resultados para realização de instrução de salto o Risco usa a técnica do delayed-branch, ou seja, a instrução depois do salto é sempre executada. Portanto, é responsabilidade do programador a inserção de nops ou de uma instrução útil após cada instrução de jump.

| i | E | I1 | I2 | D | R | O | W | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| i+1 | | | | E | I1 | I2 | D | R | O | W | | | | | | | | | |
| i+2 | | | | | | | E | I1 | I2 | D | R | O | Me | MI | M2 | Mw | | | |
| i+3 | | | | | | | | | | E | I1 | I2 | | | | D | R | O | W |
| i+4 | | | | | | | | | | | | | | | | E | I1 | I2 | D |
| i+5 | | | | | | | | | | | | | | | | | | | E |

*FIGURA A1.1 - Pipeline do Risco*

## Anexo 2 Conjunto de instruções do DLX

O conjunto de instruções abaixo foi incluído para uma comparação rápida com aquele do processador Risco. Estas instruções foram aquelas implementadas na descrição VHDL do DLX. Detalhes sobre os tipos de instruções podem ser encontrados em /HEN 90/. Todos os imediatos são de 16 bits.

Instruções de transferência de dados

- LB, LBU, SB carrega byte, carrega byte unsigned, armazena byte;
- LH, LHU, SH carrega meia-palavra, carrega meia-palavra sem sinal, armazena meia-palavra;
- LW, SW carrega palavra, armazena palavra;

Instruções aritméticas e lógicas

- ADD, ADDI, soma, soma imediato;
- ADDU, ADDUI soma sem sinal, soma imediato sem sinal;
- SUB, SUBI subtrai, subtrai imediato;
- SUBU, SUBUI subtrai sem sinal, subrai imediato sem sinal;
- AND, ANDI função E lógico, E lógico imediato;
- OR, ORI, função OU lógico, OU lógico imediato;
- XOR, XORI função OU-esclusivo, OU-esclusivo imediato;
- LHI carrega parte alta de um registrador com imediato;
- SLL, SRL deslocamento à esquerda e à direita, lógico;
- SLA, SRA deslocamento à esquerda e à direita, aritmético;
- SLLI, SRLI deslocamento à esquerda e à direita, lógico com imediato;
- SLAI, SRAI deslocamento à esquerda e à direita, aritmético com imediato;
- S_, S__I set condicional para saltos, pode ser EQ,NE,LT,GT,LE,GE;

Instruções de controle

- BEQZ, BNEZ salta se registrador igual ou não a zero;
- JR salta para um offset de 26 bits do PC, ou salta para o conteúdo de um registrador;
- TRAP transfere para o sistema operacional, através de endereço vetorado;

As instruções abaixo foram incluidas para acionar a WCS

WCS_T_ON : liga a WCS

WCS_BACK : desliga a WCS

# Anexo 3 Descrição AHDL do Risco

No disquete em anexo encontra-se o diretório Risco. Neste diretório foram colocadas as descrições em linguagem AHDL do processador Risco, na versão de 16 bits. Os arquivos referem-se as diferentes partes do Risco, como o banco de registradores (breg_bk2.tdf), a unidade lógica e aritmética além da unidade de deslocamento (ula2.tdf), a parte de controle (i_risc.tdf) e a chamada de cada um destes módulos (unido_i.tdf).

# Anexo 4 Descrição VHDL do DLX-WCS

No disquete em anexo encontra-se o diretório dlxwcs, onde está a descrição em VHDL comportamental do DLX-WCS. A descrição, originária do laboratório Masi, foi modificada para suportar a WCS, sendo que as alterações estão devidamente comentadas no código. Além da WCS foi incluida a instrução de multiplicação, realizada em um ciclo, escrevendo o resultado em dois registradores pré determinados.

# Anexo 5 Descrição VHDL do RISCO-WCS

No disquete em anexo encontra-se o diretório riscowcs, onde está a descrição em VHDL comportamental do RISCO-WCS. A descrição foi realizada no CPGCC-UFRGS, sobre uma descrição do Risco sem a memória. Originalmente composta de 5 arquivos, a descrição foi colocada em um único arquivo, que deve ser desmembrado no caso de recompilação com o sistema Alliance. Para outros sistemas que suportem VHDL não deveriam existir problemas de compatibilidade, devendo ser escrita uma função de resolução chamada reg.

# Anexo 6 Resultados de análise e profiling para rotinas

No disquete em anexo encontra-se o diretório analise, com o resultado da análise estática das rotinas e a simulação para obtenção de profiling. As extensões .bin referem-se aos arquivos binários de saída, .doc incluem as linhas de assembler mais o binário, .smb fornece a tabela de símbolos, enquanto que .crt é o arquivo resultado da avaliação estática.

O arquivo div_int.ext apresenta uma rotina para computação da divisão inteira de dois números, enquanto que o arquivo dotp.ext contém uma rotina para produto de um vetor por uma cosntante. O último exemplo de análise estática é o gag.ext, que implementa a média aritmética-geométrica Gaussiana.

Para a análise dinâmica, foram simulados os arquivos div_int e dotp. Os resultados da divisão inteira podem ser vistos nos arquivos sbs_r.ext e sbs_w.ext. O primeiro é a execução passo a passo do algoritmo, enquanto o segundo é a mesma execução quando o algoritmo é colocado na WCS. Nos arquivos stat_r.ext e stat_w.ext encontram-se as estatísticas relativas a cada arquivo, concluindo-se com o número de operações de cada tipo executadas.

Para o produto escalar de um vetor, os arquivos dpsbs_r.ext e dpsbs_w.ext referem-se a execução passo-a-passo sem e com WCS, respectivamente. O resultado da análise dinâmica encontra-se nos arquivos dpstat_r.ext e dpstat_w.ext.

# Anexo 7 Algoritmo de Controle do Motor

No diretório motor do disquete em anexo encontra-se o fonte C do algoritmo de controle de motor. As funções de seno e cosseno foram realizadas com o algoritmo de Cordic, confome o arquivo cordic.c. No mesmo diretório encontra-se um conjunto de dados de corrente para teste do algoritmo.

O mesmo algoritmo foi usado tanto para teste do Risco-WCS e do 8051.

# Anexo 8 Algoritmo de Controle do Elevador em Risco

No disquete anexo encontra-se o diretório elevador, com os arquivos ele1.ext e ele_c.c. O primeiro é a descrição em linguagem Risco de um controlador de elevadores de 4 andares. O segundo é o fonte C do programa. Para poder ser executado no Risco 16 bits, o programa foi modificado para utilizar um mínimo de registradores. O programa binário encontra-se no arquivo ele1.bin.

## Anexo 9 Exemplo de execução de ECSD

No disquete em anexo encontra-se o diretório gag, com uma rotina que passou por uma análise e otimização dentro do ambiente ECSD. A rotina executa a média aritmética-geométrica gaussiana, sendo as operações de multiplicação e divisão feitas por SW. O arquivo gag.ext apresenta o fonte assembler em risco. O arquivo gag.doc a uma das saídas do montador, com o fonte anexado. Outra saída é o arquivo gag.bin, apenas com a programação da memória.

O resultado da análise estática encontra-se no arquivo gag.crt. Mesmo que a opção correta seja o WCS-pipe, foi exigida uma síntese virtual, que se encontra no arquivo gag.dag.

# Anexo 10 Fontes assembler dos programas analisados para 8051

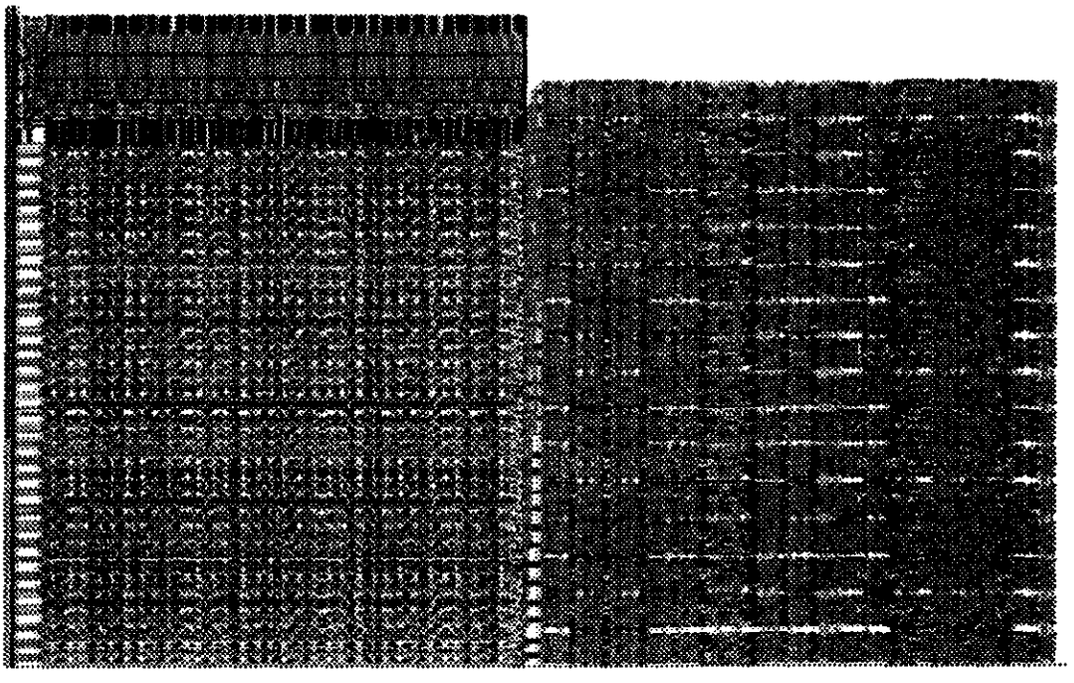
No disquete em anexo, no diretório 8051, encontram-se 3 programas em assembler para o 8051, que foram analisados com o conjunto de programas descrito no capítulo 5. Os programas foram desassemblados a partir de um código gerado por um compilador C comercial.

## Anexo 11 Fontes C dos programas de análise do código do 8051

No disquete em anexo, no diretório a8051, estão disponíveis os programas de análise utilizados para verificação das instruções mais usadas por compiladores comerciais do 8051.

## Anexo 12 Risco Full Custom

Abaixo encontra-se a figura do Risco full-custom, totalmente realizado com ferramentas desenvolvidas no CPGCC-UFRGS. À esquerda está o banco de registradores e seus decodificadores. A parte operativa está à direita, com a ULA e os registradores intermediários necessários. O shifter é de apenas uma posição.



# Anexo 13 Algoritmos em uso no trabalho

Neste trabalho foram realizadas diversas rotinas para implementação e validação de sistemas computacionais. Neste anexo serão apresentados os algoritmos utilizados durante o trabalho, de modo a se ter uma idéia mais precisa do tipo de trabalho realizado dentro do ambiente ECSD.

## 13.1 Algoritmos de análise estática

Estes algoritmos são utilizados durante a análise de rotinas, antes da classificação. O Montador Universal contém a implementação dos algoritmos mencionados.

Todo trabalho em ECSD está baseado na otimização de uma rotina crítica. Para tal, é necessário descobrir-se qual rotina é crítica em relação a alguma característica. Presentemente, ECSD trabalha apenas com uma rotina crítica, e portanto o montador realiza uma árvore de rotinas, de maneira a determinar uma rotina folha crítica. A figura A13.1 apresenta o algoritmo de seleção da rotina crítica em pseudo-código, implementado no Montador Universal.

```
Monta árvore de rotinas
Percorre a árvore, propagando constantes e determina:
                    rotina crítica chamada mais vezes em tempo conhecido
                    rotina crítica chamada mais vezes em tempo não conhecido
Para cada rotina crítica
    determina a característica da rotina
    calcula as otimizações
    emite relatórios
```

FIGURA A13.1 - Algorimo de determinação de rotina crítica

Como se pode notar, somente rotinas folhas serão candidatas a serem consideradas críticas. Apesar de ser uma limitação, nada impede ao usuário de re-escrever o código, trocando uma chamada de rotina por uma macro, por exemplo. A rotina de maior hieraquia seria naturalmente considerada crítica, por conter uma macro que implementava a rotina crítica anterior.

Para determinar a característica da rotina deve-se levar em conta quantas vezes um certo bloco básico da mesma é executado. Por exemplo, um laço com poucas instruções pode ser executado muitas vezes, dominando o tempo total de execução. Contudo, esta análise acurada só é possivel de ser feita de maneira estática se os limites de laços são conhecidos a priori, em tempo de compilação. O montador propaga as constantes disponíveis, e verifica quantas vezes cada bloco básico é chamado efetivamente. Com estes dados o custo da rotina pode ser determinado.

Caso não seja possível a propagação de constantes, é feita uma busca depth-first, para descoberta de todos os caminhos possíveis de uma rotina. O maior caminho é escolhido e analisado em termos de número de instruções computacionais, de memória, de salto e no número de blocos básicos. Assim, tem-se uma estimativa do custo da rotina em tempo de execução, mesmo sem se ter acesso ao conjunto real de dados necessários a execução da rotina. Estas operações estão resumidas na figura A13.2.

```
constrói flow-graph
se tempo de execução definido
então
        computa o tempo de execução total;
        computa o tempo individual de cada tipo de instrução;
        computa a aceleração;
        tipo_de_rotina = maior(aceleração);
senão
        acha o maior caminho a ser executado;
        computa o custo d omaior caminho apenas;
        computa a aceleração para o maior caminho;
        tipo_de_rotina = maior(aceleração);
```

FIGURA A13.2 - Algoritmo de classificação

## 13.2 Algoritmos de simulação

O algoritmo para simulação do sistema como um todo (microprocessador, memória, barramento, HW dedicado) parte do princípio que o barramento é o controlador do sistema. A partir dai a máquina de simulação fica atenta a modificações no barramento para avançar a simulação. O modelo de atrasos é transmission line, oriundo do simulador SHC original.

No simulador em código C do processador partiu-se da hipótese que o HW dedicado terá um tempo de execução fixo. Não são avaliados estados internos da função de HW, apenas se computa o resultado e se espera o número de ciclos adequados.

## 13.3 Algoritmos de movimentação de operações

Uma vez decidido que a rotina a ser otimizada é computacionalmente intensiva, o enfoque se desloca da rotina crítica para sua chamadora. Após a síntese virtual tem-se o número de ciclos necessário para execução, expondo o máximo paralelismo, sem preocupações com o custo, como explicado no item 13.6 abaixo.

O objetivo de se ter uma idéia do tempo máximo a ser utilizado advém do fato que o microprocessador deve saber quanto tempo esperar, para inserção ou não de nops. Para realmente colocar o HW dedicado trabalhando em paralelo com o microprocessador, eventualmente a rotina chamadora deve ser modificada para realizar operações no tempo morto de espera. O algoritmo de movimentação de operações encontra-se na figura 13.3.

```
determina Basic-Block que tem a chamada da rotina crítica;
para este Basic-Block
        executa loop unroll com um alista de operações
        enquanto houver operações na lista
                se é chamada de HW
                        move operações independentes para cima;
                        se mais de uma possível
                                escolhe operações das quais dependem a
                                próxima chamada de HW;
                elimina operação da lista;
```

FIGURA A13.3 - Algoritmo de movimentação de operações

## 13.4 Algoritmos de conversão WCS

Nesta situação a rotina é dominada por muitos saltos. A conversão de um programa Risco ou DLX em WCS é trivial, visto que basta descobrir o endereço de salto real, uma das etapas do montador. Se o endereço de salto e o endereço fonte estão a uma distância tal que estejam dentro da WCS, basta incluir o novo endereço relativo na memória. A sequência de operações encontra-se na figura 13.4 abaixo. O algoritmo foi também implementado no simulador, para mínimas modificações na máquina de simulação.

```
para a rotina crítica
coloca a primeira instrução no endereço zero da WCS;
para cada instrução
        monta uma tabela de símbolos com os endereços de salto, relativos ao
        endereço zero da WCS;
para cada instrução
        coloca instrução na WCS;
        se próxima instrução é de salto incondicional
                atualiza campo de próximo endereço da WCS com destino;
                remove instrução de salto da lista de próxima instrução;
        senão se próxima instrução é de salto condicional
                        atualiza campo de próximo endereço da WCS com destino;
                        atualiza campo de condição da WCS com condição da instrução;
                        remove instrução de salto da lista de próxima instrução;
        senão   atualiza campo de próximo endereço da WCS com atual + 1;
gera protocolo
```

FIGURA A13.4 - Algoritmo para obtenção da WCS

## 13.5 Algoritmos de conversão WCS-PIPE

Nesta situação a rotina é dominada por muitos acessos à memória e computações. A conversão de um programa Risco ou DLX em WCS é dependente do número de ciclos que as instruções de acesso à memória como load-store tomam. No caso do Risco, cada load ou store ocupa dois ciclos do processador, logo o agrupamento é sempre entre duas instruções computacionais.

```
para a rotina crítica
acha loop mais interno ou expande loops para aumentá-los;
se existe mais de um caminho
        inserir nops para torná-los de mesmo tamanho;
monta tabela de operações final;
monta grafo de dependências entre iterações;
se existe dependência real
então move operações para que o load fique perto do store, para diminuir dependência
para cada instrução da tabela
        se próxima é computacional
        então agrupa e insere ambas no pipe
        senão agrupa com nop e insere ambas no pipe
gera protocolo
```

FIGURA A13.5 - Algoritmo para obtenção da WCS-PIPE

## 13.6 Algoritmos de síntese virtual

Para a síntese virtual interessa somente o número de ciclos máximo necessário para a realização da função computacionalmente intensiva. Não são levados em consideração crítérios como o custo da Parte Operativa dai resultante. Uma otimização posterior sempre é possível, através da síntese real ou feita pelo usuário. Do ponto de

vista do sistema de CAD, o que interessa é o número de ciclos que o microprocessador deverá esperar.

Assim, um algoritmo para escalonamento de operações como o ASAP pode ser utilizado, visto que é rápido e ocupa pouca memória.

```
para cada bloco básico da rotina
        monta DAG
        escalona as operações As Soon As Possible
emite relatório sobre o total de ciclos de cada bloco básico
```

FIGURA 13.6 Algoritmo de previsão de ciclos para rotinas computacionais

## Bibliografia

[AHO 88] AHO, A.; SETHI, R.; ULLMAN, J. **Compilers, Principles, Techniques and Tools**. Reading, Massachusetts: Addison-Wesley. 1988.

[AIK 88] AIKEN, A.; NICOLAU, A. A Development Environment for Horizontal Microcode. **IEEE Transactions on Software Engineering**, New York, v.14, n.5, p.584-594, May 1988.

[ALB 95] ALBA, C.; CARRO, L.; SUZIM, A. Risco Based ASIP Generation. In: CONGRESS OF THE BRASILIAN MICROELECTRONICS SOCIETY AND IBERO AMERICAN MICROELECTRONICS CONFERENCE, 1., 1995, Canela, RS, Brasil. **Proceedings...** Porto Alegre: Instituto de Informática da UFRGS, 1995. p.311-317.

[ALL 92] ALLAN, V. et al. Enhanced Region Scheduling on a Program Dependence Graph. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 1992, Portland, Oregon. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 1992. p.72-80.

[ALT 93] ALTERA. **Programmable Devices Data Book**. San Jose: Altera Corporation, 1993.

[ALT 95] ALTERA. **Data Book**. San Jose: Altera Corporation, 1995.

[AMD 67] AMDAHL, G. M. Validity of the single-processor approach to achieving large scale computing capabilities. In: AFIPS CONFERENCE, 1997, Atlantic City. **Proceedings...** Montvale, New Jersey: AFIPS Press, 1967. p.483-485.

[ATH 91] ATHANAS, P.; SILVERMAN, H. An Adaptive Hardware Machine Architecture and Compiler for Dynamic Processor Reconfiguration. In: INTERNATIONAL CONFERENCE IN COMPUTER DESIGN, 1991, Cambridge, Massachusets. **Proceedings...** Los Alamitos, CA: IEEE Computer Society Press, 1991. p.397-400.

[ATH 93] ATHANAS, P.; SILVERMAN, H. Processor Reconfiguration Through Instruction-Set Metamorphosis. **IEEE Computer**, New York, v.26, n.3, p.11-18, Mar. 1993.

[BEC 92] BECKER, D.; SINGH, R.; TELL, S. An Engineering Environment for HW/SW co-simulation. In: ACM/IEEE DESIGN AUTOMATION CONFERENCE, 29., 1992, Anaheim, California. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 1992. p.129-134.

[BEN 95] BENNER, Th.et al. A Prototyping System for Verification and Evaluation in Hardware-Software Cosynthesys. In: INTERNATIONAL WORKSHOP ON RAPID SYSTEM PROTOTYPING, 1995, Chapel

Hill, North Carolina. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 1995. p. 54-59.

[BOU 91] BOUTE, R. T. Declarative Languages - still a long way to go. In: COMPUTER HARDWARE DESCRIPTION LANGUAGES, 1991, Marseille, France. **Proceedings...** Netherlands: Elsevier Science Publishers, 1991. p.165-192.

[BUR 95] BURGER, R. et al. **Implementação do processador Risco em Standard Cells e EPLDs**. Relatório de pesquisa a ser publicado.

[CAR 90] CARLSON, S. **Introduction to HDL-Based Design Using VHDL**. Mountain View, California: Synopsys , 1990.

[CAR 93] CARRO, L.; SUZIM, A., MARCON, C. SHC - SLX: A Levelized Compiled, Event Driven Interpreted VLSI simulator. **Microprocessor and Microprogramming**, Netherlands, v.38, 1993. Trabalho apresentado na conferência Euromicro, 1993, Barcelona. p.503-509.

[CAR 93a] CARRO, L. **Simulação multinível**. Exame de abrangência. Porto Alegre: CPGCC da UFRGS, 1993.

[CAR 94] CARRO, L. et al. Development of a RISC microprocessor family using a university developed tool set. In: EUROPEAN CONFERENCE ON DESIGN AUTOMATION, 1994, Paris, France. **User forum...** Los Alamitos, CA: IEEE Computer Society Press,1994. p.3-5.

[CAR 94a] CARRO, L.; SUZIM, A. Architectures and Algorithms for Computational Systems Development. In: INTERNATIONAL RAPID SYSTEM PROTOTYPING, 1994, Grenoble, France. **Proceedings...** Los Alamitos, CA: IEEE Computer Society Press, 1994. p.196-204.

[CAR 94b] CARRO, L.; SUZIM, A. Tuning HW-SW Codesign for parallel operation. In: INTERNATIONAL WORKSHOP ON HW-SW CODESIGN, Grenoble, France, 1994. **Poster session**.

[CHA 88] CHANG, P.; HWU, W.-M. Trace Selection for Compiling Large C Application Programs to Microcode. In: ACM/IEEE ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 1988, San Diego, CA. **Proceedings....** Washington: IEEE Computer Society Press, 1988. p. 21-29.

[CHA 92] CHARLES, H.-P. Loop Unrolling for processors with Instruction Cache. In: QUINTON, P.; ROBERT, Y. (Eds.). **Algorithms and Parallel VLSI Architectures II**. Netherlands: Elsevier Science Publishers, 1992. p. 311-316.

[CLA 90] SGS-THOMSON. SGS-THOMSON in breve. **CLASSE UNO**, Agrate Brianza, Itália, aprile/maggio 1990.

[COM 91] Computer Design, Design: clue to time to market. **Computer Design**, Oklahoma, v.30, n.25, Mar 91.

[DAS 85] DASARATHY, B. Timing Constraints of Real-Time Systems: constructs for expressing them, methods of validating them. **IEEE Transactions on SW Engineering**, New York, v.11, n.1, p.80-86, Jan. 1985.

[DeM 90] De MICHELE, G. et al. The Olympus Synthesis System for Digital Design. **IEEE Design and Test Magazine**, New York, v.7, n.10, p. 37-53, Oct. 1990.

[DIT 87] DITZEL, D. R.; McLELLAN, H. R. Branch Folding in the Crisp Microprocessor: reducing branch delay to zero. In: ANNUAL SYMPOSIUM IN COMPUTER ARCHITECTURE, 14., 1987, Pittsburgh, Pennsylvania. **Proceedings...** Washington: IEEE Computer Society Press, 1987. p.2-9

[ELE 91] ELECTRONIC DESIGN. Asic sales in 1990/95. **Electronic Design**, Cleveland, v.39, n.4, p.75, Feb. 1991.

[ERN 93] ERNST, R.; HENKEL, J.; BENNER, T. Hardware-Software Cosynthesis for Microcontrolores. **IEEE Design & Test of Computers**, Los Alamitos, v.10, n.2, p.64-75, Dec. 1993.

[FIS 81] FISHER, J. A. Trace Scheduling: A Technique for Global Microcode Compaction. **IEEE Transactions on Computer**, New York, v.30, n.7, p.478-490, July 1981.

[GAJ 92] GAJSKI, D. D. et al. Scheduling. In: GAJSKI, D. D. **High Level Synthesis**. Boston: Kluwer Academic Publishers, 1992. p.213-255

[GAJ 93] GAJSKI, D. D. Design process beyond ASICs. In: EUROASIC, 1993, Paris, France. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 1991. p.3-4.

[GRE 92] GREINER, A.; PÊCHEUX, F. Alliance: a complete set of CAD tools for teaching VLSI design. In: EUROCHIP WORKSHOP ON VLSI DESIGN TRAINING, 1992, Grenoble, France. **Proceedings...** [S.l.:s.n.], 1992. p.230-237.

[GUO 92] GUO, R. et al. A 1024 Pin Universal Interconnect Array with Routing Architecture. In: CUSTOM INTEGRATED CIRCUITS CONFERENCE, 1992, Austin, Texas. **Proceedings...** Los Alamitos: IEEE Computer Society Press,1992.

[GUP 92] GUPTA, R.; DE MICHELI, G. System Level Synthesis Using Re-programmable Components. In: DESIGN AUTOMATION CONFERENCE, 29., 1992, Anaheim, California. **Proceedings...** Los Alamitos, CA: IEEE Computer Society Press, 1992. p.2-7.

[GUP 92a] GUPTA, R.; COELHO, C.; DeMICHELE, G. Synthesis and simulation of digital systems containing interacting HW and SW componentes. In: DESIGN AUTOMATION CONFERENCE, 29., 1992, Anaheim, California. **Proceedings...** Los Alamitos, CA: IEEE Computer Society Press, 1992.

[HEN 84] HENNESSY, J. VLSI processor architecture. **IEEE Transactions on Computers**, New York, v. c-23, n. 12, p.1221-1236, Dec. 1984.

[HEN 90] HENNESSY, J.; PATTERSON, D. **Computer Architecture**: A quantitative approach. San Mateo, CA: Morgan Kaufmann, 1990. 648p.

[HUA 92] HUANG, S.-H. et al. A New Approach to Schedule Operations Across Nested-ifs and Nested-loops. In: ACM/IEEE ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 1992, Portand, Oregon. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 1992. p.268-271.

[IEE 93] IEEE Design & Test of Computers, Sept. 1993. Theme articles on Hardware-Software Codesign.

[ISM 94] ISMAIL, T. et al. An Approach for Hardware-Software Codesign. In: IEEE INTERNATIONAL RAPID SYSTEM PROTOTYPING, 1994, Grenoble, France. **Proceedings...** Los Alamitos, CA: IEEE Computer Society Press, 1994. p.73-80

[JER 91] JERRAYA, A.; PAULIN, P.; CURRY, S. Meta-VHDL for Higher Level Controller modeling and Synthesis. In: VLSI, 1991, Edinbourgh. **Proceedings...** [S.l.:s.n.], 1991.

[JUN 93] JUNQUEIRA, A.A. **Risco: Microprocessdor CMOS de 32 bits**. Porto Alegre: CPGCC da UFRGS, 1993. Dissertação de mestrado.

[KAR 85] KARPLUS, K.; NICOLAU, A. Efficient Hardware for Multi-way jumps and pre-fetches. In: ACM/IEEE ANNUAL WORKSHOP ON MICROPROGRAMMING, 18., 1985, Pacific Grove, CA. **Proceedings...** Washington: IEEE Computer Society Press,1985. p. 11-18.

[LAM 88] LAM, M. Software Pipelining: an Effective Scheduling Technique for VLIW Machines. **SIGPLAN notices**, New York, v.23, n.7, p.318-328, July 1988. Trabalho apresentado na Conference on Programming Languages Design & Implementation, SIGPLAN, 1988.

[LEU 95] LEUPERS, R.; MARWEDEL, P. A BDD-based Frontend for Retargetable Compilers. In: EUROPEAN DESIGN AND TEST CONFERENCE, 1995, Paris, France. **Proceedings...** Los Alamitos, CA: IEEE Computer Society Press, 1995. p.239-243.

[LIL 88] LILJA, D. J. Reducing the Branch Penalty in Pipelined Processors. **IEEE Computer**, New York, v.21, n.7, p. 47-55, July 1988.

[LIL 94] LILJA, D. J. Exploiting the Parallelism Available in Loops. **IEEE Computer**, New York, v.27, n.2, p. 13-26, Feb. 1994.

[LIM 94] LIMA, D. C. **Montador universal para família de processadores Risco**. Porto Alegre: CPGCC da UFRGS, 1994. 129p. Trabalho de Conclusão.

[LIN 91] LINDH, L.; STANISCHEWSKI, F. FASTCHART - Idea and Implementation. In: INTERNATIONAL CONFERENCE IN COMPUTER DESIGN, 1991, Cambridge, Massachusets. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 1991. p.401-404.

[LOR 93] LORENZI, C. et al. Instrumentação e Automação de um atuador Eletromecânico. In: SEMINÁRIO DE INSTRUMENTAÇÃO DO INSTITUTO DE PETRÓLEO, 1993, Porto Alegre. **Anais...** [S.l.:s.n.] , 1993. p. 344-355.

[LOR 93a] LORENZI, C. et al. Utilização de Barramento de Chão-de-fábrica. In: SEMINÁRIO DE INSTRUMENTAÇÃO DO INSTITUTO DE PETRÓLEO, 1993, Porto Alegre. **Anais...** [S.l.:s.n.] , 1993 p. 393-404.

[LOU 93] LOUCKS, W.; DORAY, B.; AGNEW, D. Experiences in Real-Time HW-SW co-simulation. In: VHDL INTERNATIONAL USER'S FORUM, 1993, Ottawa, Canada. **Proceedings...** [S.l]: VHDL International, 1993. p.47-57.

[MAL 88] MALAIYA, Y.; FENG, S. Design of a Testable RISC-TO-CISC Control Architecture. In: ACM/IEEE ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 1988, San Diego, CA. **Proceedings...** Washington: IEEE Computer Society Press, 1988. p. 57-59.

[MAR 90] MARKOWITZ, M. Technology update. **EDN**, Massachussets, v.35, n.11, p. 65-74, May 24, 1990

[MAR 92] MARCON, C. **Planejamento estrutural e simulação de Partes de Controle de Circuitos Integrados**. Porto Alegre: CPGCC da UFRGS, 1992. 308p. Dissertação de Mestrado.

[MAR 93] MARWEDEL, P.; SCHENK, W. Cooperation of Synthesis, Retargetable Code generation and Test Generation in the MSS. In: EUROPEAN DESIGN AUTOMATION CONFERENCE, 1993, Brussels, Belgium. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 1993. p.63-69.

[NIC 91] NICOLAU, A.; POTASMAN, R. Incremental Tree Height Reduction for High Level Synthesis. In: DESIGN AUTOMATION CONFERENCE, 1991,

San Francisco, CA. **Proceedings...** Los Alamitos, CA: IEEE Computer Society Press, 1991. p.770-74.

[OGA 70] OGATA, K. **Engenharia de Controle Moderno**. Rio de Janeiro: Prentice Hall, 1982. 929 p.

[ONI 95] ONION, F.; NICOLAU, A.; DUTT, N. Incorporating Compiler Feedback Into the Design of ASIPs. In: EUROPEAN DESIGN AND TEST CONFERENE, 1995, Paris, France. **Proceedings...** Los Alamitos, CA: IEEE Computer Society Press, 1995. p.508-513.

[PAP 93] PAPACHRISTOU, C. A. Vertical Migration of Software Functions and Algorithms Using Enhanced Microsequencing. **IEEE Transactions on Computers**, New York, v.42, n.1, Jan. 1993.

[PAR 91] PARK, C.; SHAW, A. Experiments with a Program Timing Tool Based on Source-Level Timing Schema. **IEEE Computer**, New York, v.24, n.5, p.48-57, May 1991.

[PAT 80] PATTERSON, D.; DITZEL, D. The case for the Reduced Instruction Set Computer. **Computer architecture news**, California, v.8, n.6, p.25-33, Oct. 1980.

[PIS 89] PISTORIO, P. *L'importanza strategica dei semiconduttori*. **Classe UNO**, Agrate Brianza, Itália, Dic. 1989.

[POL 88] POLYCHRONOPOULOS, C. Compiler optimizations for enhancing parallelism and their impact on architecture design. **IEEE Transactions on Computers**, New York, v.37, n.8, p.991-1004, Aug. 1988.

[REG 95] REGINATTO, R. **Controle por Campo Orientado do Motor de Indução com Adaptação de Parâmetros Via MRAC**. Florianópolis: Programa de Pós-graduação em Engenharia Elétrica, Universidade Federal de Santa Catarina, 1993. Dissertação de mestrado.

[RET 93] RETHMAN, N.; WILSEY, P. RAPID: A Tool for HW / SW Tradeoff analysis. In: VHDL INTERNATIONAL USER'S FORUM. 1993, Ottawa, Canada. **Proceedings...** [S.l.]: VHDL International, 1993. p.91-99.

[SAR 87] SARCCO, R.; TILANUS, P. CCITT SDL: Overview of the Language and its Applications. **Computer Networks and ISDN Systems**, Netherlands, v.13, n.2, p.65-74, 1987.

[SCH 93] SCHLOR, R.; DAMM, W. Specification and Verification of System-Level Hardware Designs using Timing Diagrams. In: EUROASIC, 1993, Paris, France. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 1993. p.518-524.

[SHA 93] SHARMA, A.; JAIN, R. Estimating Architectural Resources and performance for High Level Synthesis Applications. **IEEE transactions on Very Large Scale Integration (VLSI) Systems**, New York, v.1, n.2., p.175-190, June 1993.

[SMI 81] SMITH, J. E. A Study of Branch Prediction Strategies. In: THE ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 8., 1981, Chicago, Illinois. **Proceedings...** New York: ACM, 1981. p.135-148.

[SOU 94] SOUZA, A. Jr. et al. Multiplicador de 8 bits em FPGA. Congresso da Sociedade Brasileira de Microeletrônica, 9., 1994, Rio de Janeiro. **Anais...** Rio de janeiro: Universidade Federal do Rio de Janeiro, 1994, p.634-643.

[SOU 95] SOUZA, A. Jr.; CARRO, L.; SUZIM, A. An EPLD-Based microprocessor Family. In: ACM-IEEE EUROPEAN DESIGN AND TEST CONFERENCE, 1995, Paris, France. **User forum...** Los Alamitos, CA: IEEE Computer Society Press, 1995. p. 45-48.

[SRE 93] SREENIVASA, D.; KURDAHI, F. Hierarchical Design Space Exploration for a Class of Digital Systems. **IEEE Transactions on VLSI**, New York, v.1, n.3, p.282-295, Sept. 1993.

[STA 89] STALLMAN, R. M. Using and Porting GNU CC. Free Software Foundation, Inc. 1988, 1989. Texto agregado ao compilador GNU em distribuição magnética.

[SUN 91] SUN MICROSYSTEMS. **Mpsas User's Guide**. [S.l.]: Sun Microsystems, September 1991.

[SUN 91a] SUN MICROSYSTEMS. **Mpsas Module Programmer's Guide**. [S.l.]: Sun Microsystems, September 1991.

[SUZ 90] SUZIM, A. A. et al. RISCO: um processador RISC CMOS de 32 bits. In: SIMPÓSIO BRASILEIRO DE CONCEPÇÃO DE CIRCUITOS INTEGRADOS, 1990, Ouro Preto. **Anais...** Belo Horizonte: Universidade Federal de Minas Gerais, 1990.

[VAH 91] VAHID, F.; NARAYAN, S.; GAJSKY, D. D. SpecCharts: A Language for System Level Synthesis. In: COMPUTER HARDWARE DESCRIPRIPTION LANGUAGES, 1991, Marseille, France. **Proceedings...** Netherlands: Elsevier Science Publishers, 1991. p. 145-154.

[VAH 92] VAHID, F.; GAJSKY, D.D. Specification Partitioning for System Design. In: ACM-IEEE DESIGN AUTOMATION CONFERENCE, 29., 1992, Anaheim, California. **Proceedings...** Los Alamitos, CA: IEEE Computer Society Press, 1992. p.219-224.

[VOL 59] VOLDER, J. E. The CORDIC trigonometric computing technique. **IRE Trans. Electron. Computers**, [S.l.], v. EC-8, p.330-334, Sept. 1959.

[WIL 94] WILSON, T. et al. An Integrated approach to Retargetable Code Generation. In: INTERNATIONAL SYMPOSIUM ON HIGH LEVEL SYNTHESIS, 7., 1994, Ontario, Canada. **Proceedings...** Los Alamitos, CA: IEEE Computer Society Press, 1994. p. 71-75.

[WOL 93] WOLLAN, V. A Design Methodology Achieving Fast Development Cycles for Complex VLSI Architectures. In: EUROASIC, 1993, Paris, France. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 1993. p.532-535.

[WOO 94] WOO, N.; DUNLOP, A.; WOLF, W. Codesign from Cospecification. **IEEE Computer,** New York, v.7, n.1, p.42-47, Jan. 1994.

**Informática** UFRGS

**CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO**

*Algoritmos e Arquiteturas para o Desenvolvimento de Sistemas Computacionais*

por

Luigi Carro

Tese apresentada aos senhores:

Prof. Dr. Flávio Rech Wagner

Prof. Dr. Júlio Salek Aude (NCE/UFRJ)

Prof. Dr. Marcelo Soares Lubaszewski (DELET/UFRGS)

Examinador enviou parecer por escrito

Prof. Dr. Wilhelmus A. M. Van Noije (LSI/USP)

Vista e permitida a impressão.
Porto Alegre, 05 /07 /96 .

Prof. Dr. Altamiro Amadeu Suzim,
Orientador.

*Prof. Flávio Rech Wagner*
Coordenador do Curso de Pós Graduação
em Ciência da Computação CPGCC
Instituto de Informática - UFRGS