José C. dos Santos Alves

Processadores Dedicados

DEEC
FEUP
1997

Universidade do Porto
Faculdade de Engenharia

Processadores Dedicados
Concepção e Realização em Sistemas Digitais Reconfiguráveis

José Carlos dos Santos Alves

Dissertação apresentada na Faculdade de Engenharia da Universidade do Porto para obtenção do grau de Doutor em Engenharia Electrotécnica e de Computadores.

Porto
Outubro 1997

José Carlos dos Santos Alves

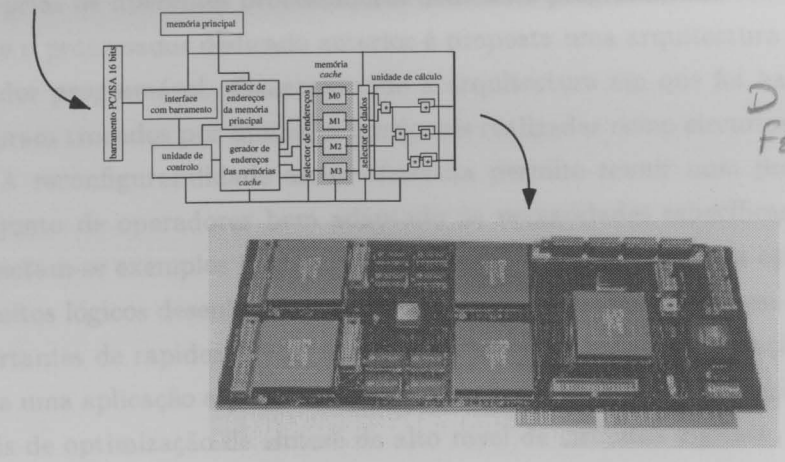
Processadores Dedicados

Concepção e Realização em Sistemas Digitais Reconfiguráveis

$\frac{1}{L}\Sigma X[i].X[i+\tau_1]$

$\frac{1}{L}\Sigma X[i].X[i+\tau_1].X[i+\tau_2]$

$\frac{1}{L}\Sigma X[i].X[i+\tau_1].X[i+\tau_2].X[i+\tau_3]$



Dissertação apresentada na Faculdade de Engenharia da Universidade do Porto para obtenção do grau de Doutor em Engenharia Electrotécnica e de Computadores.

Porto

Outubro 1997

Os trabalhos de Investigação apresentados nesta tese de Doutoramento foram apoiados pelo Programa PRODEP, medida 4.2, acção 12.19.

Resumo

O trabalho apresentado nesta dissertação incide sobre a concepção e realização de processadores dedicados em sistemas digitais reconfiguráveis, e aborda problemas fundamentais a partir de estudos desenvolvidos em torno de casos concretos. Os temas tratados abarcam o desenvolvimento de suporte para programação em alto nível, a selecção e construção de operadores dedicados, a optimização da configuração para uma arquitectura programável e reconfigurável e o desenvolvimento de raiz de um processador dedicado a uma aplicação específica, passando pelo estudo do problema, concepção da arquitectura e a sua implementação num novo sistema reconfigurável construído para essa aplicação.

A questão do desenvolvimento de suporte para programação em linguagens de alto nível é abordada no quadro de um processador dedicado programável por microcódigo, que foi construído para uma aplicação de controlo em tempo real. Foi adaptado um compilador de C para produzir instruções de baixo nível suportadas pelo processador estudado e foi desenvolvida uma aplicação para compactar o microcódigo executado pelo processador. Apesar de ter por objecto um processador dedicado a uma aplicação específica, a abordagem seguida é igualmente aplicável no desenvolvimento de ferramentas de suporte para programação em alto nível, adaptadas às características próprias de diferentes processadores dedicados programáveis.

Tendo por base o processador dedicado anterior é proposta uma arquitectura reconfigurável para um processador programável. Generalizando a arquitectura em que foi baseada, os seus operadores fixos foram trocados por unidades funcionais realizadas como circuitos programáveis do tipo FPGA. A reconfigurabilidade assim oferecida permite reunir num processador programável um conjunto de operadores bem adaptado às necessidades específicas de diferentes aplicações. Apresentam-se exemplos ilustrativos de situações onde o recurso a operadores construídos como circuitos lógicos desenhados expressamente para tarefas específicas pode conduzir a benefícios importantes de rapidez. Pelas características exibidas, a configuração da arquitectura proposta para uma aplicação específica requer uma abordagem de optimização conjunta dos problemas centrais de optimização de síntese de alto nível de circuitos digitais. Esse problema é tratado com uma aplicação que foi desenvolvida com base no algoritmo *Simulated Annealing*, permitindo considerar operadores com características muito genéricas, incluindo a capacidade de reconfiguração dinâmica suportada pelos FPGAs das gerações recentes.

Finalmente, é abordado o desenvolvimento de raiz de uma arquitectura optimizada para um problema concreto e a sua realização num sistema digital reconfigurável. Foi concebido um processador dedicado para a estimação de momentos de ordem superior que foi desenhado propositadamente para essa aplicação, explorando eficazmente as características próprias do problema estudado. Tendo por objectivo a realização, experimentação e afinação da arquitectura proposta, foi concebido e construído um novo sistema digital reconfigurável baseado em circuitos programáveis FPGA, no qual foi implementado e validado o processador dedicado desenvolvido.

Abstract

The work presented in this dissertation looks into the design and implementation of dedicated processors on reconfigurable digital systems, and addresses important problems that arise from developments around concrete case studies. High-level language programming support, selection and construction of dedicated operators, and optimized configuration of programmable reconfigurable processors are some of the problems that are addressed. The architectural design of an application-specific processor and its implementation on a special purpose reconfigurable system are also presented.

The issue of high-level programming support is tackled in the context of a microcoded dedicated processor that was developed for a real-time application. A general purpose C compiler was adapted in order to generate the low-level instructions executed by this processor, and a microcode compaction application was also built. Although these developments were targeted at a specific processor architecture the methods and techniques that are used are of interest in the general task of providing high-level language support for a wide range of application-specific programmable processors.

Inspired in this work, the architecture of a reconfigurable and programmable processor is presented next. In order to expand and add generality to the original architecture, the fixed set of operators is replaced with functional units realized on FPGA circuits. The reconfigurability thus offered allows building a processor whose set of operators may be adapted to specific tasks of different applications. Examples are presented illustrating situations where operators implemented as specially designed logic circuits can be used to speed-up the execution of certain tasks. The special characteristics identified in the configuration of this architecture require that the three central problems of high-level synthesis are adressed simultaneously in an integrated fashion. Based on this idea an application of Simulated Annealing was developed, allowing the consideration of operators with very generic characteristics that include the capability of dynamic reconfigurability found in modern FPGA circuits.

Finally, an application of digital signal processing involving the estimation of higher-order moments on long vectors of data, provided the basis of the specification of a dedicated processor that was designed from ground-up with the goal of exploiting the specific characteristics of the problem. It was implemented on a reconfigurable digital system that was built to offer a platform for the realization of the dedicated processor, and is also intended to serve as a target for the implementation of processors with similar general requirements. The results of this effort were found to be fully in line with the advantages of increased speed and greater flexibility that are to be expected from dedicated hardware implemented on a reconfigurable platform.

Résumé

Le travail présenté dans cette dissertation porte sur la conception et réalisation de processeurs dédiés sur systèmes digitaux reconfigurables, et aborde des problèmes fondamentaux à partir d'études développées autour de cas concrets. Les thèmes traités englobent le développement des outils de support pour la programmation de haut niveau, la sélection et construction d'opérateurs dédiés, l'optimisation de la configuration pour une architecture programmable et reconfigurable et tous les stades du développement d'un processeur dédié à une application spécifique, en passant par l'étude du problème, conception de l'architecture et son implémentation dans un nouveau système reconfigurable construit pour cette application.

La question portant sur le développement du support pour la programmation en langages de haut niveau est abordée dans le cadre d'un processeur dédié programmable par microcode, qui a été construit pour une application de contrôle en temps réel. Un compilateur de C a été adapté pour produire des instructions de bas niveau supportées par le processeur étudié et une application a été développée pour compacter le microcode exécuté par le processeur. Bien qu'elle ait pour objet un processeur taillé sur mesure pour une application spécifique, l'approche suivie est également applicable au développement d'outils de support pour la programmation de haut niveau, adaptés aux caractéristiques propres de différents processeurs dédiés programmables.

Ayant pour base le processeur antérieur est proposée une architecture reconfigurable pour un processeur programmable. Généralisant l'architecture sur laquelle elle est basée, ses opérateurs fixes ont été remplacés par des unités fonctionnelles réalisées comme des circuits programmables du type FPGA. La reconfiguration ainsi donnée permet de réunir dans un processeur programmable un ensemble d'opérateurs bien adapté aux nécessités spécifiques de différentes applications. Des exemples illustratifs de situations où le recours à des opérateurs construits comme des circuits logiques dessinés spécialement pour des tâches spécifiques, pouvant conduire à d'importants bénéfices de rapidité, sont présentés. Pour les caractéristiques affichées, la configuration de l'architecture proposée pour une application spécifique requiert un abordage d'optimisation conjointe des problèmes centraux d'optimisation de synthèse de haut niveau de circuits digitaux. Ce problème est traité avec une application développée sur la base de l'algorithme Simulated Annealing, permettant de considérer des opérateurs avec des caractéristiques très génériques, incluant la capacité de reconfiguration dynamique soutenue par les FPGAs des générations récentes.

Finalement, sont abordés tous les stades du développement d'une nouvelle architecture optimisée pour un problème concret et sa réalisation dans un système digital reconfigurable. Un processeur dédié a été conçu pour l'estimation de moments d'ordre supérieur qui a été dessiné précisément pour cette application, explorant efficacement les caractéristiques propres du problème étudié. Ayant pour objectif la réalisation, l'expérimentation et l'affinage de l'architecture proposée, un nouveau système digital reconfigurable basé sur des circuits programmables FPGA a été conçu, sur lequel a été implémenté et validé le processeur dédié développé.

Agradecimentos

Ao longo da realização deste trabalho e escrita desta dissertação, contei com o apoio de muitas pessoas e instituições a quem desejo exprimir publicamente os meus agradecimentos.

Ao meu orientador, Professor José Silva Matos pelo seu permanente apoio, motivação e orientação que foram decisivos na condução dos trabalhos realizados e apresentados nesta dissertação.

Aos meus colegas do grupo CAD e Microelectrónica do INESC, Ana Leão, António Araújo, Hélio Mendonça, João Canas Ferreira e José Alberto Machado pelo sempre bom ambiente de trabalho proporcionado e discussões proveitosas que contribuíram também em variados aspectos deste trabalho. Agradeço igualmente a todos os outros colegas da FEUP e do INESC que de uma forma ou de outra me apoiaram durante este trabalho.

Ao Professor Luís Corte-Real pela supervisão e coordenação do projecto HAREOS, e também ao colega André Puga pelas contribuições para o trabalho realizado no âmbito deste projecto.

A todos os colegas com quem trabalhei durante os 3 meses passados em Darmstadt, Alemanha. Em especial ao Professor Manfred Glesner e Michael Held que orientaram o trabalho aí realizado, mas também aos colegas Raj Singh, Rogério Oliveira e Zsolt Kohari pelo apoio e boa camaradagem que sempre proporcionaram durante essa estadia.

Desejo também agradecer publicamente ao Instituto de Engenharia de Sistemas e Computadores e à Faculdade de Engenharia da Universidade do Porto por todos os meios logísticos e materiais postos ao meu dispor durante a realização deste trabalho.

Finalmente, não posso terminar sem deixar de agradecer à minha família, aos meus amigos e em particular à minha esposa, pelo apoio que sempre mantiveram ao longo deste trabalho.

Índice

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Processadores dedicados - motivação

A evolução constante da tecnologia microelectrónica a que se tem assistido nas últimas décadas, desde a produção em série dos primeiros circuitos integrados (Texas Instruments, 1961) e do primeiro microprocessador comercial (Intel, 1972), tem permitido a construção de sistemas e equipamentos electrónicos cada vez mais rápidos, miniaturizados, fiáveis, consumindo menos energia e com custo mais baixo. Um exemplo bem conhecido de um tipo de equipamento que nos anos recentes tem beneficiado de forma clara desta evolução é o telefone celular, traduzindo-se num constante melhoramento dos principais atributos que são visíveis aos olhos do grande público: autonomia, dimensões físicas, peso e naturalmente custo.

As áreas de aplicação de sistemas que utilizam tecnologias microelectrónicas são muito variadas. O grande destaque no que se refere à diversidade de aplicações é concentrado nos sistemas baseados em circuitos digitais, e em particular em sistemas para controlo ou processamento de informação. Actualmente, controlo electrónico faz já parte de um número crescente de bens de consumo, ao ponto de cada vez menos tal ser usado como argumento para valorização de um novo produto no mercado. Como exemplo, refira-se que já nenhuma marca de automóveis utiliza como argumento de venda o facto de ser electrónico o controlo dos sistemas de ignição e injecção dos seus motores ou dos sistemas de anti-bloqueio de travagem (ABS—*Anti-Blocking System*).

Os computadores são naturalmente a grande área de aplicação que mais veio beneficiar da evolução das tecnologias microelectrónicas. Desde o lançamento do *Personal Computer* pela IBM, no início da década de 80, e como consequência da divulgação da sua arquitectura, o mercado mundial tem vindo a ser inundado pelos chamados *clones*, produzidos por fabricantes de computadores espalhados pelo mundo, com incidência maior

nos países do extremo oriente. As capacidades, dimensões e custo deste tipo de máquina tem vindo a ultrapassar todas as expectativas. A versatilidade permitida pela combinação de uma grande variedade de equipamentos, o baixo custo e a grande produção mundial de programas de aplicação para praticamente todas as áreas de trabalho, faz com que os chamados "PCs" ocupem actualmente o lugar cimeiro do mercado de computadores pessoais destinados ao grande público, estando definitivamente condenados a ocupar o lugar do computador doméstico por excelência.

Um dos principais atributos dos computadores é a flexibilidade permitida pela programação, o que possibilita a sua utilização num vasto conjunto de aplicações. Outro não menos importante é a sua rapidez, ditada em grande medida pelo desempenho do seu processador. O compromisso entre flexibilidade e rapidez, dentro de limites razoáveis de custo, é um dos factores determinantes na concepção da arquitectura de um novo processador e na definição do seu conjunto de instruções. Os fabricantes de processadores desenvolvem as suas arquitecturas favorecendo a flexibilidade da sua utilização em diferentes aplicações e por razões que se prendem com a baixa relação benefício/custo não incluem instruções que realizem tarefas específicas que só possam ser exploradas em áreas restritas de aplicação.

Em praticamente qualquer programa realizado por um computador existem partes que têm um impacto determinante em termos do seu tempo de execução. Uma afirmação corrente que ilustra essa constatação diz que 90% do tempo de execução é gasto por apenas 10% das instruções de um programa. Obviamente que estes números não são exactos, mas para quem desenvolve um programa de computador é normalmente fácil identificar conjuntos geralmente pequenos e bem localizados de tarefas que condicionam o desempenho global de um programa.

Esta constatação constitui a grande motivação para o desenvolvimento de processadores auxiliares dedicados, como forma de adicionar capacidades específicas a processadores de uso geral. Um exemplo comum são os processadores auxiliares aritméticos, que acrescentam operações sobre números representados em vírgula flutuante, ao conjunto de instruções suportados por um processador. Por razões de custo, essa funcionalidade não é incluída em alguns processadores, sendo introduzida como complemento à custa de um circuito integrado adicional. Isto permite que programas cujo desempenho não seja condicionado por operações daquele tipo possam ser executados por computadores equipados com processadores de custo mais baixo. Este é no entanto um tipo de operação usado pela generalidade dos programas de aplicação e não apresenta por isso o carácter de especificidade que justifique o recurso a circuitos integrados adicionais dedicados a essa tarefa. Na realidade, a tendência seguida actualmente pelos fabricantes de processadores

é incluírem já no mesmo circuito integrado unidades de cálculo em vírgula flutuante, o que é possível dado o elevado grau de integração que as tecnologias microelectrónicas actuais possibilitam.

Existem muitas outras situações em que o recurso a processadores auxiliares é vantajoso e por vezes necessário, para realizar partes bem definidas de um programa de forma mais rápida do que a conseguida por um processador de convencional. A necessidade de unidades de processamento auxiliar acontece também em aplicações com sistemas autónomos para controlo ou processamento de informação (normalmente referidos pela palavra inglesa *embbeded*), quando a mera função de acelerador é superada pelo requisito de cumprir determinadas tarefas em intervalos de tempo impostos pelo processamento em tempo real. Outras aplicações existem em que a necessidade de interfaces com características específicas requeridas por outros sistemas é a razão do recurso a processadores auxiliares dedicados, permitindo ultrapassar limitações relacionadas não só com a rapidez de tratamento de informação mas também com a velocidade de comunicação de dados.

Dado um problema complexo sob a forma de um programa ou de um algoritmo, a tarefa de seleccionar as partes que devem ser realizadas num processador dedicado faz parte de uma área de competência usualmente designada por projecto combinado *hardware/software* (*hardware/software-codesign*). Este termo refere os problemas e as metodologias associadas ao projecto de sistemas digitais, envolvendo por um lado, programas executados em processadores genéricos e por outro, circuitos dedicados a funções específicas da aplicação. Um problema que tem sido alvo de trabalhos de investigação nesta matéria consiste na divisão automática da funcionalidade pretendida para um sistema, entre um programa a executar num processador convencional e as tarefas a entregar a um processador auxiliar desenhado expressamente para essa aplicação. Apesar de várias abordagens propostas no sentido da automatização dessa tarefa, a dependência dos resultados obtidos relativamente à forma como o sistema é especificado e às características da arquitectura considerada para um processador auxiliar, dificultam o desenvolvimento e a utilização de metodologias sistemáticas para resolver esse problema.

No contexto da aplicação de processadores dedicados para acelerar programas executados por computadores convencionais, a tarefa de seleccionar a funcionalidade a realizar por um processador dedicado requer um bom conhecimento do problema, mesmo antes da sua codificação numa linguagem de programação. Na realidade, a facilidade ou dificuldade dessa escolha, partindo de uma codificação de um programa numa linguagem de programação de alto nível, depende em grande medida da forma e do estilo como essa codificação é feita. Requerendo competências que ultrapassam a programação de computadores, decidir sobre a selecção das partes críticas de uma aplicação e avaliar o benefício

real da sua realização num processador auxiliar criado especificamente para essas tarefas pode exigir uma equipa interdisciplinar que reuna as várias áreas de conhecimento requeridas nesta actividade.

1.2 Sistemas digitais reconfiguráveis

Um processador dedicado consegue melhores desempenhos do que um processador de uso geral realizado em tecnologia semelhante, à custa da exploração de paralelismos exibidos pelas operações da aplicação e da natureza das próprias operações. Com efeito, existem situações para as quais podem ser construídos operadores eficientes com circuitos lógicos pouco complexos, realizando funções que em processadores convencionais obrigam à execução de sequências de várias das suas instruções elementares e que por isso conduzem a tempos substancialmente mais longos.

O projecto de um processador para uma aplicação específica requer uma análise cuidada do problema e da avaliação dos benefícios resultantes da sua utilização. Dado o carácter específico das funções normalmente atribuídas a um processador dedicado, a melhor solução no que se refere a rapidez de funcionamento é, na generalidade das situações, o desenvolvimento de uma arquitectura inteiramente dedicada e optimizada para a aplicação em causa e a sua realização física como um circuito integrado de aplicação específica (ASIC—*Application Specific Integrated Circuit*[1]).

Devido ao tempo e custos envolvidos na fabricação de um novo circuito integrado, nem sempre é praticável recorrer ao desenvolvimento e fabricação de um ASIC quando se pretende construir um processador dedicado para uma aplicação específica. Existem naturalmente excepções em que o grande volume de produção ou o custo elevado do produto final justificam plenamente os custos e riscos associados à fabricação de circuitos integrados inteiramente dedicados a uma aplicação. Um exemplo são os processadores dedicados a acelerar certos tipos de operações no processamento de gráficos computacionais, incluídos em cartas gráficas de computadores pessoais.

Com a introdução pela XILINX em 1985 dos primeiros circuitos lógicos programáveis para usos genéricos (FPGA—*Field-Programmable Gate Array*), personalizados pelo utilizador final por escrita em células de memória estática (SRAM—*Static Random Access Memory*), teve início a era dos sistemas computacionais reconfiguráveis. Este tipo de dispositivo veio oferecer um meio para a realização física de circuitos lógicos, oferecendo a vantagem de não requerer qualquer dispositivo especial para a sua programação e de

[1] No contexto deste trabalho, o termo ASIC é utilizado na sua interpretação restrita de circuito integrado produzido em fábrica para uma aplicação específica.

poder ser re-usado um número arbitrário de vezes. Embora não podendo competir em termos de densidade de integração e desempenho com ASICs fabricados em tecnologia semelhante, a flexibilidade permitida pela sua reconfigurabilidade permite acomodar diferentes circuitos lógicos, sem obrigar a qualquer modificação física do meio em que é inserido.

Este tipo de dispositivo conduziu já a diversas realizações de sistemas digitais reconfiguráveis baseados nesta tecnologia, destinados a complementar computadores ou processadores convencionais. As áreas de aplicação são variadas e vão desde sistemas orientados para a emulação de circuitos lógicos genéricos até unidades de processamento dedicadas que, em aplicações seleccionadas, chegam a superar em várias ordens de magnitude o desempenho de super-computadores.

São claros os benefícios de recorrer a plataformas de realização física reconfiguráveis como meio de implementação de sistemas lógicos dedicados a aplicações específicas. Para além de poderem competir em termos de custo com soluções baseadas em circuitos integrados fabricados expressamente para uma aplicação, o factor risco normalmente associado à implementação de um novo sistema como um ASIC e o tempo longo de fabricação são praticamente inexistentes quando se recorre à implementação em plataformas reconfiguráveis. Além disso, a flexibilidade oferecida pela modificação rápida da estrutura de um circuito lógico torna possível rentabilizar um sistema deste tipo na construção de sistemas lógicos com arquitecturas optimizadas para servir de forma eficaz as necessidades específicas de variadas aplicações.

A introdução de circuitos lógicos programáveis, oferecendo a capacidade de reconfiguração rápida e parcial dos seus recursos, permitiu uma abordagem nova na implementação de sistemas digitais: reconfiguração dinâmica de *hardware*. Este novo conceito refere-se à possibilidade de modificar a funcionalidade de um sistema lógico durante a sua operação, de modo a adaptá-la de forma dinâmica às necessidades específicas em diferentes estágios do seu funcionamento. A exploração deste conceito pode ser feita a diferentes níveis de complexidade das tarefas que partilham ao longo do tempo a mesma plataforma física. Embora existam há já alguns anos FPGAs que oferecem a possibilidade de reconfiguração parcial (por exemplo ATMEL 6000), a nova família de FPGAs introduzida pela XILINX em 1996 (XC6200), suportando reconfiguração rápida e parcial e oferecendo capacidades que actualmente ultrapassam as $100K$ portas lógicas, veio abrir caminhos promissores à exploração desta nova metodologia na implementação de processadores dedicados para aplicações específicas.

1.3 Síntese de processadores dedicados

Com a escolha da tarefa de uma aplicação a realizar num processador dedicado construído sobre uma plataforma reconfigurável, coloca-se o problema de obter uma arquitectura para um circuito lógico que possa ser realizada de forma eficiente nos recursos existentes na plataforma reconfigurável eleita.

A tradução de uma descrição algorítmica de um problema numa arquitectura de um circuito lógico que o realize, representa uma tarefa normalmente conhecida como síntese de alto nível de circuitos digitais. Partindo de uma representação da função a realizar, o objectivo é atingir a descrição de um circuito lógico ao nível da transferência entre registos (RTL—*Register Transfer Level*) envolvendo registos, memórias, unidades funcionais (somadores ou multiplicadores, por exemplo) e a descrição de uma máquina de estados capaz de produzir a sequência de sinais de controlo necessária para ser realizada a operação pretendida. A implementação dessa estrutura num suporte de realização física envolve outros problemas que são normalmente vistos como passos posteriores à definição da arquitectura e, em certa medida, separados do problema de síntese de alto nível: síntese lógica e implementação física. Síntese lógica refere-se, em termos gerais, ao processo de optimizar um circuito lógico tendo em vista a sua construção numa tecnologia dada; implementação física consiste em especificar a forma como é efectuada a realização desse circuito lógico na tecnologia alvo escolhida.

Síntese de alto nível parte de uma descrição algorítmica de um problema, normalmente representado numa linguagem de descrição de *hardware* (HDL—*Hardware Description Language*). Para além da interpretação e optimização da representação algorítmica, onde são empregues procedimentos semelhantes aos usados por compiladores de linguagens de programação de alto nível, a obtenção de uma arquitectura optimizada para uma aplicação envolve três tarefas fundamentais: sequenciamento das operações atribuindo-as a ciclos determinados do relógio do sistema, selecção de um conjunto de elementos funcionais adequado às operações do problema e atribuição dessas operações aos elementos funcionais seleccionados. Embora em situações particulares estes três problemas possam ser tratados de forma individual, são na realidade intimamente interdependentes e na generalidade das situações requerem uma abordagem conjunta que permita a sua optimização simultânea.

A grande actividade em torno de síntese de alto nível teve início no final da década de 80. As técnicas desenvolvidas para os diversos problemas que lhe são associados foram criadas considerando principalmente como tecnologia alvo circuitos integrados de aplicação específica. Mais do que um processo automático para substituir um projectista na construção de uma arquitectura dedicada, esta metodologia é hoje vista como um meio

de obter de forma rápida variadas alternativas que possibilitem a um projectista comparar e decidir entre diversos compromissos arquitecturais.

O recurso a sistemas de síntese de alto nível como meio de traduzir uma descrição algorítmica de um problema numa arquitectura de um circuito lógico com vista à sua implementação num sistema reconfigurável deve ser encarado à luz de diferentes considerações e requisitos. De facto, a concepção de uma arquitectura para um circuito lógico tendo por alvo a sua realização num sistema reconfigurável já construído, não pode ser conduzida seguindo a mesma abordagem que é tomada quando se considera uma realização física como um ASIC. Para além da natureza diferente dos elementos construtivos disponíveis, um sistema reconfigurável contém geralmente vários FPGAs com estruturas de interligação variadas, incluindo também outros componentes destinados a certos tipos de funções específicas (por exemplo memórias ou circuitos de interligações programáveis).

Em sistemas digitais reconfiguráveis, o número e tipo dos FPGAs e a arquitectura de interligação entre eles é muito variável de sistema para sistema, e resultam normalmente das características das primeiras aplicações que motivam a sua realização. Juntando a isso os condicionamentos resultantes dos limites de capacidade e desempenho dos circuitos FPGA, resùlta que a concepção de uma arquitectura para um processador dedicado, destinada à realização num sistema reconfigurável já construído, seja intimamente dependente das características próprias dessa plataforma de realização física. Por essa razão, na generalidade das situações são pressupostas estruturas padrão para os sistemas a implementar em plataformas desse tipo, o que condiciona uma metodologia para o desenho de circuitos lógicos destinados a esse tipo de plataforma. Naturalmente, a automatização do processo de construção de arquitecturas destinadas à implementação em sistemas digitais reconfiguráveis obriga ao recurso a ferramentas de apoio que fazem uso de técnicas de síntese de alto nível. No entanto, dadas as características estruturais próprias exibidas por diferentes plataformas alvo baseadas nesta tecnologia, as ferramentas desenvolvidas são também específicas para o sistema a que se destinam.

1.4 Trabalho realizado

O trabalho apresentado nesta dissertação aborda problemas fundamentais relacionados com a concepção, utilização e implementação de processadores dedicados em sistemas digitais reconfiguráveis, tendo por base desenvolvimentos realizados em torno de estudos de aplicações concretas, que envolvem processadores dedicados exibindo variados graus de flexibilidade. Para cada um deles apresentam-se soluções desenvolvidas para problemas que se colocam na sua concepção e utilização, e que vão desde a construção de ferramentas

de suporte para programação de processadores dedicados com linguagens de alto nível até à concepção e realização de raiz de processadores dedicados para áreas concretas de aplicação bem como de sistemas reconfiguráveis que lhes sirvam de suporte.

O primeiro trabalho abordado tem por base um processador dedicado que foi desenvolvido no Instituto de Sistemas Microelectrónicos da Universidade Técnica de Darmstadt, Alemanha, (THD—*Technische Hochschule Darmstadt*) para uma aplicação de controlo num sistema electrónico-mecânico. O autor fez parte da equipa de investigação que desenvolveu esse sistema, durante um estágio que realizou naquele instituto. Trata-se de um circuito programável por micro-instruções que foi fabricado como um ASIC para satisfazer os requisitos de tempo real da aplicação a que se destina. Para possibilitar a programação dessa arquitectura com uma linguagem de alto nível, foi adaptado um compilador de C e foi desenvolvida uma aplicação para sequenciar e compactar o microcódigo para este processador. Apesar de se tratar de uma arquitectura particular que foi criada para servir as necessidades de uma aplicação concreta, os problemas ilustrados e os resultados e conclusões obtidos aplicam-se de igual modo em situações similares, em que seja desejável a construção de ferramentas para suportar a programação de arquitecturas programáveis específicas com linguagens de programação de alto nível.

A rigidez apresentada pelo processador dedicado explorado no trabalho anterior condiciona a gama de aplicações em que pode ser usado e sugeriu a concepção de uma arquitectura para um processador reconfigurável baseado em operadores construídos sobre circuitos FPGA. Para além da flexibilidade resultante da capacidade de executar programas, é acrescentada a reconfigurabilidade dos operadores implementados em circuitos lógicos programáveis, o que permite reunir, para diferentes aplicações, um conjunto de operadores optimizados para as suas necessidades. Para além das operações normalmente disponíveis nos processadores convencionais, o uso de circuitos lógicos programáveis torna possível a construção de operadores dedicados a tarefas seleccionadas, para as quais sejam conseguidas realizações eficientes com circuitos lógicos desenhados expressamente para essas tarefas.

Com a introdução da reconfigurabilidade dos operadores, o processo de tradução de uma especificação de um problema para a arquitectura proposta transcende o problema da compilação de um problema especificado numa linguagem de alto nível. Com efeito, a adaptação dessa arquitectura para uma aplicação específica, seleccionando os operadores mais adequados às suas necessidades, envolve a solução dos três problemas principais de síntese de alto nível de circuitos digitais: selecção de operadores, sequenciamento das operações, e atribuição das operações aos operadores escolhidos. Para resolver estes problemas com uma abordagem de optimização conjunta foi desenvolvida uma aplicação

baseada num método de optimização combinatória (*Simulated Annealing*), que considera operadores com características muito genéricas, que não são contempladas pelas técnicas conhecidas aplicadas em síntese de alto nível de circuitos digitais. A aplicação desenvolvida permite optimizar a configuração da arquitectura proposta para satisfazer as necessidades específicas de diferentes aplicações.

A implementação daquela técnica de optimização foi generalizada de forma a incluir restrições que permitem modelar a capacidade de reconfiguração dinâmica dos circuitos lógicos onde são implementados os operadores. Desta forma, o conjunto de operadores dedicados às necessidades específicas de uma aplicação pode ser mais vasto do que os recursos físicos permitem ter simultaneamente, modificando dinamicamente as funções realizadas pelos circuitos programáveis, de acordo com as necessidades em diferentes estágios da resolução de um problema. A consideração dos tempos necessários às reconfigurações dos circuitos lógicos onde são implementados os operadores introduz restrições adicionais que não são tidas em conta pelas técnicas de optimização usadas em síntese de alto nível, mas são contempladas pela aplicação desenvolvida.

O estudo de um problema concreto de processamento digital de sinal, conduziu à construção de raiz de um processador dedicado e de um sistema reconfigurável que lhe serviu de suporte de implementação. Este trabalho decorre no âmbito de um projecto de investigação iniciado em Dezembro de 1995 e financiado pela Junta Nacional de Investigação Científica e Tecnológica (projecto HAREOS, *Hardware* Reconfigurável para Estatísticas de Ordem Superior). O processador desenvolvido (ProHos) destina-se à estimação de momentos de ordem superior, o que é realizado por um conjunto de cálculos computacionalmente pesados e que representam a parte crítica da estimação de estatísticas de ordem superior. Este processador dedicado foi desenvolvido para avaliar a viabilidade de utilização em tempo real daquela ferramenta de análise, aplicada a técnicas de codificação de vídeo digital para baixas cadências. Ao contrário das duas arquitecturas abordadas nos dois trabalhos anteriores, este sistema foi desenhado como um circuito inteiramente dedicado ao problema a resolver, tendo sido colocada em *hardware* toda a sua funcionalidade. Deste modo, foi possível explorar de forma eficaz as características de paralelismo exibidas pelo problema estudado, resultando numa arquitectura que realiza processamento vectorial e que foi optimizada expressamente para a aplicação estudada.

Com vista à implementação e avaliação desse processador dedicado, foi concebido e construído um novo sistema digital reconfigurável, que se destina à realização de processadores auxiliares dedicados para computadores pessoais do tipo PC. A arquitectura deste sistema reconfigurável foi influenciada pelo requisitos colocados pelo ProHos, mas mostra-se igualmente adequada à implementação de processadores dedicados para ou-

tros problemas que apresentem características de processamento vectorial semelhantes às identificadas no problema estudado. Para além de arquitecturas destinadas a diferentes problemas na área da estimação de estatísticas de ordem superior, outras aplicações estão planeadas para esta plataforma reconfigurável.

1.5 Organização da tese

A tese está organizada em 7 capítulos cujos conteúdos se resumem em seguida. No capítulo 1 é feita uma breve introdução aos assuntos abordados, e são apresentadas as contribuições relevantes do trabalho realizado. No capítulo 2 é realizada uma visita pelos principais aspectos relacionados com sistemas digitais reconfiguráveis, e são apresentados alguns exemplos representativos das diferentes áreas de aplicação.

O capítulo 3 apresenta o estudo e desenvolvimento que foi realizado em torno de um processador dedicado, criado para uma aplicação concreta de controlo de um sistema electrónico-mecânico. É apresentado o enquadramento da aplicação do processador estudado e descreve-se na secção 3.4 o trabalho realizado com vista à construção de um compilador de uma linguagem de alto nível para essa arquitectura. Este trabalho inclui duas partes principais que são a adaptação de um compilador configurável para a arquitectura estudada e a implementação de uma técnica de sequenciamento com vista à geração e compactação do microcódigo para o processador dedicado.

No capítulo 4 propõe-se uma arquitectura para um processador reconfigurável, que foi inspirada na do processador dedicado estudado no capítulo anterior. É apresentada na secção 4.3 a arquitectura proposta e são discutidos detalhes associados à sua realização física. Na secção 4.4 são apresentadas e comparadas implementações para operadores específicos seleccionados, que pela sua natureza mostram ser adequados à realização como circuitos lógicos dedicados nos recursos reconfiguráveis do processador proposto.

O problema da configuração da arquitectura proposta é abordado no capítulo 5. Ultrapassando o problema da tradução e simplificação de uma especificação de alto nível de uma aplicação, já tratado no capítulo 3, é aqui abordado o problema da adaptação dos recursos reconfiguráveis disponíveis na arquitectura, de forma a optimizá-los para as necessidades específicas dessa aplicação. Este problema identifica-se com as tarefas nucleares em síntese de alto nível de circuitos digitais, embora as características de reconfigurabilidade que se consideram acrescentem dificuldades que não são contempladas pelas técnicas conhecidas para resolver esses problemas. São revistos trabalhos de outros autores que propõem aplicações de técnicas de programação matemática para optimização daquele problema, mas que não suportam as características da arquitectura alvo considerada. É proposta e

apresentada na secção 5.7 uma aplicação baseada em *Simulated Annealing* para a optimização do problema estudado, que satisfaz as características genéricas de funcionamento consideradas para a arquitectura proposta. A generalização dessa implementação é apresentada na secção 5.7.8, permitindo contemplar a reconfiguração dinâmica dos recursos reconfiguráveis da arquitectura proposta.

No capítulo 6 é apresentado o estudo detalhado de uma aplicação concreta na área de processamento digital de sinal para a qual foi desenvolvido um processador dedicado e um sistema digital reconfigurável destinado à sua realização física. Após o enquadramento do problema estudado, apresenta-se o desenvolvimento que conduziu a este processador dedicado e descreve-se na secção 6.4 o desenvolvimento e realização de um novo sistema digital reconfigurável que lhe serviu de suporte. Esta plataforma reconfigurável foi concebida tendo por base a arquitectura do processador dedicado desenvolvido, e destina-se à implementação de processadores auxiliares dedicados para computadores pessoais.

Finalmente, no capítulo 7 são tecidas conclusões finais e apresentam-se algumas perspectivas para futuros desenvolvimentos, na linha do trabalho que se apresenta nesta dissertação.

Na escrita de um documento em língua Portuguesa sobre os assuntos abordados nesta dissertação, um problema corrente que se coloca é a opção por utilizar ou não certos termos em língua Inglesa, que são consagrados pela comunidade científica mundial e para os quais não existem palavras na nossa língua que exprimam correctamente a mesma ideia.

Alguns termos em língua Inglesa entraram já na nossa linguagem corrente e não existem traduções adequadas que sejam bem aceites, por não exprimirem correctamente os conceitos e ideias que lhes estão associados. Um exemplo são *hardware* e *software* que podem ser traduzidos, em determinados contextos, para "plataforma de realização física" e "programas de aplicação", mas que noutras situações são empregues com um significado mais lato do que o sugerido por aquelas definições. Por este motivo, optou-se por escrever em *itálico* algumas palavras em Inglês para as quais não se encontrou tradução que exprimisse de modo adequado as ideias que lhes são associadas.

Outra opção tomada na escrita desta dissertação diz respeito à utilização de acrónimos, que na generalidade dos casos resultam da composição de palavras em língua Inglesa e que são normalmente aceites para representar conceitos que fazem já parte da nossa linguagem corrente, tal como RAM, FPGA ou LED. Qualquer tentativa de tradução para Português resultaria não natural e seria uma fonte de distracção durante a leitura deste texto. Por

este motivo, é mantida a forma original para a generalidade dos acrónimos usados, sendo apresentado o seu significado a primeira vez que são referidos no contexto de um capítulo. No início desta dissertação apresenta-se uma lista dos acrónimos referidos ao longo deste trabalho que são relevantes na área em que este trabalho se insere.

Capítulo 2

Sistemas digitais reconfiguráveis

2.1 Introdução

A introdução pela XILINX em 1985 dos primeiros agregados programáveis de portas lógicas, genericamente designados por FPGA (*Field-Programmable Gate Array*) e programáveis por escrita em células de memória RAM estática (SRAM), constituiu um avanço importante nas tecnologias de implementação de sistemas electrónicos digitais. Esse evento deu origem ao nascimento de um novo meio de implementação de sistemas digitais, permitindo a re-utilização de uma mesma plataforma física para realizar circuitos lógicos diferentes, abrindo as portas para o desenvolvimento de sistemas digitais reconfiguráveis.

Se for definindo "sistema digital reconfigurável" como qualquer sistema lógico que possa ver alterada a sua funcionalidade sem obrigar à construção ou modificação da plataforma de realização física, então podem também incluir-se nesta definição aqueles sistemas que são construídos em torno de microprocessadores ou microcontroladores, já existentes antes da introdução dos circuitos FPGA. Na verdade, a flexibilidade oferecida pela sua programação permite modificar a função que um sistema desse tipo realiza, mas à custa da construção de sequências de operações padrão que são bem definidas e determinadas pelo circuito lógico fixo que implementa o processador. Esta definição tão abrangente contempla também qualquer circuito lógico que possa realizar diferentes funções à custa, por exemplo, da especificação do estado de linhas de controlo, tal como acontece numa unidade lógica e aritmética. Também circuitos integrados de aplicação específica (ASICs) podem ser construídos oferecendo flexibilidade semelhante para realizarem diferentes funções, embora normalmente limitados aos domínios de aplicação restritos para que o circuito é concebido. Um exemplo disso é o processador aritmético reconfigurável apresentado em [RKJ93], capaz de realizar 8 operações aritméticas complexas, à custa da

configuração das funções realizadas pelos operadores e das interligações entre eles. Contudo, em qualquer destes casos a estrutura do circuito lógico é fixa e está "gravada" de forma permanente na pastilha de silício que constitui o circuito integrado.

Considerar ou não os exemplos acima referidos como sistemas digitais reconfiguráveis depende de serem vistos apenas ao nível da função que realizam ou ao nível da estrutura do circuito lógico que os constitui. Do ponto de vista macroscópico, e tomando apenas em consideração a função que realizam, aquele tipo de sistema oferece um meio de realização que permite a sua adaptação para diferentes aplicações. No entanto, descendo ao nível do seu circuito lógico é encontrada uma estrutura rígida que condiciona o grau de flexibilidade oferecido.

No sentido de clarificar os termos empregues para identificar diferentes tipos de sistemas lógicos que exibam capacidade de adaptação a diferentes aplicações, considere-se a definição lata de sistemas flexíveis para incluir qualquer sistema que exiba, por qualquer processo, capacidade de ser adaptado para diferentes aplicações; sistemas reconfiguráveis aqueles que possibilitam a modificação da estrutura do circuito lógico que realizam; sistemas programáveis aqueles onde, como os processadores convencionais, é conseguida flexibilidade à custa da realização de sequências de operações padrão determinadas pela estrutura (fixa) do circuito lógico que os realiza.

O aparecimento dos circuitos FPGA tornou possível alargar o âmbito da flexibilidade à própria estrutura do circuito lógico, abrindo as portas para o aparecimento de sistemas digitais reconfiguráveis. Este tipo de dispositivo veio oferecer um meio de implementar circuitos lógicos para aplicações específicas, no local onde a aplicação é desenvolvida e sem recorrer aos serviços normalmente caros e demorados das fundições de silício. A sua personalização estabelece a funcionalidade e as interligações de circuitos que realizam funções lógicas elementares, de forma a constituir um circuito complexo, que pode actualmente incluir num único circuito integrado o equivalente a várias dezenas de milhar de portas lógicas.

Neste capítulo é feita uma breve introdução a dispositivos programáveis do tipo FPGA, como o elemento construtivo fundamental de sistemas digitais reconfiguráveis, e apresentam-se alguns sistemas reconfiguráveis que contemplam várias das áreas de aplicação desta tecnologia. Não se pretende nesta síntese enumerar todos os sistemas que existem actualmente, porque esse número é grande e cresce de dia para dia. Apresentam-se aqueles que pelas suas características e pelo carácter pioneiro da sua apresentação representam marcos importantes nos vários domínios de aplicação que esta nova tecnologia tem vindo a conquistar.

2.2 FPGAs—Agregados programáveis de portas lógicas

Agregados programáveis de portas lógicas (FPGA—*Field-Programmable Gate Array*) são circuitos integrados personalizados pelo utilizador final. Um circuito FPGA é constituído por uma estrutura regular de blocos lógicos configuráveis, interligados entre si por ligações que são estabelecidas através de interruptores configuráveis. A personalização de um dispositivo deste tipo é feita electricamente pelo utilizador final, definindo as funções a realizar pelos blocos lógicos configuráveis e o modo como são criadas as interligações entre eles. As ferramentas de suporte que os fabricantes disponibilizam para os seus circuitos, traduzem uma especificação de um circuito lógico representado por listas de ligações, na informação adequada para programar o circuito. Na figura 2.1 mostra-se a estrutura genérica de um circuito FPGA, evidenciando os seus elementos constituintes fundamentais: blocos lógicos configuráveis, blocos de entrada/saída, recursos de interligação e interruptores programáveis.

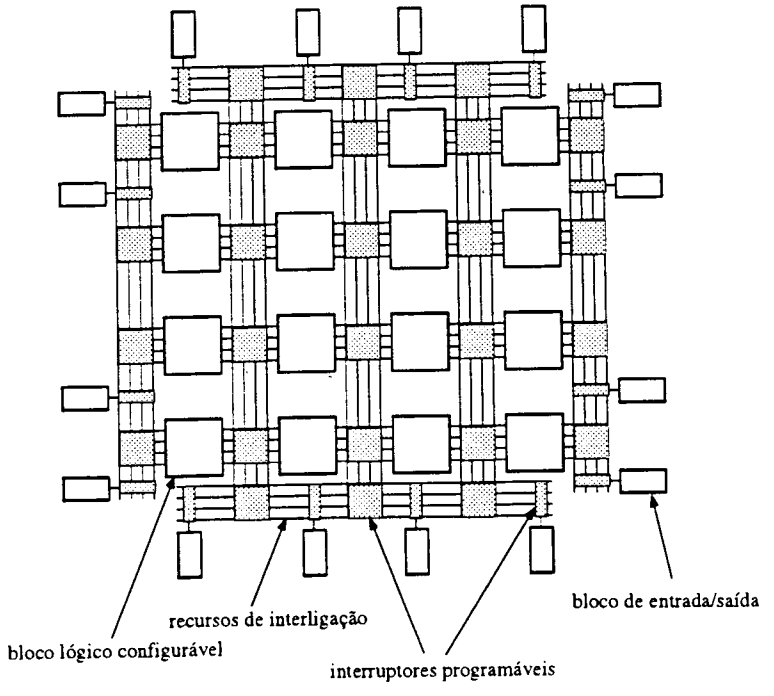


Figura 2.1: Arquitectura de um circuito FPGA hipotético.

Na figura 2.2 exemplifica-se a forma como esse FPGA hipotético poderia ser usado para construir um circuito somador de 2 números de dois *bits*, considerando que cada bloco lógico pode implementar portas lógicas E, OU ou OU-EXCLUSIVO de duas ou 3

entradas. Neste exemplo não são usados 2 dos 16 blocos lógicos configuráveis, 6 são usados apenas para estabelecer interligações e 8 implementam as funções lógicas necessárias ao circuito.

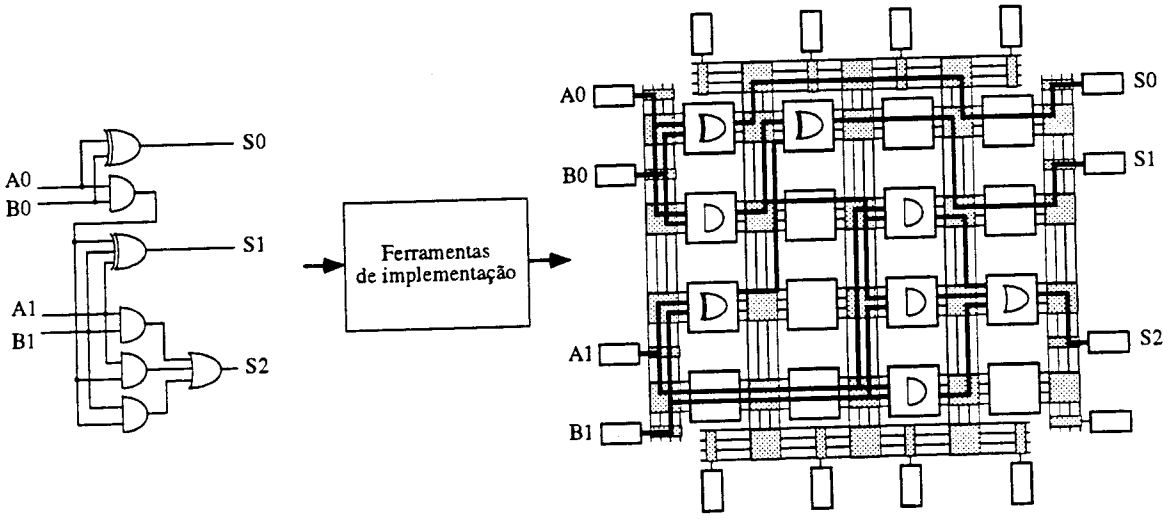


Figura 2.2: Implementação de um somador de números de 2 *bits* no FPGA da figura anterior.

As principais vantagens deste tipo de circuito programável, quando comparado com os dispositivos programáveis tradicionais geralmente designados por PAL (*Programmable Array Logic*), são oferecer maior capacidade e flexibilidade para a construção de circuitos com lógica multi-nível. Actualmente são vulgares dispositivos que permitem implementar circuitos lógicos com complexidades equivalentes a 10 mil portas lógicas e várias dezenas de entradas e saídas. O fabricante XILINX que lidera o mercado mundial deste tipo de dispositivo, colocou já no mercado circuitos com capacidades que ultrapassam 100 mil portas lógicas.

Existem actualmente vários tipos de circuitos FPGA oferecidos por diversos fabricantes, que se diferenciam principalmente em duas grandes características: tecnologia usada para a sua programação, e arquitectura dos blocos lógicos configuráveis e dos recursos de interligação [RGSV93].

2.2.1 Tecnologias de programação

A programação de um FPGA define a funcionalidade pretendida para o circuito, e é feita especificando o estado de interruptores que estabelecem as funções a realizar pelos blocos lógicos configuráveis e a forma como são criadas as interligações entre eles. O modo como esses interruptores são construídos traduz-se em variadas características eléctricas e

dinâmicas, para além de diferentes áreas de silício requeridas para cada tipo. Uma grande divisão é feita entre circuitos com programação definitiva (anti-fusíveis), programação não volátil (EPROM e EEPROM) e programação volátil (RAM estática).

Como o nome indica, programação definitiva só pode ser realizada uma única vez num dispositivo. A tecnologia empregue actualmente consiste em programar de forma permanente o estado de anti-fusíveis [GHB93]. Ao contrário dos fusíveis normais onde a fusão de um elemento condutor interrompe um circuito eléctrico, anti-fusíveis são dispositivos normalmente desligados, onde a ruptura do elemento dieléctrico permite estabelecer a passagem de corrente eléctrica. A baixa resistência eléctrica desses interruptores no estado ligado, a pequena capacidade eléctrica no estado desligado e a reduzida área de silício necessária à sua implementação representam as principais vantagens desta tecnologia de programação. No entanto, para além dos dispositivos só poderem ser usados uma única vez, é necessário equipamento específico para a sua programação.

Circuitos FPGA com programação não volátil têm interruptores construídos com tecnologia semelhante à usada nas memórias EPROM ou EEPROM (*Erasable Read-Only Memory* e *Electrically Erasable Read-Only Memory*). Em ambos os casos a programação do circuito é não volátil, mas pode ser modificada de forma semelhante à usada na programação de memórias daqueles tipos. A tecnologia EPROM obriga à remoção do circuito do sistema em que se integra para ser apagado com radiação ultra-violeta. A configuração dos FPGAs com tecnologia EEPROM é feita recorrendo a sinais eléctricos, não sendo por isso necessário remover o circuito do sistema em que é usado. No entanto é necessário incluir circuitos com características eléctricas dedicadas para a sua configuração. Quando comparados com os anti-fusíveis, as células de EPROM ou EEPROM requerem mais espaço e apresentam valores de resistência e capacidade eléctricas tipicamente 10 vezes mais elevados.

Nos dispositivos FPGA com programação volátil os interruptores que definem a funcionalidade do circuito são controlados por *bits* gravados em células de memória estática (SRAM). Uma consequência imediata desta forma de programação é a necessidade de reprogramar o circuito sempre que é estabelecida a alimentação eléctrica, o que obriga a ter outro dispositivo ou sistema onde a configuração seja mantida de forma permanente. No entanto, como a programação é feita escrevendo em células de memória RAM, pode ser realizada através de sinais lógicos com características eléctricas e dinâmicas semelhantes às existentes num sistema digital de uma família lógica compatível. Por essa razão não são necessários equipamentos dedicados à sua programação, podendo essa operação ser realizada através de outros circuitos lógicos existentes no mesmo sistema (memórias, microprocessadores ou sistemas construídos com outros circuitos programáveis). Para além

disso, a configuração é feita em tempos curtos que geralmente não ultrapassam poucos milissegundos, e não há limite para o número de vezes que o circuito pode ser reconfigurado. Como a reconfiguração deste tipo de dispositivo é realizada pela escrita de informação numa memória RAM, é comum dizer que este tipo de circuito é programável por *software*.

Apesar da área de silício necessária para construir as células de SRAM ser superior à usada por células de memória não volátil, a rapidez e facilidade de reconfiguração constituem o principal motivo da crescente aceitação que este tipo de dispositivo tem vindo a conquistar nos últimos anos, com especial aplicação em sistemas que tiram partido da reconfiguração rápida de *hardware*.

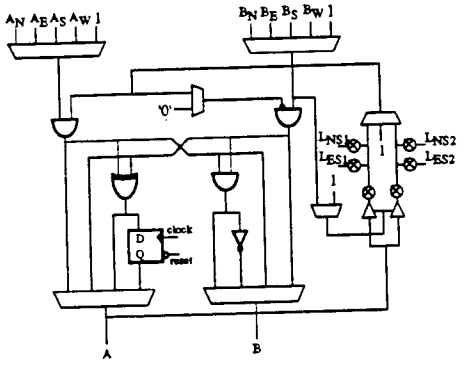
2.2.2 Arquitecturas

Para além da forma como é armazenada a configuração de um dispositivo FPGA, outra característica importante que distingue as propostas dos vários fabricantes é a sua arquitectura, que engloba a estrutura dos blocos lógicos configuráveis e das interligações que podem ser estabelecidas entre eles.

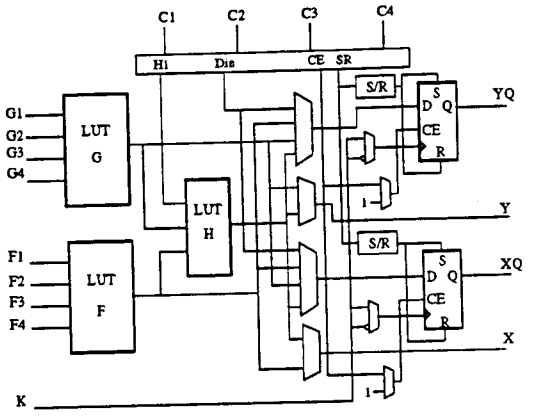
A grande divisão no que se refere à estrutura dos blocos configuráveis é feita em termos da sua granularidade—complexidade da função lógica que pode ser implementada em cada um [OD95]. Uma classificação possível identifica três níveis diferentes de complexidade. Blocos lógicos de pequena granularidade podem ser considerados os oferecidos pela Crosspoint (um par de transistores PMOS e NMOS), Plessey (uma porta NÃO-E de duas entradas) e ATMEL (três portas E, um inversor e uma porta NÃO-OU-EXCLUSIVO). Blocos lógicos de média complexidade são oferecidos pela Actel e Quicklogic (circuitos selectores[1]) e XILINX (blocos de memória estática usados como tabelas de consulta). Os FPGAs da Altera da família FLEX 8000 utilizam blocos configuráveis de grande complexidade, constituídos por agregados de outros blocos mais simples, interligados entre si através de ligações locais. Para possibilitar a implementação de circuitos sequenciais, todos os dispositivos oferecem um ou mais *flip-flops* em cada bloco configurável. Na figura 2.3 mostram-se três tipos de blocos lógicos configuráveis com diferentes granularidades: *Logic Cell* da ATMEL 6000 [ATM95], *Configurable Logic Block* da XILINX 4000 [XIL95] e *Logic Array Block* da ALTERA FLEX 8000 [ALT96].

Blocos lógicos de pequena complexidade permitem utilizar de forma eficiente a funcionalidade oferecida por cada um. No entanto, o uso de blocos configuráveis que implementem funções pouco complexas obriga a grandes densidades de interligações entre eles, o que resulta geralmente em baixo desempenho devido aos atrasos introduzidos pelas in-

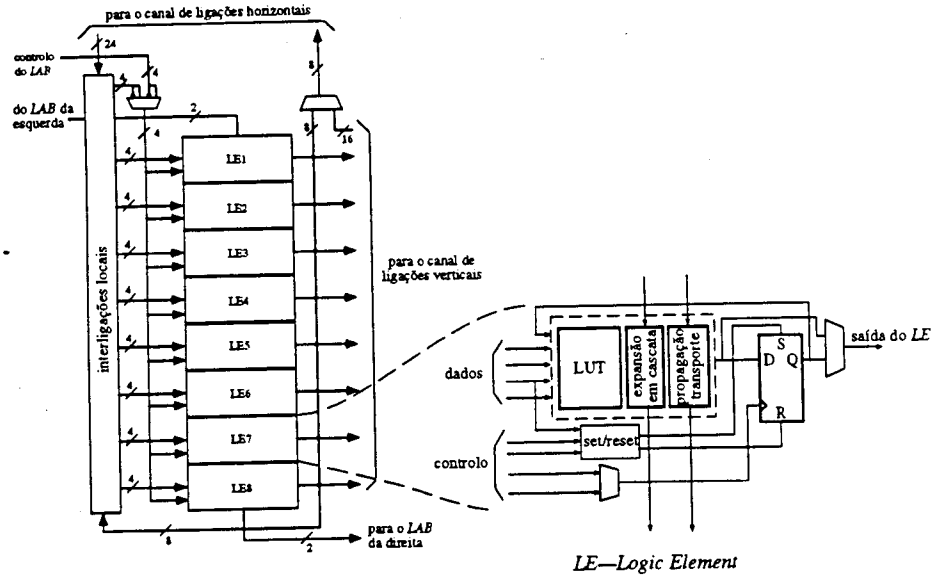
[1] *multiplexers*



ATMEL AT6000
Logic Cell



XILINX XC4000
CLB—Configurable Logic Block



ALTERA FLEX 8000
LAB—Logic Array Block

LE—Logic Element

Figura 2.3: Exemplos de blocos lógicos configuráveis de pequena (ATMEL), média (XILINX) e grande complexidade (ALTERA).

terligações programáveis. A utilização de blocos lógicos de grande granularidade permite acomodar em cada um funções mais complexas e utilizar de forma mais eficiente a área de silício do FPGA, requerendo também menor densidade de interligações. Em [RGSV93] é apresentado um resumo de trabalhos que estudam o efeito da granularidade dos blocos lógicos configuráveis na eficácia de utilização da área de implementação oferecida por um FPGA e nos atrasos que daí resultam para vários exemplos de circuitos de teste.

A arquitectura de interligações programáveis é também característica de cada família

de FPGA, e determinante na capacidade de realização de todas as ligações necessárias para a implementação de um circuito lógico. De uma forma geral a todas as arquitecturas, esses recursos são constituídos por segmentos de condutores, que podem ser ligados entre si de forma a constituírem pistas interligando terminais dos blocos lógicos configuráveis. Todas as arquitecturas dispõem também de ligações dedicadas à distribuição de sinais de relógio para os *flip-flops*, garantindo as diferenças de atraso mínimas que são necessárias para a construção de circuitos síncronos.

No que refere à forma como são construídos os recursos de interligação, existem também compromissos que são factores determinantes nas decisões tomadas pelos fabricantes. Idealmente, um FPGA deve dispor de um número suficiente de ligações programáveis que permitam realizar qualquer circuito lógico que consiga ser atribuído aos seus blocos lógicos configuráveis. No entanto, o número de segmentos de ligação e de interruptores programáveis é determinante na área total do circuito e reflecte-se naturalmente no seu custo. No mesmo trabalho referido antes [RGSV93], é apresentado um resumo de estudos do efeito do número de recursos de interligação programáveis na capacidade de estabelecer todas as ligações necessárias a um circuito, bem como na eficiência de utilização da área de silício.

2.3 Sistemas reconfiguráveis—exemplos e aplicações

Como foi definido atrás, um sistema digital reconfigurável é uma plataforma de realização física que permite implementar diferentes circuitos lógicos sem requerer a construção ou modificação dessa plataforma. Dada a rapidez e facilidade de reconfiguração, a generalidade dos sistemas digitais reconfiguráveis são construídos em torno de circuitos FPGA com programação por escrita em células de SRAM.

Uma grande divisão de sistemas deste tipo distingue entre sistemas reconfiguráveis estáticamente e sistemas reconfiguráveis dinamicamente. Os primeiros utilizam a característica de reconfigurabilidade do *hardware* para se adaptarem a diferentes aplicações, mantendo fixa essa configuração durante a sua execução. Sistemas reconfiguráveis dinamicamente exploram a reconfiguração rápida e em certos casos parcial, permitida pelos circuitos FPGA de alguns fabricantes, para utilizar repetidas vezes o mesmo *hardware* físico na realização de tarefas não simultâneas de uma aplicação.

A figura 2.4 ilustra o conceito de reconfiguração estática. O problema a resolver é traduzido num circuito lógico que é implementado nos dispositivos programáveis que compõem a plataforma reconfigurável (4 no exemplo da figura). Usando reconfiguração dinâmica (figura 2.5), o problema é dividido em tarefas que possam ser realizadas de forma

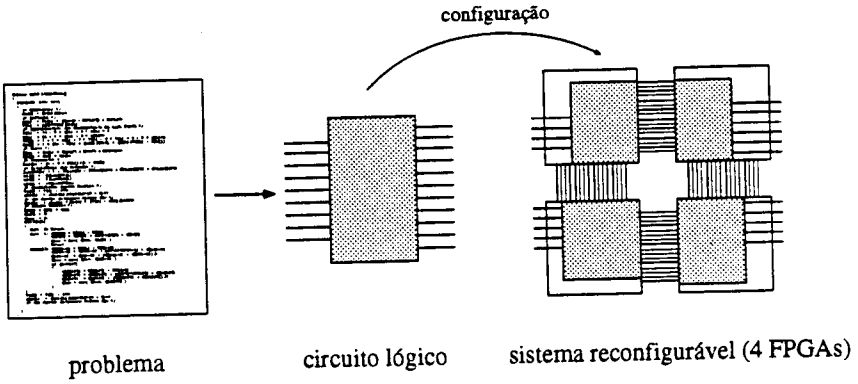


Figura 2.4: Reconfiguração estática.

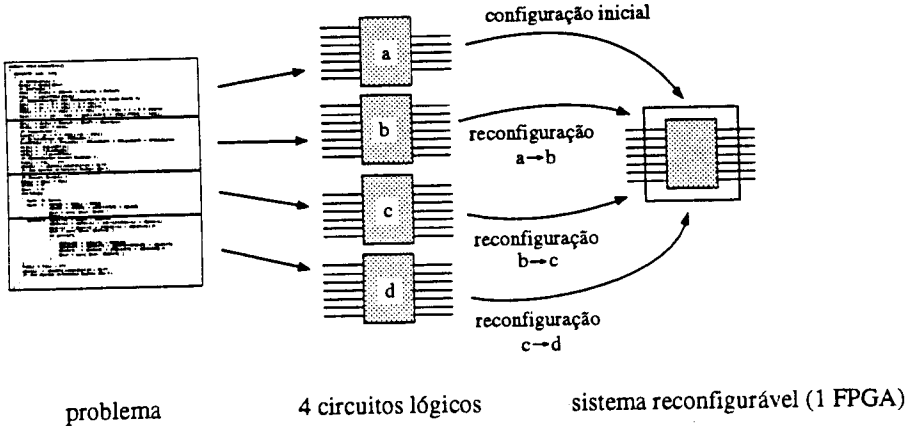


Figura 2.5: Reconfiguração dinâmica.

não simultânea, podendo por isso partilhar no tempo o mesmo espaço físico oferecido pela plataforma reconfigurável (1 FPGA no exemplo apresentado). As tarefas em que o problema é dividido são traduzidas em circuitos lógicos, sendo realizadas em sequência na mesma plataforma física, reconfigurando-a sucessivamente para implementar os diferentes circuitos em que o problema foi traduzido. Embora seja necessário menos *hardware* físico do que o necessário com reconfiguração estática, o tempo de resolução do problema é acrescido com o tempo gasto nas trocas de configurações dos circuitos lógicos.

Sistemas reconfiguráveis dinamicamente começaram a produzir resultados promissores com a introdução de circuitos lógicos FPGA com possibilidade de reconfiguração rápida e parcial, tal como é o caso dos FPGAs ATMEL 6000 e da recente família introduzida pela XILINX (XC6200). No entanto, a grande parte dos sistemas digitais reconfiguráveis desenvolvidos correntemente explora apenas reconfiguração estática para criar sistemas digitais dedicados a aplicações específicas.

de FPGA, e determinante na capacidade de realização de todas as ligações necessárias para a implementação de um circuito lógico. De uma forma geral a todas as arquitecturas, esses recursos são constituídos por segmentos de condutores, que podem ser ligados entre si de forma a constituírem pistas interligando terminais dos blocos lógicos configuráveis. Todas as arquitecturas dispõem também de ligações dedicadas à distribuição de sinais de relógio para os *flip-flops*, garantindo as diferenças de atraso mínimas que são necessárias para a construção de circuitos síncronos.

No que refere à forma como são construídos os recursos de interligação, existem também compromissos que são factores determinantes nas decisões tomadas pelos fabricantes. Idealmente, um FPGA deve dispor de um número suficiente de ligações programáveis que permitam realizar qualquer circuito lógico que consiga ser atribuído aos seus blocos lógicos configuráveis. No entanto, o número de segmentos de ligação e de interruptores programáveis é determinante na área total do circuito e reflecte-se naturalmente no seu custo. No mesmo trabalho referido antes [RGSV93], é apresentado um resumo de estudos do efeito do número de recursos de interligação programáveis na capacidade de estabelecer todas as ligações necessárias a um circuito, bem como na eficiência de utilização da área de silício.

2.3 Sistemas reconfiguráveis—exemplos e aplicações

Como foi definido atrás, um sistema digital reconfigurável é uma plataforma de realização física que permite implementar diferentes circuitos lógicos sem requerer a construção ou modificação dessa plataforma. Dada a rapidez e facilidade de reconfiguração, a generalidade dos sistemas digitais reconfiguráveis são construídos em torno de circuitos FPGA com programação por escrita em células de SRAM.

Uma grande divisão de sistemas deste tipo distingue entre sistemas reconfiguráveis estáticamente e sistemas reconfiguráveis dinamicamente. Os primeiros utilizam a característica de reconfigurabilidade do *hardware* para se adaptarem a diferentes aplicações, mantendo fixa essa configuração durante a sua execução. Sistemas reconfiguráveis dinamicamente exploram a reconfiguração rápida e em certos casos parcial, permitida pelos circuitos FPGA de alguns fabricantes, para utilizar repetidas vezes o mesmo *hardware* físico na realização de tarefas não simultâneas de uma aplicação.

A figura 2.4 ilustra o conceito de reconfiguração estática. O problema a resolver é traduzido num circuito lógico que é implementado nos dispositivos programáveis que compõem a plataforma reconfigurável (4 no exemplo da figura). Usando reconfiguração dinâmica (figura 2.5), o problema é dividido em tarefas que possam ser realizadas de forma

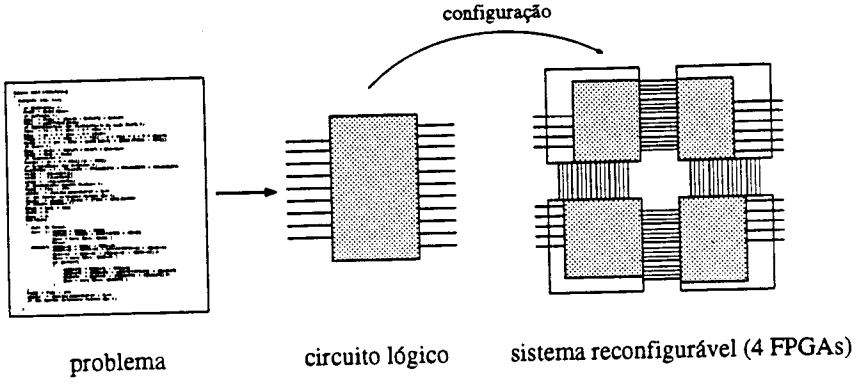


Figura 2.4: Reconfiguração estática.

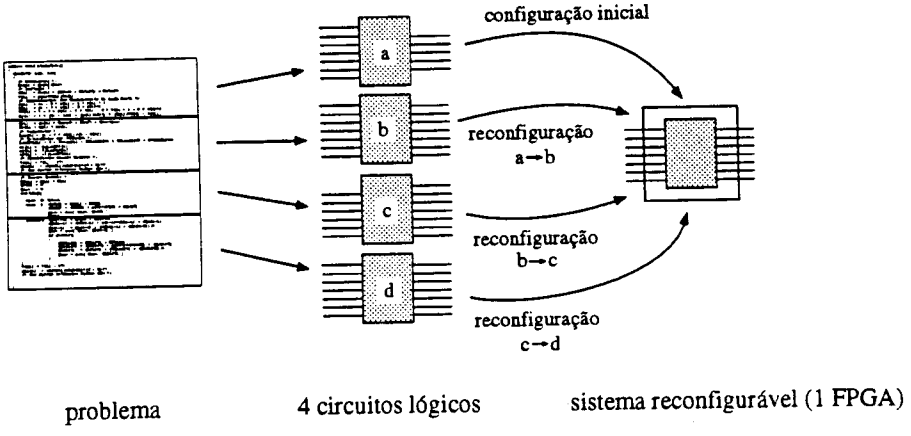


Figura 2.5: Reconfiguração dinâmica.

não simultânea, podendo por isso partilhar no tempo o mesmo espaço físico oferecido pela plataforma reconfigurável (1 FPGA no exemplo apresentado). As tarefas em que o problema é dividido são traduzidas em circuitos lógicos, sendo realizadas em sequência na mesma plataforma física, reconfigurando-a sucessivamente para implementar os diferentes circuitos em que o problema foi traduzido. Embora seja necessário menos *hardware* físico do que o necessário com reconfiguração estática, o tempo de resolução do problema é acrescido com o tempo gasto nas trocas de configurações dos circuitos lógicos.

Sistemas reconfiguráveis dinamicamente começaram a produzir resultados promissores com a introdução de circuitos lógicos FPGA com possibilidade de reconfiguração rápida e parcial, tal como é o caso dos FPGAs ATMEL 6000 e da recente família introduzida pela XILINX (XC6200). No entanto, a grande parte dos sistemas digitais reconfiguráveis desenvolvidos correntemente explora apenas reconfiguração estática para criar sistemas digitais dedicados a aplicações específicas.

Os exemplos que se apresentam nas secções seguintes representam marcos importantes na história recente dos sistemas digitais reconfiguráveis. De entre as várias áreas de aplicação que tiram partido desta tecnologia, são de especial interesse duas categorias de aplicações: sistemas dedicados para emulação ou prototipagem de *hardware* e sistemas reconfiguráveis orientados para implementar processadores para aplicações específicas. Para além dos desenvolvimentos apresentados que se inserem naquelas categorias, são também revistos outros trabalhos em áreas que começam actualmente a produzir resultados promissores: integração de processadores convencionais com circuitos FPGA e aplicações de sistemas reconfiguráveis dinamicamente. Outra área de aplicação interessante, no âmbito da qual é apresentado um trabalho realizado pelo autor, consiste na exploração de sistemas digitais reconfiguráveis para construir interfaces dedicadas a equipamentos de instrumentação.

Em `http://www.io.com/~guccione /HW_list.html` pode ser obtida uma lista que resume os desenvolvimentos recentes de sistemas digitais reconfiguráveis.

2.3.1 Emulação e prototipagem de *hardware*

Emulação ou prototipagem de *hardware* tem em vista a aceleração de uma classe de problemas muito específica: simulação de sistemas digitais. Tradicionalmente, a simulação de um sistema digital é feita recorrendo a programas de computador que permitem simular o seu comportamento, em reposta a sequências de estímulos definidas pelo projectista. Com a crescente complexidade dos circuitos permitida pelo avanço das tecnologias microelectrónicas, e apesar da evolução paralela da capacidade de processamento dos computadores, é cada vez mais difícil simular de forma rigorosa e em tempo útil o comportamento de sistemas electrónicos complexos com milhares ou milhões de transistores ou portas lógicas.

O conceito de emulação permite verificar o funcionamento de um sistema digital, "executando" um seu modelo físico (protótipo). Antes da vulgarização dos circuitos reconfiguráveis, a construção de protótipos era feita à custa de circuitos integrados com funções padrão e técnicas de montagem caras, demoradas e muito sujeitas a erros humanos. Para além de possibilitar a verificação em tempo real de um sistema digital usando, em muitos casos, estímulos obtidos do sistema em que vai ser utilizado, permite também a simulação conjunta de sistemas que envolvam componentes de *hardware* e de *software*.

Emuladores de ASICs

Uma aplicação de sistemas reconfiguráveis destina-se a emular o funcionamento de um ASIC, de forma a permitir a sua simulação eficaz no ambiente para que foi projectado, antes da sua fabricação como circuito integrado. Emuladores de circuitos integrados para usos gerais podem ser vistos como grandes circuitos FPGA, embora, por limitações de capacidade dos FPGAs sejam compostos por vários circuitos físicos interligados entre si. As ferramentas de suporte permitem dividir a funcionalidade do circuito a emular pelos vários componentes físicos, de forma a satisfazer as restrições de capacidade, número de pinos e ligações físicas impostas pela arquitectura do sistema emulador.

Um emulador de circuitos digitais comercial baseado nesta tecnologia foi introduzido em 1989 pela empresa Quickturn [Wal91]. São aceites descrições de circuitos provenientes de várias ferramentas de projecto de circuitos integrados e são produzidas configurações para os circuitos FPGA que constituem o sistema de emulação. As ferramentas de suporte permitem traduzir circuitos construídos em torno de bibliotecas de vários fabricantes de ASICs e respeitam a fidelidade da resposta temporal do circuito emulado. A rapidez anunciada actualmente é de 3 a 5 ordens de magnitude maior do que a conseguida por simulações em *software*, tornando assim possível a verificação de forma exaustiva de um sistema digital complexo [Tur97]. Isto é particularmente importante em sistemas combinados *hardware/software*, onde é necessário o desenvolvimento e verificação conjuntas de componentes de *hardware* e de *software*. Actualmente são disponíveis duas versões deste sistema, *System Realizer M250* e *System Realizer M3000*, com capacidades nominais de 250 e 3000 milhares de portas lógicas, respectivamente. Este sistema constitui uma solução integrada para emulação de circuitos lógicos, e pelo seu custo elevado é fundamentalmente orientado para ambientes de produção industrial.

O sistema SEMCI [SSAM93] desenvolvido no INESC em Lisboa é um emulador de circuitos integrados, constituído por módulos com interface idêntico, que são empilhados para satisfazer os requisitos de complexidade de uma aplicação. A construção de módulos desse tipo dedicados às necessidades do circuito a emular permite incluir no emulador funções que estão disponíveis nas bibliotecas dos fabricantes de circuitos integrados, mas que não podem ser implementadas eficazmente nos circuitos reconfiguráveis. Exemplos são circuitos analógicos (amplificadores operacionais, comparadores ou conversores) ou circuitos digitais que, por razões de complexidade, são mais facilmente integrados como circuitos discretos do que usando os recursos programáveis dos FPGAs (por exemplo memórias RAM).

Em [WZM95] é proposta a utilização de plataformas reconfiguráveis baseadas em FPGAs para emulação de faltas em circuitos lógicos. Cada porta lógica do circuito a

testar é substituída por um circuito que emula os modelos de faltas a considerar para essa porta, à custa da selecção de linhas de controlo. Para um circuito de teste estudado nesse trabalho, o número médio de portas lógicas necessárias para emular as faltas de uma única porta é igual a 18, incluindo os circuitos de controlo necessários para activar os modelos de faltas pretendidos.

Aplicações para ensino

Uma área onde sistemas de baixo custo para emulação de circuitos digitais de pequena complexidade têm particular interesse, é no ensino de sistemas digitais, substituindo montagens com circuitos integrados discretos pela configuração de circuitos FPGA. Isto permite concentrar o esforço dos alunos na componente de concepção de um projecto, reduzindo os problemas associados à montagem física e ao tempo geralmente longo de implementação de um protótipo. Exemplos de sistemas que foram desenvolvidos tendo em vista a prototipagem rápida de *hardware* digital especialmente orientados para ensino são o BORG [CSM92, Cha94], AnyBoard [Mor92, BMT+92, dB93] e o mEMAX desenvolvido pelo autor.

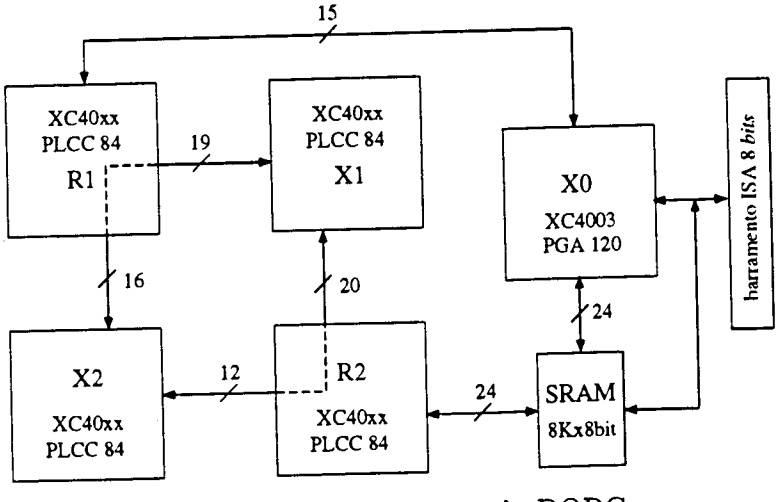


Figura 2.6: Arquitectura do BORG.

O BORG, desenvolvido na Universidade da California, contém 5 FPGAs XILINX e liga ao barramento ISA de PCs. Dois FPGAs são dedicados à implementação do sistema digital, enquanto que outros dois têm por função principal a interligação entre os primeiros. A atribuição de ligações aos terminais físicos de cada circuito é apoiada por uma ferramenta de *software*, que realiza essa função considerando as restrições impostas pelas ligações fixas existentes entre os vários circuitos. Um FPGA adicional com uma configuração

fixa lida de uma memória EPROM implementa a interface com o barramento ISA. O BORG foi inicialmente desenvolvido com FPGAs XILINX da família XC3000 [CSM92], e numa versão posterior [Cha94] foram usados FPGAs da família XC4000. Para além dos circuitos programáveis, contém também uma memória estática de 8 K*Bytes* que é atribuída ao espaço de endereçamento do PC, dois visores de 7 segmentos e uma zona de prototipagem que permite incluir num sistema realizado naquela plataforma dispositivos que não podem ser implementados nos circuitos FPGA. Na figura 2.6 [Cha94] mostra-se a arquitectura do sistema BORG. Em [Cha94] é apresentada de forma detalhada a sua implementação, e são propostos vários trabalhos práticos para este sistema que exercitam diferentes aspectos de projecto de sistemas digitais.

O AnyBoard [Mor92, BMT+92] foi desenvolvido na Universidade do Estado da Carolina do Norte (NCSU—*North Carolina State University*) tendo em vista a sua utilização no ensino de sistemas digitais, como uma plataforma configurável flexível e de baixo custo para prototipagem rápida de *hardware*. O sistema foi construído como uma carta de expansão para o barramento ISA de PCs e contém 6 FPGAs XILINX da família XC3000, sendo um dedicado à implementação do interface com o PC (figura 2.7 [Mor92]). Os restantes 5 FPGAs constituem a plataforma reconfigurável propriamente dita, e são ligados em anel, através de um barramento local dedicado, sendo também possível a comunicação entre quaisquer FPGAs através de um barramento global. O sistema tem ainda 3 memórias RAM de 128K *Bytes*, cada uma com os seus barramentos de dados e controlo ligados a 3 FPGAs distintos e um barramento de endereços comum alimentado por um quarto FPGA.

O AnyBoard é usado em PCs inseridos numa rede local, permitindo assim a utilização remota da plataforma reconfigurável inserida num PC (*design station*), por utilizadores de outras máquinas ligadas na rede. Cada *design station* contém dois sistemas AnyBoard, sendo um usado para realizar o protótipo do circuito lógico e o outro dedicado para implementar um gerador de vectores de teste para o primeiro. A utilização deste ambiente é suportada por várias ferramentas de *software*. O circuito a emular é desenhado e simulado usando uma linguagem de descrição de *hardware* desenvolvida na NCSU (SOLDER). Os projectos são mantidos numa base de dados orientada para objectos (DOOD—*Digital Object Oriented Database*) que constitui a forma de armazenar e transferir a informação nas diversas fases do projecto, entre ferramentas de desenvolvimento e entre utilizadores. Um programa para partição automática divide o projecto em circuitos lógicos que satisfaçam os requisitos de complexidade e interligações impostos para cada FPGA. Finalmente são usadas as ferramentas de implementação da XILINX para produzir as configurações para cada FPGA. Em [dB93] são exemplificados os passos seguidos para o projecto nesta pla-

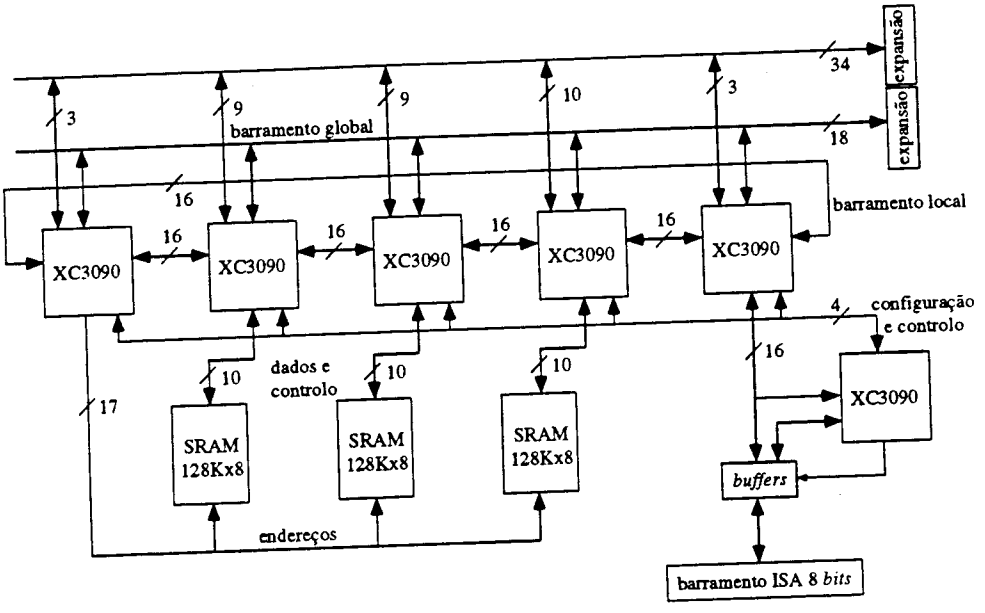


Figura 2.7: Arquitectura do AnyBoard.

taforma de um sistema detector de direcção, baseado na medida do atraso da chegada de um som a dois receptores afastados de uma distância conhecida.

O sistema mEMAX (mini EMulador de Asics com Xilinx) desenvolvido pelo autor é um sistema de emulação de baixo custo, que contém um único FPGA XILINX (XC4010, 10K portas lógicas), uma memória EPROM para a configuração do circuito e um número reduzido de componentes adicionais. A configuração do FPGA pode ser feita pela porta paralela de um PC, recorrendo a programas desenvolvidos que também permitem enviar dados para o sistema implementado no FPGA. O mEMAX oferece dois conectores de emulação: o primeiro tem 30 pinos funcionais que podem ser usados sem restrições e que mostraram ser suficientes para a realização de pequenos projectos; o segundo conector oferece 24 terminais adicionais que são também usados pela EPROM de configuração (figura 2.8).

A realização de um sistema nesta plataforma é feita com as ferramentas de implementação da XILINX, captura esquemática da ViewLogic e síntese a partir de descrições nas linguagens de descrição de *hardware* Abel ou VHDL, utilizando símbolos que foram criados para facilitar a ligação de um circuito desenhado para o FPGA com os conectores de emulação ou com a memória EPROM.

Este sistema foi construído para apoiar a disciplina "Projecto de ASICs" do 4º ano da Licenciatura em Engenharia Electrotécnica e de Computadores da Faculdade de Engenharia de Universidade do Porto, oferecendo um meio de implementação rápida de circuitos

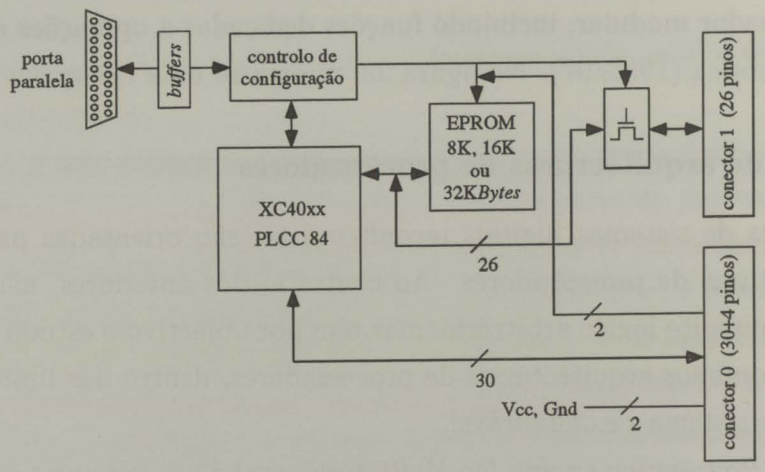


Figura 2.8: Arquitectura do mEMAX.

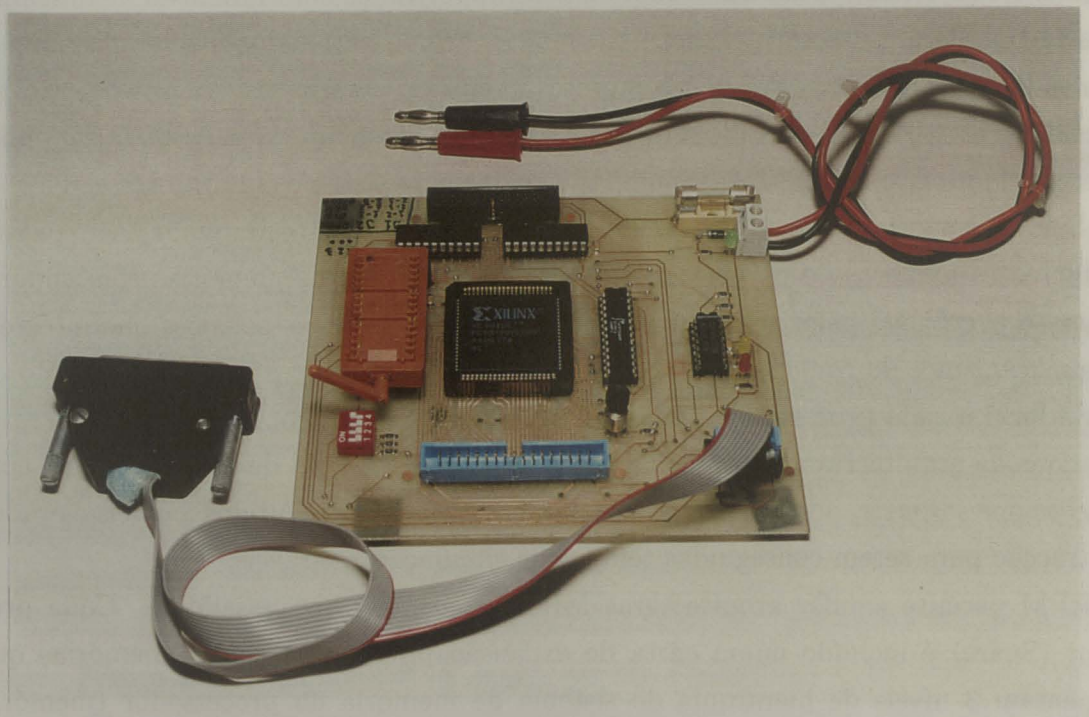


Figura 2.9: Fotografia do mEMAX.

digitais para aplicações específicas, desenvolvidos pelos alunos no âmbito dessa disciplina. Apesar da sua simplicidade, permite aos alunos a realização física e validação imediata dos seus projectos, o que seria impossível de conseguir, pelo menos em tempo útil, realizando montagens com componentes discretos MSI e LSI. Entre os projectos realizados nesta plataforma refira-se um relógio digital com função de temporização e alarme (1995/96) e

um microprocessador modular, incluindo funções dedicadas a operações de controlo para aplicações de robótica (1996/97). Na figura 2.9 mostra-se uma fotografia deste emulador.

Prototipagem de arquitecturas de processadores

Outras aplicações de sistemas digitais reconfiguráveis são orientadas para a prototipagem de arquitecturas de processadores. Ao contrário dos anteriores, não se destinam à emulação de um circuito lógico arbitrário, mas têm por objectivo o estudo e a avaliação de diferentes compromissos arquitecturais de processadores, dentro das limitações impostas pela estrutura do sistema reconfigurável.

RPM (*Rapid Prototyping engine for Multiprocessors*) é um sistema para prototipagem de arquitecturas MIMD (*Multiple Instruction Multiple Data*) de sistemas multiprocessador [BIJ+95], desenvolvido na Universidade da Califórnia do Sul, EUA (USC—*University of Southern California*). Uma das características importantes determinantes no desempenho deste tipo de arquitectura é a forma como os vários processadores comunicam entre si, nomeadamente no que se refere à comunicação entre processos executados por processadores diferentes. Se por um lado a utilização de áreas de memória partilhada simplifica a migração de programas de sistemas uniprocessador para sistemas multiprocessador, por outro lado a comunicação por passagem de mensagens permite simplificar a expansão para diferentes números de processadores e não requer o uso de memórias com suporte para acesso partilhado pelos vários processadores. A avaliação de soluções que integrem os dois mecanismos de comunicação, bem como de diferentes estratégias para gestão da memória local a cada processador, requer a comparação de desempenhos para diferentes alternativas de arquitectura. Essa comparação não pode ser feita de forma rigorosa com simulações por *software*, porque os modelos usados têm que ser mantidos a níveis elevados de abstracção para serem conseguidos tempos de simulação aceitáveis.

O RPM permite emular arquitecturas MIMD com até 8 processadores. Cada processador (Sparc) é incluído numa carta de expansão juntamente com 3 memórias que implementam 3 níveis de hierarquia do sistema de memória do processador (memória principal, e dois níveis de memória *cache*). Cada memória é controlada por um conjunto de circuitos FPGA que constituem a parte reconfigurável deste sistema. Para além do controlo das memórias, os FPGAs são também usados para implementar extensões ao conjunto de instruções do processador Sparc, controlar o funcionamento do processador e realizar os sistemas responsáveis pela comunicação com os outros processadores.

Aplicações realizadas sobre o RPM mostraram conseguir ganhos em rapidez de 100 a 1000 vezes, quando comparadas com simulações em *software*. No entanto, é referido pelos seus autores que esse ganho pode até atingir 1 milhão de vezes, se for comparado com o

tempo necessário à simulação em *software* com o mesmo nível de detalhe que é conseguido pela emulação de *hardware*.

MUSHROOM é um sistema reconfigurável criado na Universidade de Manchester, Reino Unido, dedicado à prototipagem de novas arquitecturas de processadores [Wil89, Wil91], com vista a avaliar o seu desempenho em aplicações de processamento simbólico. Segundo os seus autores, existem vários aspectos que condicionam o desempenho de processadores convencionais, quando utilizados em aplicações de processamento simbólico. Um é a forma como é gerida a memória, no que se refere principalmente à disparidade entre a dimensão dos objectos usados pelas aplicações e o tamanho das páginas de memória usadas pelo sistema de memória virtual. Outra característica associada a aplicações deste tipo resulta do facto de os tipos de dados dependerem do ambiente em que ocorrem, o que faz com que só possam ser conhecidos durante a execução do programa. Num processador convencional tal obriga a incluir instruções para verificar se um dado é ou não adequado a certa operação, o que representa um acréscimo importante no tempo de execução de uma aplicação. De forma semelhante, também as chamadas a funções dependem do ambiente em que acontecem e dos tipos dos seus argumentos. A consequência disso é que só pode ser conhecido o endereço da rotina a invocar durante a execução do programa, o que num processador clássico obriga geralmente a realizar pesquisas em tabelas de consulta, o que pode degradar de forma significativa o processo de chamada a uma função.

De forma semelhante ao RPM, também este sistema é construído em torno de circuitos integrados de uso geral e circuitos FPGA, sendo estes usados apenas naquelas partes onde se deseja introduzir reconfigurabilidade para avaliar o desempenho de diferentes arquitecturas: gestão da memória *cache*, controlo do registo apontador de programa (*program counter*) e do carregamento de instruções, controlo do banco de registos e interface com o barramento do computador principal.

ArMen é uma máquina paralela que foi construída no laboratório de Informática de Brest, França [RLRP93, CPGP94] e que associa *transputers* a FPGAs. O sistema é constituído por um conjunto de nós de processamento, contendo cada um 1 *transputer* T800, memória e um circuito programável FPGA. Os *transputers* são ligados em anel através dos seus canais de comunicação série, e os FPGAs são também ligados entre si em anel, oferecendo um canal configurável para comunicação rápida e paralela entre os *transputers*. Os circuitos implementados nos FPGAs podem trabalhar como processadores auxiliares associados a cada *transputer* ou cooperar em unidades de processamento complexas partilhadas por mais do que um processador.

Um modelo de computação proposto para esta arquitectura é baseado em execução *pipelined* onde cada estágio é implementado num nó de processamento do ArMen, comu-

nicando entre si através do anel de circuitos programáveis e interfaces construídas nos circuitos FPGA. Foram desenvolvidas ferramentas para automatizar a criação das configurações dos circuitos FPGA, considerando modelos pré-definidos para processadores auxiliares globais realizados sobre os FPGAs [CPGP94]. O desenvolvimento de aceleradores locais a cada nó é realizado usando as ferramentas de implementação para os FPGAs e bibliotecas de componentes desenhados especificamente para este sistema. O processo de programação de aplicações sistólicas nesta arquitectura é apresentado em [RLRP93] e usa a linguagem ReLaCS, desenvolvida para programar arquitecturas paralelas com suporte para comunicações sistólicas.

Em [HWGG93] é proposto um emulador para aplicações integradas de sistemas mecânicos e electrónicos. Nesta área de aplicação, a grande dificuldade reside em modelar de forma realista o comportamento de um sistema mecânico complexo, de modo a produzir os estímulos necessários para a correcta verificação e validação de um sistema electrónico para controlo de sistemas mecânicos. Para além da validação de um circuito antes da sua fabricação como um ASIC, o sistema proposto tem por objectivo permitir a emulação numa fase mais cedo do seu ciclo de projecto, possibilitando a comparação e avaliação do desempenho para diferentes decisões arquitecturais, usando como fonte de estímulos o ambiente real de funcionamento do sistema.

2.3.2 Processadores para aplicações específicas

Uma área importante de aplicação de sistemas reconfiguráveis é a implementação de processadores auxiliares, dedicados à aceleração de partes de programas executados por computadores de usos gerais. A grande motivação resulta da constatação de que programas computacionalmente intensivos gastam tipicamente a maior parte do tempo de execução em partes geralmente pequenas e bem localizadas do seu código [HP90]. Processadores auxiliares construídos como circuitos lógicos desenhados expressamente para executar de forma eficiente essas partes, permitem acelerar eficazmente programas executados por computadores equipados com processadores de uso geral. Sistemas digitais reconfiguráveis são um meio de realizar processadores auxiliares dedicados, que permite ultrapassar os principais factores que normalmente condicionam o recurso à fabricação de circuitos integrados de aplicação específica: especificidade para uma aplicação e principalmente custo e tempo de fabricação elevados.

Esta é talvez a área de aplicação de sistemas digitais reconfiguráveis onde se tem produzido mais trabalho nos últimos anos, dado os vastos domínios que podem usufruir desta tecnologia. São propostas várias arquitecturas baseadas em circuitos programáveis

FPGA, dedicadas a determinadas áreas de aplicação ou a características específicas de problemas. Sistemas deste tipo são geralmente desenvolvidos para oferecer plataformas de computação reconfiguráveis que permitam resolver problemas de forma mais eficiente do que computadores baseados em processadores de uso geral. Contudo, podem também ser vistos como meios de prototipagem rápida, com vista ao estudo e avaliação de diferentes arquitecturas para posterior realização de processadores dedicados como circuitos integrados de aplicação específica.

PAM

Um dos trabalhos pioneiros na utilização de FPGAs na construção de processadores para aplicações específicas foi realizado nos laboratórios da Digital em Paris, no fim da década de 80 [BRV89]. O conceito de memória activa programável (PAM—*Programmable Active Memories*) é definido pelos seus autores como "um agregado regular de células idênticas, todas ligadas na mesma forma repetitiva; cada célula é chamada um *bit* activo programável (PAB—*Programmable Active Bit*) e funciona como um processador capaz de armazenar e processar um único *bit* de informação. A funcionalidade permitida para um PAB deve ser suficientemente geral para permitir que qualquer sistema digital síncrono possa ser realizado (através de programação adequada) numa PAM suficientemente grande, e com uma frequência de relógio suficientemente baixa".

Uma PAM é vista por um processador como uma memória RAM, na qual pode ler e escrever dados. No entanto, ao contrário da memória RAM, a PAM processa a informação entre a escrita e a leitura, sendo o tipo de processamento ditado pelo conteúdo da sua memória de configuração. Foram implementadas duas arquitecturas PAM, usando FPGAs XILINX XC3000, onde cada PAB é implementado por um bloco lógico configurável (CLB) daquela família de FPGA, incluindo um *flip-flop* e podendo realizar qualquer função lógica combinacional de 4 entradas. O sistema DECPeRLe0 tem 25 FPGAs XC3020, oferecendo um total de 3200 PABs e 512 K*Bytes* de memória RAM, podendo trabalhar com frequências máximas de 30 MHz. Dois FPGAs adicionais realizam a interface com o barramento do computador principal. O sistema DECPeRLe1 tem 23 FPGAs XC3090 (14000 PABs), 4 M*Bytes* de RAM e pode operar com frequências até 100 MHz (figura 2.10 [VBR+96]). Ambos os sistemas suportam a leitura do estado interno dos FPGAs e juntamente com a possibilidade de provocar transições de relógio sob controlo do computador principal facilitam as tarefas de detecção e depuração de erros de um sistema realizado sobre estas plataformas.

A implementação de um algoritmo na PAM necessita da selecção da secção crítica do problema a acelerar. Isto é feito manualmente e requer a avaliação da influência

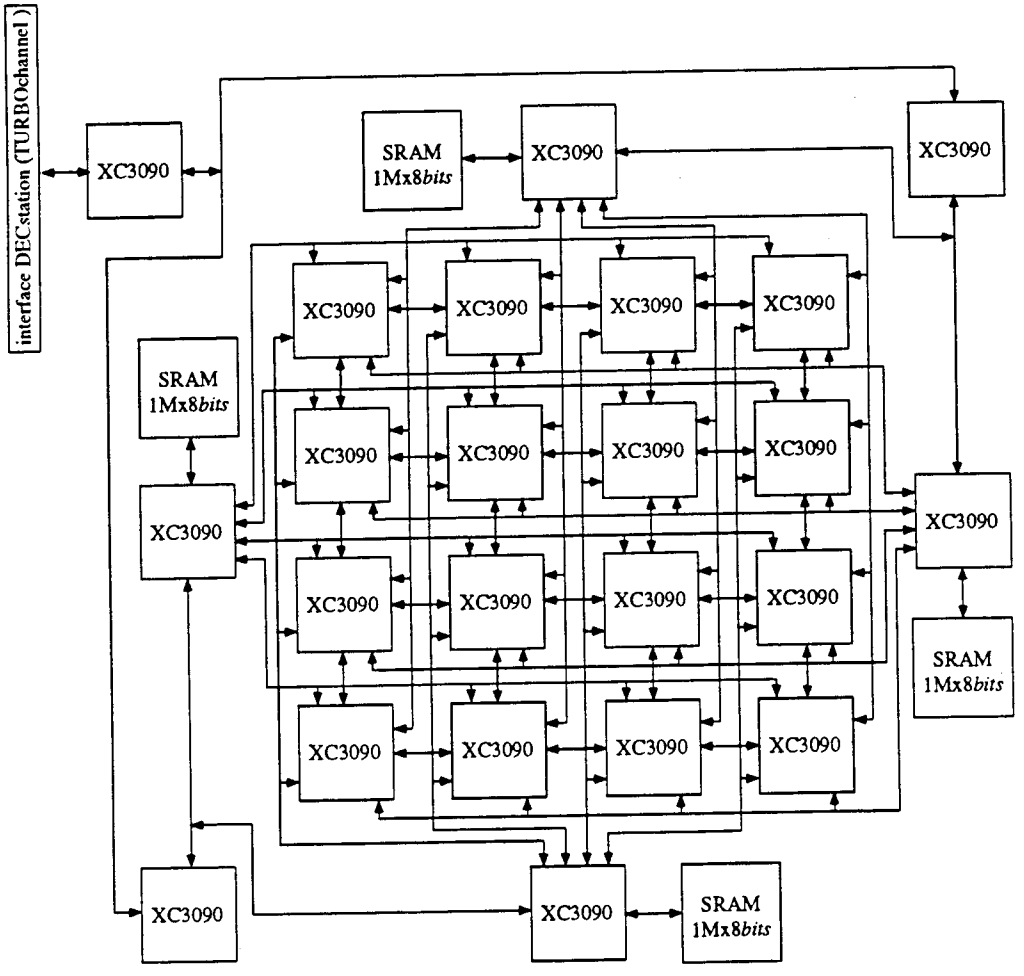


Figura 2.10: Arquitectura do DECPeRLe1.

da comunicação dos dados e resultados entre o processador principal e a PAM. A programação da plataforma reconfigurável é feita ao nível da função realizada em cada PAB e inclui informação que define o posicionamento de cada PAB no agregado de circuitos programáveis. Esta operação é automatizada por um conjunto de ferramentas que permitem descrever o problema numa linguagem de alto nível anotada com informação de posicionamento geométrico, recorrendo a uma biblioteca de funções dedicadas para a implementação na PAM. As ferramentas produzem listas de ligações que são traduzidas para as configurações dos FPGAs com o sistema de desenvolvimento e implementação da XILINX. O sistema Perle1DC para configuração das PAM apresentado em [BT94], utiliza a linguagem de programação C++, redefinindo os operadores lógicos e introduzido um tipo de dado `Net`, usado para representar sinais lógicos. A descrição do *hardware* é feita ao nível estrutural e é anotada com informação geométrica que define a divisão do

circuito lógico pelos vários dispositivos programáveis e a posição relativa que cada bloco físico deve ocupar no espaço configurável da PAM.

Em [BRV92] são apresentadas 10 aplicações que foram implementadas nas PAM e que foram seleccionadas por apresentarem partes críticas bem definidas passíveis de serem bem realizadas nos recursos programáveis da PAM. Para todos os casos, o nível de desempenho atingido foi comparável ou mesmo superior ao conseguido com super-computadores ou circuitos integrados dedicados. Os desempenhos anunciados para esse conjunto de aplicações vão desde 300 MIPs para um sistema para pesquisar palavras próximas num dicionário (implementado no DECPeRLe0), 15000 MIPs para um calculador da transformada directa de cosseno (DCT) que realiza 48 operações sobre números de 32 *bits* em vírgula fixa em cada ciclo de 40ns (DECPeRLe1) e 25000 MIPs para um calculador da equação de Laplace, baseado no método das diferenças finitas (DECPeRLe1).

Cada implementação foi realizada e testada em menos de dois meses, que representam tempos comparáveis aos necessários para produzir implementações optimizadas em *software* para super-computadores, mas uma ordem de magnitude inferior ao tempo necessário para projectar e fabricar um circuito integrado de aplicação específica. A arquitectura das PAM possibilita ainda a sua utilização em aplicações de processamento em tempo real que necessitem de grande largura de banda de comunicação de dados com o exterior, que não podem ser conseguidas com super-computadores sem o recurso a equipamentos auxiliares.

PRISM

O projecto PRISM da Universidade de Brown (*Processor Reconfiguration through Instruction Set Metamorphosis*) constitui uma das primeiras abordagens na utilização de sistemas reconfiguráveis para processadores auxiliares de um processador convencional, integrando no processo de compilação de uma linguagem convencional de programação (C), a identificação das secções da aplicação a realizar no processador auxiliar e a tradução automática dessas partes para *hardware*.

O primeiro protótipo, PRISM-I [AS93], foi construído com 4 FPGAs XILINX XC3090 (9K portas lógicas), incluído numa carta com um processador Motorola 68010. Cada FPGA da plataforma reconfigurável pode implementar um operador combinacional, recebendo como operando uma palavra de 32 *bits* e produzindo como resultado também uma palavra de 32 *bits*. Apesar da simplicidade do sistema, foram conseguidos ganhos de rapidez que atingiram as 54 vezes para aplicações seleccionadas, comparando os tempos de execução com e sem a utilização de operadores implementados na plataforma reconfigurável. O sistema de compilação analisa funções de um programa em C para seleccionar

potenciais candidatas a traduzir para *hardware*. Isto requer que o programador tenha em atenção a forma como deve estruturar o código, de forma a construir funções que apresentem complexidade compatível com o que pode ser realizado na plataforma reconfigurável. As funções candidatas são seleccionadas de acordo com o impacto no desempenho global do programa, sendo necessária interacção posterior com o utilizador para escolher a função a implementar na plataforma reconfigurável. O processo de construção do *hardware* é feito a partir do código fonte em C, traduzindo a descrição algorítmica para equações lógicas e usando ferramentas de *software* para partição do circuito lógico pelos recursos configuráveis dos circuitos FPGA. O programa fonte original é modificado de forma a incluir as instruções necessárias para invocar o operador criado na plataforma reconfigurável.

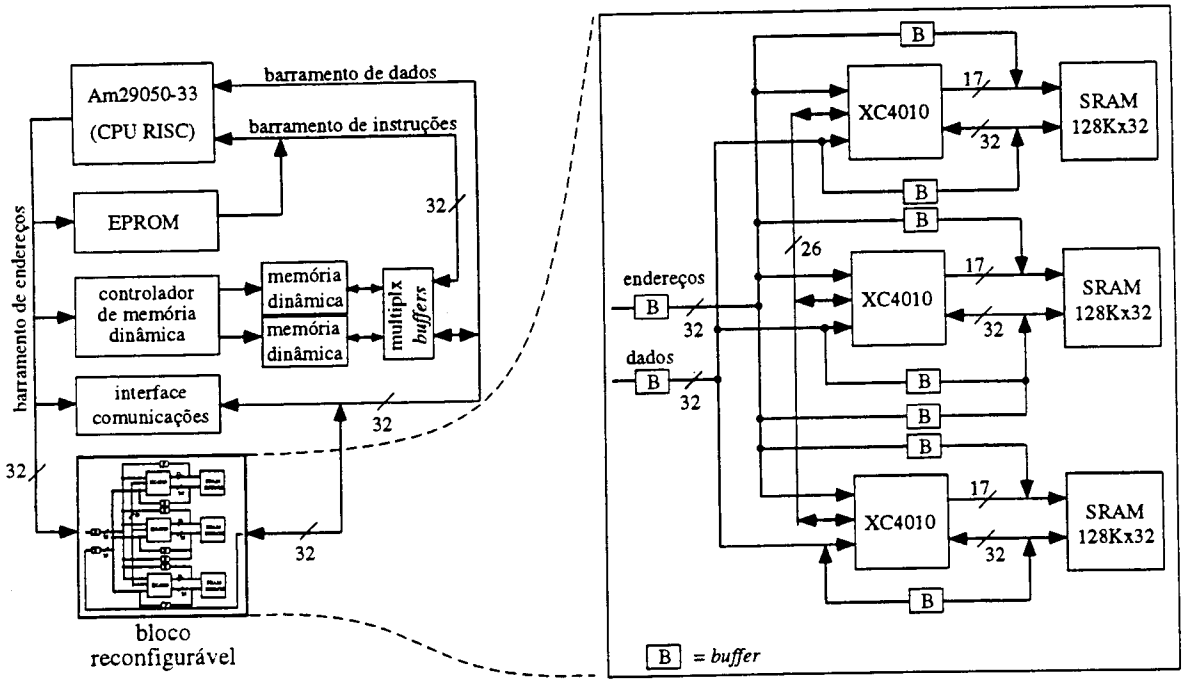


Figura 2.11: Arquitectura do PRISM-II.

O sistema PRISM-II [AWG94] é o sucessor do PRISM-I e suporta a compilação de funções que contenham ciclos com número de iterações desconhecido na fase de compilação. Neste sistema é introduzido um novo modelo de execução assíncrona, onde as operações são executadas logo que os seus operandos estejam disponíveis. O *hardware* gerado é constituído por uma rede de operadores e um controlador que define a sequência de operações a realizar de forma assíncrona e dependente dos operandos, sinalizando o processador principal quando é produzido o resultado final. A plataforma de *hardware* do PRISM-II é constituída por um processador RISC (Am29050-33MHz), memória dinâmica, circuitos de interface para comunicações e um bloco de lógica reconfigurável.

A parte reconfigurável contém 3 circuitos FPGAs XILINX XC4010, cada um associado a uma memória estática e interligados através de interruptores programáveis electricamente. Deste modo é possível utilizar os 3 FPGAs como operadores individuais ou dividir a funcionalidade de um operador complexo pelos 3 FPGAs do bloco reconfigurável. Na figura 2.11 [SWSS95] apresenta-se a arquitectura do PRISM-II.

O sistema de compilação do PRISM-II assume que as secções críticas a traduzir para *hardware* são funções identificadas e seleccionadas pelo programador. A tradução é feita usando o compilador de C da GNU para interpretar e optimizar o código fonte em C. A síntese da rede de operadores e do controlador é feita com análise do grafo de fluxo de dados (DFG—*Data Flow Graph*) e do grafo que representa a sequência de controlo das operações (CFG—*Control Flow Graph*), respectivamente. Um exemplo ilustrativo da aplicação do algoritmo de síntese das duas componentes (*hardware* e *software*) para um operador dedicado é apresentado em [AWG94], com a implementação de uma função que calcula o quociente da divisão entre dois números inteiros. A implementação física nos circuitos FPGA é realizada com as ferramentas da XILINX, a partir de especificações na linguagem de descrição de *hardware* VHDL, ou usando a biblioteca de módulos parametrizáveis XBLOX [XBL94] integrada nas ferramentas de desenvolvimento da XILINX [WAL+93].

Em [SWSS95] apresenta-se uma implementação de algoritmos genéticos ao problema de partição de um circuito lógico por vários FPGAs, na máquina paralela Armstrong III constituída por 20 nós PRISM-II. Nesta aplicação é anunciado um ganho de desempenho de 3 vezes usando um único nó PRISM-II, e cerca de 8 vezes com uma implementação distribuída por vários processadores, em relação a implementações em *software* executadas numa estação de trabalho SPARCstation 20/61.

Splash

Splash é um processador paralelo constituído por uma arquitectura linear de circuitos programáveis que foi desenvolvido no *Supercomputing Research Center*, EUA com o objectivo de investigar a implementação de algoritmos sistólicos em *hardware* dedicado [GHK+91]. O sistema contém 32 FPGAs XILINX XC3090, com 128 K*Bytes* de memória RAM associada a cada um. O interface com o computador principal é realizado através do barramento VME de uma estação de trabalho SUN, controlado por um FPGA adicional. Outro FPGA controla o interface com uma carta de memória acessível pelo computador principal e pelo Splash. Os circuitos FPGA são programados ao nível lógico usando uma linguagem dedicada à programação daquela arquitectura (LDG—*Logic Description Generator*). Este sistema obteve sucesso em aplicações de comparação de sequências de nucleótidos em cadeias de DNA, onde foi conseguido um ganho de 45 vezes no tempo de execução relati-

vamente ao tempo gasto por um sistema dedicado a essa aplicação (P-NAC—*Princeton Nucleic Acid Comparator*) [GHK+91].

Splash 2 é o sucessor do Splash [BAK96] (figura 2.12). São utilizados FPGAs XILINX XC4010 associados a memórias de 512 K*Bytes* e o sistema é expansível possibilitando interligar em cadeia um máximo de 16 cartas com 17 FPGAs cada. Foi aumentada a conectividade entre os FPGAs de cada carta recorrendo a circuitos de interligações configuráveis (*crossbar network*) que permitem estabelecer conectividade entre os circuitos programáveis, armazenando até 8 configurações de ligações diferentes que podem ser trocadas dinamicamente em cada ciclo de relógio. A transferência de dados entre o computador principal e o Splash 2 é feita através de canais de acesso directo à memória (DMA) controlados por dois FPGAs adicionais, o que permite atingir taxas máximas de transferência de dados com o computador principal de 50 M*Bytes*/s.

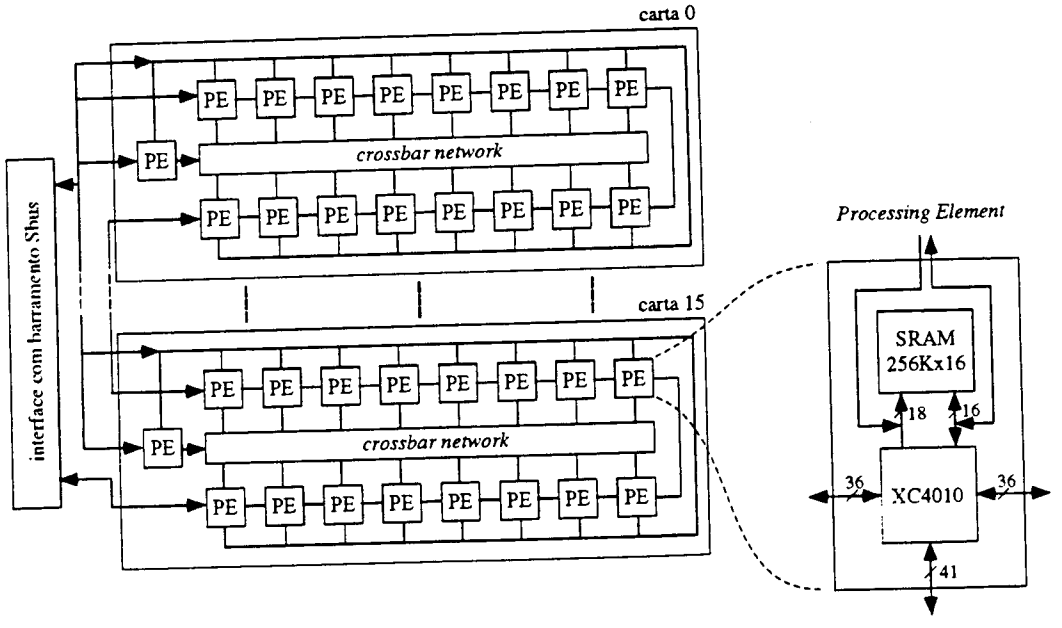


Figura 2.12: Arquitectura do Splash 2.

O desenvolvimento de aplicações para este sistema é feito a partir de descrições em VHDL, oferecendo um ambiente de simulação, biblioteca de componentes do sistema e ferramentas de compilação para os circuitos programáveis (FPGAs e circuitos de interligações) [Arn93]. Uma aplicação é descrita em VHDL e é dividida manualmente pelos vários elementos de processamento (constituídos por um FPGA e memória), usando ferramentas de *software* para optimizar e compilar as descrições em VHDL para cada circuito programável. Uma biblioteca em C disponibiliza as funções necessárias para construir os programas a executar no computador principal com interfaces específicas para cada

aplicação. Foi também desenvolvido um programa interactivo para controlo e depuração simbólica de uma aplicação executada no Splash 2. A mesma equipa propõe em [GM93] um sistema para programação desta arquitectura a partir de uma linguagem de programação paralela de alto nível (dbC–*Data-parallel Bit-serial C*).

Entre as aplicações que foram programadas com sucesso nesta arquitectura e que conseguiram ganhos de desempenho importantes em relação a implementações em *software*, refira-se a detecção de palavras em sequências de texto [PTS93], pesquisa de sequências de genes em bases de dados genéticas [Hoa93] e algoritmos para processamento de imagem em tempo real [AACE94, AA95, RJR95].

Spyder

O sistema Spyder (*Système de Développement à Processeur Reconfigurable*), desenvolvido na Escola Politécnica Federal de Lausanne, Suíça, é um processador VLIW (*Very Large Instruction Word*) com 3 unidades de execução reconfiguráveis construídas com circuitos FPGA [IS93, Ise96], que foi implementado como uma carta para o barramento VME de uma estação de trabalho SUN (figura 2.13 [Ise96]). O Spyder é visto pelo sistema operativo Unix como um dispositivo periférico, comandado através de canais para programar os FPGAs, ler e escrever dados e ler o estado interno dos circuitos programáveis.

O processador executa instruções de 128 *bits* que controlam directamente os elementos do processador, sem envolver qualquer operação de descodificação. Todas as instruções são executadas em 4 ciclos de relógio e de forma *pipelined* com um ciclo de latência, o que permite iniciar a execução de uma instrução em cada ciclo de relógio. São usadas memórias separadas para dados e programa, e dois bancos de registos que realizam a interface entre a memória de dados e as unidades de execução. De forma semelhante a uma arquitectura RISC, todas as operações realizadas pelas unidades de execução consomem dados de registos e produzem resultados para registos. A existência de dois bancos de registos permitindo o acesso simultâneo a dois operandos por cada uma das 3 unidades de execução possibilita o funcionamento em paralelo dessas unidades.

As unidades de execução são implementadas por circuitos FPGA, e são configuradas como operadores específicos para cada aplicação. Esses operadores podem ser obtidos de uma biblioteca já construída, ou desenhados de acordo com as necessidades da aplicação. Cada unidade recebe dois operandos de 16 *bits*, um de cada banco de registos, produzindo dois resultados de 16 *bits*, também um para cada banco de registos.

A implementação de uma aplicação neste processador requer a realização de três tarefas distintas: seleccionar e construir o conjunto de 3 operadores a implementar nas unidades de execução, construir o programa de controlo do processador e desenvolver a

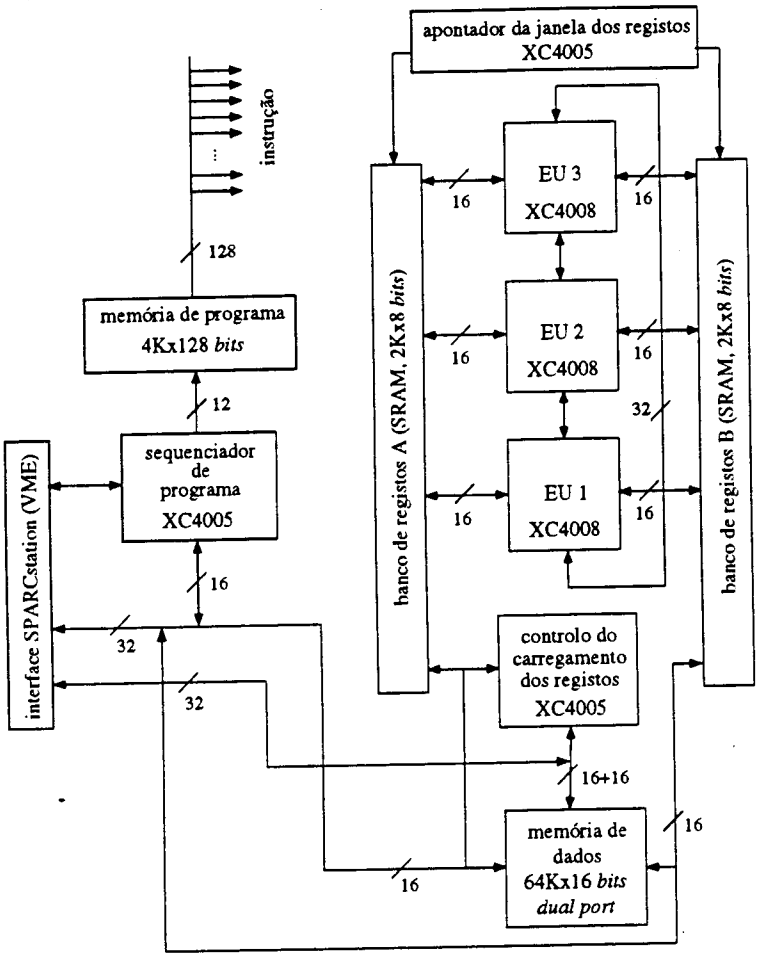


Figura 2.13: Arquitectura do Spyder.

interface entre o sistema operativo da estação de trabalho e a aplicação realizada pelo processador dedicado. Estas tarefas são automatizadas por ferramentas de *software* que utilizam a linguagem de programação C++ para especificar as várias partes que constituem a aplicação [IS95, Ise96]. A utilização da mesma linguagem de programação para especificar as várias componentes de uma aplicação possibilita a sua execução no processador principal, o que simplifica a tarefa de detecção e correcção de erros a um nível elevado de abstracção.

Um exemplo de uma aplicação programada nesta arquitectura é o jogo *life*. Trabalhando com uma frequência de relógio de 8 MHz, a implementação no Spyder consegue calcular cerca de 17 vezes mais células por segundo do que o programa `xlife`[2] executando numa estação de trabalho SPARCstation 5. Outras aplicações relatadas em [Ise96] onde se obtiveram ganhos de algumas dezenas de vezes no tempo de execução, comparando com

[2] `xlife` corre sobre o sistema X Windows e é disponível por ftp anónimo em `ftp.x.org`

realizações em *software* são operações de processamento de imagem ("emagrecimento" de linhas (*thinning*) e detecção de arestas).

2.3.3 Integração de processadores e plataformas reconfiguráveis

A utilização de circuitos FPGA para implementar processadores para aplicações específicas tem o inconveniente do baixo desempenho e densidade de integração, quando comparado com circuitos integrados fabricados numa tecnologia semelhante. Segundo é referido em [LvIW93], FPGAs apresentam tipicamente uma densidade de integração 10 vezes menor e uma rapidez 3 vezes inferior, quando comparados com versões programadas por máscara. Para além disso, as limitações resultantes da comunicação de dados com um computador principal usando barramentos de usos gerais, faz com que sistemas reconfiguráveis construídos como periféricos externos de processadores convencionais apenas obtenham vantagens significativas na resolução de problemas com granularidade elevada. Nesses casos consegue-se explorar o paralelismo e as características das operações de uma aplicação para conseguir realizar de forma paralela aquilo que um processador convencional tem de fazer de modo sequencial. Como é referido em [JEÖH94], outro factor que também limita o desempenho conseguido por processadores auxiliares externos, é o estrangulamento resultante do acesso lento à memória principal, sem tirar partido das memórias *cache* internas do processador.

O recurso a plataformas reconfiguráveis para acelerar operações de pequena granularidade, de nível semelhante ao das instruções usuais dos processadores, obriga a uma integração íntima entre um processador e essa plataforma reconfigurável. Isso permite que circuitos lógicos construídos à medida para uma aplicação tenham acesso pleno aos recursos internos do processador, ultrapassando também as limitações de rapidez impostas pelo uso dos barramentos externos do processador. A ideia de juntar num mesmo circuito integrado um processador de usos gerais com circuitos lógicos reconfiguráveis foi já sugerida por vários autores [DeH94, Raz94, HW97], que também propõem a sua reconfiguração rápida durante a execução de uma aplicação, tornando possível a utilização de reconfiguração dinâmica. Noutros trabalhos são propostos novos circuitos do tipo FPGA baseados em blocos configuráveis de granularidade mais elevada da que é usual nos circuitos FPGA comerciais, tornando-os adequados a domínios de aplicação específicos [QKSZ94, HKR94]. Em [WHG94, WH96a, WC96] são apresentadas realizações de processadores em FPGAs comerciais, que utilizam recursos do próprio FPGA para implementar funções específicas para diferentes aplicações.

nP: nano processador

Como o nome sugere, o nano processador (nP) é um processador muito simples suportando um conjunto reduzido de 6 instruções que é realizado num único FPGA [WHG94]. A utilização de um número reduzido de recursos do FPGA permite deixar espaço disponível para aumentar o seu conjunto de instruções, à custa de circuitos lógicos que implementem instruções dedicadas às necessidades de diferentes aplicações. O núcleo do processador ocupa apenas 24 blocos lógicos configuráveis de um FPGA XILINX XC3000, deixando livres até 92% desses recursos, dependendo do dispositivo utilizado.

Uma aplicação comercial deste sistema é a carta de aúdio X2, da National Technologies Inc. Esta carta implementa um sistema reconfigurável com FPGAs XILINX para processamento de audio digital para PCs, permitindo a sua utilização numa gama variada de aplicações. Entre as várias configurações desenvolvidas para este sistema incluem-se aplicações do nP que utilizam instruções dedicadas a diferentes operações de processamento de audio e controlo do sistema de aquisição.

PRISC

O acrónimo PRISC (*Programmable Reduced Instruction Set Computer*) foi introduzido em [Raz94, RS94] e designa uma arquitectura RISC acrescida de unidades funcionais programáveis (PFU—*Programmable Functional Units*) incluídas no mesmo circuito integrado de forma a estender o conjunto de operações convencionais suportadas pela unidade lógica e aritmética (ALU). Um processador PRISC explora a utilização de *hardware* dedicado ao nível de operações de pequena complexidade, que são identificadas e traduzidas por um compilador adequado para configurações da área programável que constitui as PFU. A arquitectura proposta para uma PFU só pode realizar funções lógicas combinacionais e é baseada em 3 níveis de lógica realizados por tabelas de consulta construídas como blocos de memória RAM estática e interligados por interruptores programáveis, também controlados por células de memória estática.

O processo de compilação inclui uma fase designada pelos seus autores por "extracção de *hardware*", após a geração de código máquina para o processador principal por um compilador convencional. Nessa fase são identificadas sequências de instruções que possam ser realizadas por funções lógicas combinacionais e são sintetizados os circuitos a implementar nas PFU. Resultados obtidos por simulação, considerando um processador MIPS R2000 trabalhando a 200 MHZ e com uma única PFU, mostram que o ganho médio de desempenho para um conjunto de programas de teste (SPECint92) é de 22%, atingindo um máximo de 91%. Estes números foram obtidos considerando que a PFU é sempre pro-

gramada antes de ser invocado o operador nela realizado, gastando nessa operação 500 ciclos de relógio. À custa de optimizações manuais no código máquina gerado pelo compilador, de forma a criar sequências de operações mais adequadas à implementação nas PFU, são conseguidos ganhos de desempenho que atingem 500%, comparando o tempo gasto a executar a sequência original de instruções e o tempo de execução da operação realizada na PFU.

DISC

DISC (*Dynamic Instruction Set Computer*) [WH96a, WH96b] é um processador constituído por um núcleo programável e uma área reconfigurável onde são implementados módulos de *hardware* que realizam as instruções necessárias a uma aplicação. Ao contrário do nano processador apresentado antes, a utilização de FPGAs da National Semiconductor CLAy31 que suportam reconfiguração parcial, permitiu construir um sistema onde as instruções são carregadas de forma dinâmica à medida que são necessárias durante a execução de uma aplicação.

A primeira versão deste processador [WH96a] foi realizada num único FPGA e contém um controlador global que implementa um conjunto elementar de instruções, gastando apenas cerca de 1/6 dos recursos configuráveis do circuito. Os restantes blocos configuráveis do FPGA são vistos como um espaço linear de *hardware*, que é ocupado dinamicamente por módulos previamente construídos para realizarem as instruções necessárias a cada aplicação. Durante a execução de uma aplicação, são carregados os circuitos necessários para executar diferentes instruções, re-utilizando o espaço ocupado por instruções usadas anteriormente. O mecanismo de reconfiguração parcial desta família de FPGA permite reconfigurar apenas a parte do circuito onde é colocado um bloco de *hardware* e possibilita também a sua colocação em zonas diferentes do espaço físico do circuito, tornando assim mais eficaz a forma de ocupação do *hardware* reconfigurável.

A segunda versão desta arquitectura, DISC-II [WH96b], foi implementada numa carta comercial da National Semiconductor e utiliza dois FPGAs CLAy para realizar o processador: num é implementado o núcleo do processador e o outro é usado para o espaço de *hardware* reconfigurável, onde são colocados os módulos que constituem as instruções necessárias para cada aplicação. Numa aplicação de processamento de imagem são anunciados ganhos de rapidez entre 6.5 e 10 vezes, comparando com uma versão executada num PC 486 funcionando a 66 MHz. Estes resultados foram obtidos com o DISC-II a trabalhar a 8 MHz, e gastando uma parte importante do tempo de execução no processo de reconfiguração do *hardware*. Na figura 2.14 [WH96b] mostra-se a arquitectura deste sistema e a forma como é organizado o espaço físico do FPGA para carregar dinamicamente

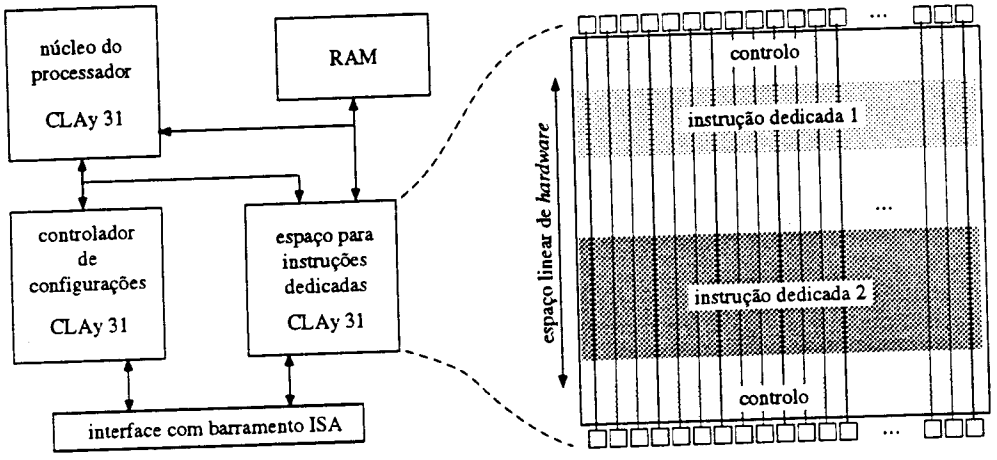


Figura 2.14: Arquitectura do DISC-II.

as instruções necessárias à execução de uma aplicação.

O ambiente de programação para este sistema utiliza a linguagem de programação C e requer a identificação manual das instruções específicas necessárias a cada aplicação [CH96]. A construção dos módulos que implementam essas instruções tem de ser feita realizando a colocação e interligação manuais dos blocos configuráveis do FPGA, de forma a satisfazer restrições as físicas impostas pela arquitectura do sistema.

Garp

Garp é uma proposta de um circuito integrado que combina um núcleo de um processador convencional (MIPS) e uma área reconfigurável similar a um FPGA [HW97], em curso na Universidade de Berkeley, EUA. A parte reconfigurável é constituída por um agregado de blocos lógicos semelhantes aos encontrados nos FPGAs comerciais, e tem acesso à memória externa através da mesma hierarquia de memória *cache* vista pelo processador. Nesta arquitectura, a configuração e utilização do agregado lógico reconfigurável é realizada sob controlo do programa executado pelo processador principal.

A parte reconfigurável do sistema só pode ter activa uma configuração de cada vez, embora seja suportada reconfiguração parcial, permitindo reduzir o tempo necessário à configuração de circuitos que não utilizam todos os blocos configuráveis. Distribuída pelos blocos lógicos configuráveis existe uma memória *cache* onde são mantidas as configurações mais usadas.

A programação para esta arquitectura é feita usando um compilador de C convencional, e um tradutor de linguagem *assembly* modificado para incluir instruções que invocam os operadores realizados na parte reconfigurável. A especificação da configuração da parte

reconfigurável é feita pelo programador, definindo a programação dos blocos configuráveis, numa linguagem de baixo nível semelhante à linguagem *assembly*.

Comparações de uma implementação hipotética do Garp com um processador UltraSPARC 1/170 mostram tempos de execução de 2 a 24 vezes menores e favoráveis ao primeiro, trabalhando com uma frequência de relógio de apenas 80% do processador SPARC. Estes números foram obtidos por simulação e para aplicações seleccionadas que tiram partido da arquitectura da área reconfigurável proposta.

2.3.4 Sistemas reconfiguráveis dinamicamente

Reconfiguração dinâmica permite re-utilizar o mesmo *hardware* físico para realizar em sequência várias tarefas não simultâneas em que um problema pode ser dividido. A principal dificuldade na aplicação desta técnica reside nos tempos de reconfiguração relativamente longos exibidos pelos circuitos FPGA actuais, o que num grande número de aplicações condiciona a eficiência que pode ser conseguida pela utilização de circuitos lógicos dedicados a uma aplicação. No entanto, o uso de reconfiguração dinâmica permite realizar um sistema digital utilizando menos *hardware* físico do que seria necessário usando reconfiguração estática.

O processador DISC [WH96a] apresentado antes é um exemplo de um sistema deste tipo. Apesar de uma parte significativa do tempo de execução de aplicações executadas nesta arquitectura ser gasto na troca de configurações, permite dotar um processador programável de um conjunto de instruções muito superior ao que seria possível manter em simultâneo nesse circuito lógico. Os sistemas PRISC e Garp são também propostas que exploram reconfiguração dinâmica, embora usando arquitecturas de circuitos reconfiguráveis que foram criadas propositadamente para essas aplicações.

A primeira aplicação conhecida de sistemas reconfiguráveis dinamicamente baseados em FPGAs é o RRANN (*Run-time Reconfiguration Artificial Neural Network*), um processador dedicado para treino de redes neuronais [EH94, EH96]. O sistema implementa o algoritmo de propagação reversa (*backpropagation*) para treinar uma rede neuronal, dividindo-o em três tarefas que são realizadas em sequência: avaliação da resposta da rede, propagação reversa dos resultados e actualização dos parâmetros que caracterizam a rede neuronal. Cada parte do algoritmo é implementada reconfigurando o mesmo *hardware* físico com os circuitos lógicos que realizam cada parte. Este sistema foi implementado numa carta comercial com FPGAs XILINX XC3090 e permitiu aumentar em 5 vezes o número de neurónios processados, quando comparado com uma realização na mesma plataforma sem usar reconfiguração dinâmica.

Em [HH95] é apresentada uma evolução deste sistema (RRAN2), usando FPGAs CLAy com suporte para reconfiguração parcial. Esta característica desta família de FPGA permite reconfigurar a função realizada por um circuito, programando apenas os recursos que são modificados entre duas configurações diferentes. A redução do tempo gasto nas reconfigurações foi conseguida optimizando as partes comuns entre os 3 circuitos que implementam cada uma das fases do algoritmo, de forma a ocuparem os mesmos recursos configuráveis. Para conseguir isto, foi necessário desenhar manualmente os circuitos lógicos e implementá-los também manualmente nos recursos programáveis do FPGA, de forma a maximizar as partes de cada circuito lógico que são mantidas de uma configuração para a seguinte.

Uma arquitectura para um circuito FPGA que pode manter várias configurações em simultâneo é proposta em [DeH94], possibilitando a troca rápida entre diferentes configurações. Esta arquitectura é sugerida como um bloco a incluir em futuros processadores de usos gerais, de forma a oferecer um meio para implementar circuitos lógicos dedicados a diferentes funções requeridas durante a execução de uma aplicação. Em [LA93] é proposta uma máquina reconfigurável baseada no mesmo princípio (WASMII—*What A Stupid Machine It Is*), para processamento conduzido pelos dados (*data-driven computing*). Neste tipo de aplicação, as funções realizada pela plataforma reconfigurável são modificadas dinamicamente, em função dos dados e resultados envolvidos durante o processamento.

Uma aplicação a processamento de vídeo que utiliza reconfiguração dinâmica é apresentada em [SjV95]. Usando um único FPGA CLAy (5K portas lógicas) foi implementado um codificador/descodificador de vídeo, operando a uma cadência de 8 imagens por segundo. Se não fosse usada reconfiguração dinâmica seriam necessários 3 FPGAs daquele tipo para realizar o mesmo sistema. Nesta aplicação o tempo gasto nas reconfigurações representa apenas 4% do tempo disponível entre cada imagem ($125\mu s$).

Em [LM95] é apresentada uma aplicação de reconfiguração em tempo real no sistema DECPerLe-1 para pesquisa de bases de dados genéticas. Ao contrário das aplicações referidas atrás, reconfiguração dinâmica é aqui entendida num contexto de utilização interactiva, personalizando os FPGAs com base em informação dada por um utilizador. Definida a sequência genética a pesquisar, é criado um circuito lógico optimizado para pesquisa dessa sequência e é configurada a plataforma reconfigurável. Nesta aplicação são anunciados ganhos de desempenho de 2 a 3 ordens de magnitude em relação a implementações em *software* executadas numa estação de trabalho DECstation 5000/240. Esse ganho é traduzido em tempos aproximados de 100 segundos para pesquisar a base de dados genética humana, tornando assim praticável a sua utilização de forma interactiva.

2.3.5 Interfaces flexíveis com equipamentos de instrumentação

Uma aplicação interessante de sistemas reconfiguráveis é na realização de interfaces flexíveis entre computadores e equipamentos electrónicos de instrumentação ou de aquisição de dados. O uso de circuitos reconfiguráveis facilita a construção de interfaces que implementem protocolos de comunicação adaptados às exigências de diferentes equipamentos. Para além disso, a possibilidade de realizar processamento local da informação recolhida, ou de sintetizar dados a enviar para equipamentos daquele tipo, liberta dessas tarefas o processador principal e também reduz a quantidade de informação a transferir com o sistema que implementa a interface. Em certas situações, isso é importante como meio de conseguir ultrapassar limitações de rapidez impostas pelo tipo de interface disponível entre o computador principal e o equipamento de instrumentação.

Um exemplo de uma aplicação deste tipo é o processador dedicado XBST que foi implementado sobre o sistema reconfigurável ISAX, ambos desenvolvidos pelo autor. O processador dispõe de interface para a infra-estrutura normalizada de teste digital BST (*Boundary Scan Test, standard IEEE 1149.1* [Tes90]) e tem sido utilizado com sucesso num trabalho de investigação na área do teste de circuitos com sinais mistos analógicos e digitais [dSLAM97]. Neste trabalho, o processador dedicado é responsável por controlar a interface BST de um circuito integrado que implementa o acesso a um barramento dedicado ao teste de circuitos analógicos, actualmente em processo final de normalização pelo IEEE (proposta de normalização IEEE P1149.4).

O sistema reconfigurável ISAX foi construído pelo autor para o barramento ISA de PCs para permitir a realização rápida de sistemas digitais com interface para este barramento. O sistema contém dois FPGAs XILINX da família XC4000, 256 K*Bytes* de memória RAM estática e circuitos dedicados ao controlo da interface com o barramento ISA. Foram desenvolvidos programas para possibilitar a configuração dos FPGAs e a transferência de blocos de dados com a memória, usando operações de leitura e escrita em portas de entrada/saída do PC. Na figura 2.15 mostra-se a arquitectura deste sistema e na figura 2.16 apresenta-se uma fotografia da carta onde foi realizado, inserida no barramento de expansão de um PC.

O processador dedicado desenvolvido nesta plataforma inclui um conjunto de instruções específicas para controlo da cadeia BST, que é compatível ao nível do código *assembly* com as instruções suportadas por um processador construído anteriormente para esta aplicação [Fer92]. Deste modo foi possível utilizar *software* já existente para gerar programas em *assembly* para controlo da infra-estrutura BST.

A utilização de uma plataforma reconfigurável como meio de realização física deste processador possibilitou o seu desenvolvimento incremental e a adaptação rápida do seu

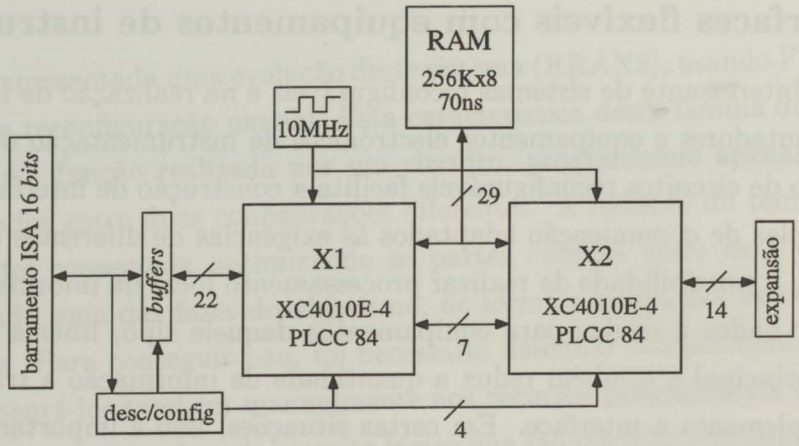


Figura 2.15: Arquitectura do ISAX.

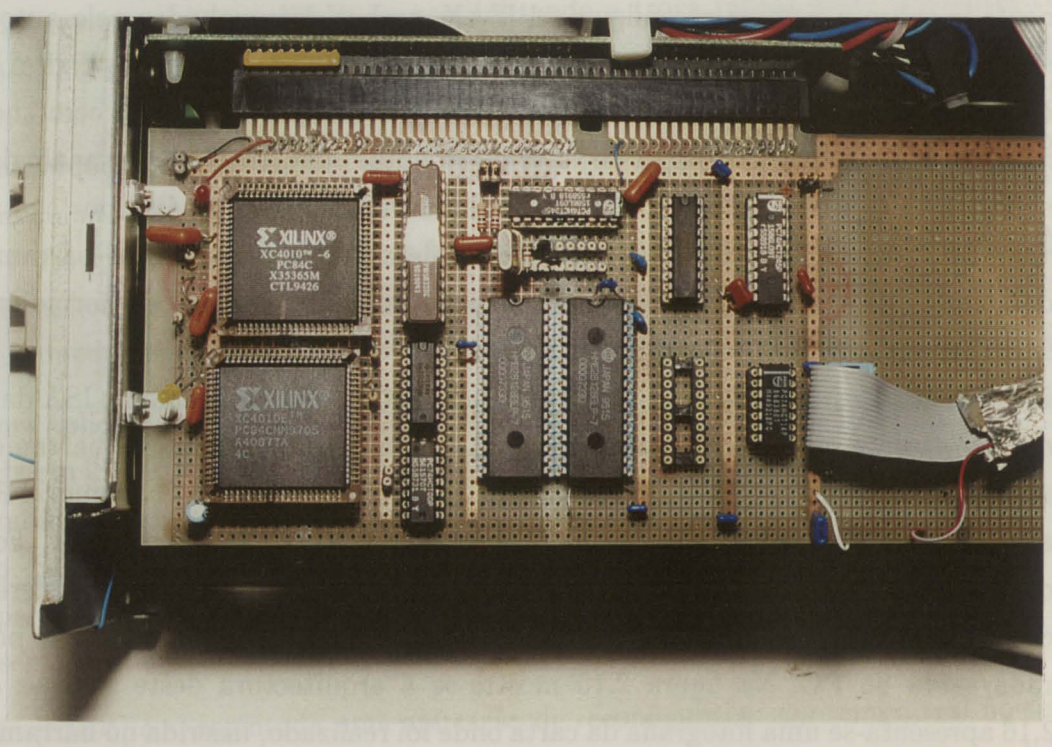


Figura 2.16: Fotografia do ISAX.

conjunto de instruções às necessidades da aplicação. A versão inicial suportava apenas instruções para controlo da infra-estrutura BST, semelhantes às do processador referido em [Fer92], o que foi inicialmente julgado suficiente para o tipo de trabalho experimental que se pretendia realizar. Foram posteriormente incluídas operações aritméticas e lógicas, saltos condicionais, acesso indexado à memória, interrupções e temporizadores

para geração de sinais de sincronização do disparo de varrimento de um osciloscópio. Estas operações adicionais foram acrescentadas e refinadas de forma a cumprir variadas solicitações resultantes da experiência adquirida com a sua utilização. A programação do processador é feita em linguagem *assembly* e a tradução para código máquina é realizada com uma configuração do tradutor de *assembly* TASM [And92].

O XBST foi implementado com as ferramentas da XILINX XACT Step 6, captura esquemática da Viewlogic e a linguagem ABEL para descrever a máquina de estados que implementa a unidade de controlo do processador. São usados 100% dos recursos configuráveis do FPGA X2 daquele sistema (XC4010E-4), sendo em X1 (XC4010-6) implementada a interface com o barramento ISA, acesso à memória por parte do PC e controlo do funcionamento do processador. A frequência de trabalho pode ser configurada entre 1.25, 2.5, 5 e 10 MHz, embora a frequência suportada pela presente versão do XBST seja limitada a 5 MHz.

2.4 Conclusões

Sistemas digitais reconfiguráveis constituem a plataforma ideal para criar processadores dedicados para aplicações específicas, que combinam a rapidez do *hardware* com a flexibilidade oferecida pela programação desse *hardware*. No estado actual da tecnologia microelectrónica, os circuitos FPGA programáveis por escrita em células de memória estática são a peça fundamental para a construção deste tipo de sistema, principalmente pela facilidade e rapidez de reconfiguração que oferecem. Apesar de vários fabricantes oferecerem circuitos deste tipo, a XILINX tem um domínio claro neste sector, reflectindo-se isso na utilização dos seus dispositivos pela grande maioria dos sistemas reconfiguráveis conhecidos, construídos em torno de FPGAs comerciais.

São conhecidas várias dezenas de sistemas reconfiguráveis, dos quais se fez um breve resumo neste capítulo, ilustrando as áreas de aplicação mais relevantes. Não há dois sistemas iguais e as arquitecturas propostas são tão variáveis quanto as complexidades apresentadas por cada um. De um modo geral, pode dizer-se que a arquitectura concebida para um sistema reconfigurável é ditada pelas características das primeiras aplicações que nela se pretendem implementar. Esta afirmação é verificada pela experiência, e o sistema reconfigurável CVR construído pelo autor e apresentado no capítulo 6 desta dissertação é disso uma confirmação.

Pode também afirmar-se que não existe um sistema reconfigurável que satisfaça de forma ideal as necessidades de qualquer aplicação. Uma consequência disso é o aumento constante do número de diferentes propostas orientadas para áreas determinadas

de aplicação, sendo a grande maioria desenvolvida em instituições académicas ou de investigação. O número de plataformas reconfiguráveis que se podem adquirir no mercado é reduzido e são mais vocacionados para prototipagem de *hardware* genérico do que para implementar processadores dedicados eficientes, destinados a acelerar processadores convencionais na execução de aplicações seleccionadas.

Apesar da crescente aceitação que os FPGAs têm vindo a conquistar na construção de processadores para aplicações específicas, provavelmente nunca tomarão o lugar dos microprocessadores e microcontroladores, no papel de circuitos programáveis para usos gerais. Uma diferença fundamental que os coloca em dois domínios bem distintos resulta daquilo que é produzido pela sua personalização e das competências exigidas para o atingir: uma sequência de operações padrão (programa) ou um circuito lógico dedicado.

O futuro da utilização de lógica reconfigurável para acelerar processadores de usos gerais sugere a necessidade de uma aproximação íntima dos domínios *hardware* e *software*. Por um lado os FPGAs tenderão a incluir elementos programáveis de maior complexidade, semelhantes aos encontrados nos processadores convencionais (FPUs, ALUs, memórias) e orientados para a realização de unidades de processamento de dados. Por outro lado, processadores de usos gerais serão complementados por plataformas reconfiguráveis com ligação próxima aos seus recursos internos, possibilitando a construção de unidades de processamento específicas que ultrapassem as limitações resultantes da utilização de circuitos integrados externos. Segundo [VMS97], dentro de 10 anos os processadores serão seguramente constituídos por uma mistura destas duas componentes, e os programas serão formados em parte por instruções a executar de forma sequencial tal como nos processadores convencionais actuais, e em parte por configurações para o *hardware* reconfigurável incluído no processador. Esta abordagem foi já sugerida por vários autores e actualmente uma equipa de investigação da Universidade de Berkeley, EUA, está a desenvolver um circuito integrado que combina um processador convencional com uma plataforma de *hardware* reconfigurável dinamicamente [HW97].

A adaptação de um sistema digital reconfigurável para uma aplicação específica requer a transformação de um algoritmo em circuitos lógicos que possam ser implementados nos dispositivos programáveis da plataforma reconfigurável. Entre um processo automático e uma implementação manual existem variadas alternativas que conduzem a diferentes compromissos entre tempo de desenvolvimento, competências exigidas ao projectista e resultados obtidos. Num extremo, a identificação e tradução automática das partes críticas de uma aplicação para circuitos lógicos, partindo de uma descrição algorítmica de um problema não permite geralmente tirar o máximo benefício da utilização de *hardware* dedicado realizado sobre um sistema reconfigurável. No entanto, possibilita que uma pla-

taforma reconfigurável possa ser usada por programadores sem conhecimentos de projecto de *hardware* nem dos detalhes do suporte de implementação utilizado. Isto será concerteza desejável para a utilização de futuros processadores que incluam uma área digital reconfigurável, onde o processo de identificação e transformação de secções do problema a realizar na área reconfigurável deverá ser parte integrante do processo de compilação de um programa. O trabalho apresentado em [Raz94] e referido na secção 2.3.3 representa um avanço neste sentido. No outro extremo, a concepção e implementação de um circuito lógico numa plataforma reconfigurável por um projectista de sistemas digitais experiente, conhecedor dos detalhes do problema e das características do sistema reconfigurável, é um processo demorado e que requer a reunião de competências em diversas áreas, mas é a forma eficaz de explorar aos limites a tecnologia alvo de implementação.

E hoje em dia bem aceite que para se atingirem bons ganhos de desempenho as partes críticas de um programa a acelerar devem ser seleccionadas pelo programador. O desenvolvimento e a estruturação de programas deve ter em vista a sua posterior implementação em *hardware* dedicado, pelo que não podem ser feitos do mesmo modo como são criados programas destinados a computadores convencionais. Uma vez escolhidas as partes de uma aplicação se deseja realizar num processador auxiliar dedicado construído sobre um sistema reconfigurável, o processo para a sua adaptação envolve várias actividades que vão desde a compilação e optimização de especificações de alto nível em que geralmente são descritos os problemas, até à concepção de circuitos lógicos e à sua realização física numa plataforma alvo elegida. Dadas as especificidades associadas a diferentes arquitecturas alvo, as técnicas empregues nestas transformações têm de ser adaptadas para as características particulares exibidas por cada sistema reconfigurável.

Nesta dissertação são abordados aspectos representativos dos tipos de problemas e considerações que se colocam na definição e utilização de processadores dedicadas em sistemas digitais reconfiguráveis, resultantes de estudos realizados em torno de aplicações concretas. Nos restantes capítulos desta tese são apresentados esses estudos, dos quais se apresenta em seguida um breve resumo.

A construção de compiladores de uma linguagem de programação de alto nível para a processadores dedicados é uma tarefa que pode ser aliviada recorrendo a compiladores configuráveis que traduzem e optimizam uma descrição algorítmica de um problema numa representação interna mais adequada às transformações subsequentes para a plataforma destino. No capítulo 3 é apresentada uma aplicação para a qual foi adaptado um compilador de linguagem C para um processador vectorial microprogramado, que foi construído

como um circuito integrado de aplicação específica para satisfazer os requisitos de processamento em tempo real de uma aplicação de controlo. Embora este processador dedicado seja realizado como um circuito lógico com uma estrutura rígida, foi conseguida flexibilidade à custa da definição da sequência de palavras de controlo (micro-instruções) que estabelecem a tarefa realizada por esse sistema. No capítulo 4 propõe-se uma arquitectura para um processador programável baseado em unidades funcionais reconfiguráveis construídas com FPGAs, que foi derivado do estudado no trabalho anterior. A consideração de unidades funcionais reconfiguráveis podendo implementar vários operadores com características de funcionamento diversas introduz, no processo de adaptação desta arquitectura para diferentes aplicações, problemas similares aos encontrados em síntese de alto nível de circuitos digitais. A implementação de uma técnica de optimização combinatória aplicada a esses problemas é apresentada no capítulo 5, sendo também generalizada para contemplar unidades funcionais com capacidade de reconfiguração dinâmica. O estudo de um problema concreto na área de processamento digital de sinal, apresentado no capítulo 6, conduziu ao desenvolvimento de um processador vectorial dedicado e de um novo sistema digital reconfigurável, vocacionado para essa aplicação. A arquitectura desse processador resultou de transformações e optimizações do algoritmo, conduzindo a uma implementação vectorial *pipelined* que, apesar de envolver apenas a realização de operações aritméticas convencionais em dados inteiros, permite um ganho em rapidez superior a 8 vezes, comparado com um PC-Pentium funcionando com uma frequência 10 vezes superior.

Capítulo 3

Suporte para programação em alto nível de um processador dedicado

3.1 Introdução

O trabalho apresentado neste capítulo tem por base um processador dedicado desenvolvido para uma aplicação de controlo de um sistema mecânico [LGW91a, LGW+91b], e aborda a construção de um compilador de uma linguagem de alto nível para essa arquitectura. O processador foi concebido e projectado no Instituto de Sistemas Microelectrónicos da Universidade Técnica de Darmstadt, Alemanha[1], e integra-se num sistema para controlo da ignição e injecção de um motor de combustão interna a gasolina, desenvolvido no âmbito de um projecto de investigação financiado pelo governo da República Federal Alemã[2]. A necessidade do desenvolvimento de um processador dedicado surgiu do elevado desempenho requerido para processar em tempo real dados recolhidos do funcionamento do motor, para actuação sobre as unidades de injecção e ignição. O autor fez parte da equipa deste projecto durante um estágio realizado em 1993 naquele Instituto, tendo desenvolvido trabalho que foi focado na adaptação de um compilador de C e na compactação de microcódigo para esse processador [Alv93, AHG94].

Este é um exemplo de um processador dedicado concebido para resolver uma classe de problemas bem definida e que foi construído como um circuito integrado de aplicação específica, programável por micro-instruções horizontais, i.e. cujos *bits* são os sinais de controlo dos elementos funcionais que integram o processador. Embora o circuito que o

[1]Institute für Datentechnik, FG Mikroelektronische Systeme, Technische Hochschule Darmstadt, Karlstr 15, D-64283 Darmstadt, Alemanha

[2]Projecto financiado pelo *Deutsche Forschungsgemeinschaft, Sonderforschungsbereich 241 "Integrated Mechanical-Electronical Systems for Mechanical Engineering"*.

implementa exiba uma arquitectura rígida e não apresente por isso programabilidade ao nível da estrutura dos seus circuitos lógicos, é conseguida flexibilidade estabelecendo a sequência de sinais de controlo que determinam a tarefa realizada. Apesar de ser por isso um processador programável por instruções, distingue-se dos processadores convencionais pela granularidade das tarefas que cada micro-instrução realiza. Na verdade, um programa para esta arquitectura é formado por uma sequência de palavras de controlo, cujos *bits* comandam directamente os registos, circuitos selectores e unidades aritméticas que o constituem.

A complexidade do processo de geração de microprogramas para este processador motivou o desenvolvimento de ferramentas de *software* para automatizar essa tarefa. Uma primeira abordagem consistiu num tradutor de uma linguagem de programação do tipo *assembly* suportando operações sobre vectores de dados, reunindo num microprograma as sequências de micro-instruções correspondentes a cada instrução *assembly*. Com o objectivo de possibilitar a programação com linguagens de alto nível, foi decidido adaptar um compilador de C configurável (gcc) para esta arquitectura e foi desenvolvido um programa para sequenciar as instruções produzidas pelo compilador, de modo a permitir compactar o microcódigo a executar pelo processador.

O trabalho que se apresenta neste capítulo, com o estudo de um caso numa área de aplicação concreta, é ilustrativo da tarefa a realizar em situações similares, em que seja necessário construir ferramentas de suporte para programação em linguagens de alto nível para arquitecturas dedicadas. O recurso a compiladores configuráveis, como é o caso do gcc, permite simplificar grande parte do trabalho de desenvolvimento de um compilador para uma nova arquitectura. No entanto, e particularizando para o compilador que foi utilizado neste trabalho, o facto do gcc ter sido concebido tendo em vista a sua adaptação para arquitecturas convencionais de processadores, apresenta limitações que não permitiram explorar completamente algumas das características específicas deste processador.

3.2 Aplicação - controlo de um motor a gasolina

O controlo do funcionamento de um motor de combustão interna a gasolina, com alimentação por carburador aspirado, é feito essencialmente à custa da determinação do instante em que deve ser activada a ignição da mistura, o que é normalmente designado como ponto de ignição ou avanço do ponto motor. Durante muito tempo esse controlo foi realizado por sistemas mecânicos e electromecânicos, baseados na velocidade de rotação do motor e no vácuo produzido no interior do colector de admissão, como medida indi-

recta da relação ar/gasolina na mistura aspirada para o interior do cilindro [CA94]. A incorporação progressiva de sistemas de ignição electrónica, conduzindo a menores consumos e redução das emissões poluentes dos gases de escape, permitiu fazer uso de outras variáveis de controlo como a posição do acelerador ou a temperatura de funcionamento do motor, por exemplo.

Hoje em dia, a maioria dos motores a gasolina que equipam os automóveis convencionais dispõem de alimentação por injecção, controlada por sistemas electrónicos que determinam o instante de tempo e a quantidade de gasolina a injectar em cada cilindro. Os sistemas de controlo electrónico calculam estes parâmetros com base na velocidade de rotação do motor, temperatura do motor e dos gases de escape, posição do acelerador, mas não entram geralmente em conta com as características termodinâmicas das combustões da mistura ar/gasolina no interior dos cilindros, nomeadamente a evolução da libertação de calor durante a combustão. Este parâmetro é relevante para a determinação das variáveis de controlo referidas acima, já que tem uma influência importante no rendimento do motor (consumo) e na qualidade de combustão da gasolina (concentração de elementos poluentes no gases de escape) [Hey88].

O trabalho que se apresenta neste capítulo teve por base um processador dedicado usado num sistema para controlo por computador de um motor de combustão interna a gasolina. O processo de controlo é baseado na caracterização termodinâmica de cada combustão, realizada à custa da variação da pressão instantânea no interior da câmara de combustão e do seu volume. Para satisfazer os requisitos de processamento em tempo real impostos pela aplicação, foi desenvolvido um processador dedicado à execução da sua parte crítica: cálculo em tempo real da evolução com o tempo do calor libertado durante cada combustão.

O sistema protótipo de controlo foi construído em torno de um computador principal (HP9000) e de uma unidade de processamento auxiliar que recolhe os dados do funcionamento do motor e calcula a função que representa o calor libertado durante cada combustão. O computador principal funciona como plataforma de desenvolvimento de *software* para a unidade de processamento auxiliar, como unidade de monitorização do funcionamento do motor e para controlo das unidades de injecção e ignição do motor (figura 3.1). A unidade de processamento auxiliar inclui um microcontrolador que implementa a interface com o computador principal e com o sistema de aquisição de dados, memórias para dados e programa e o processador vectorial dedicado que é a base do trabalho apresentado neste capítulo e constitui o núcleo de processamento onde é calculada a parte crítica referida da aplicação de controlo (figura 3.2). Este processador dedicado será referido daqui em diante por processador vectorial ou processador THD (de *Technische*

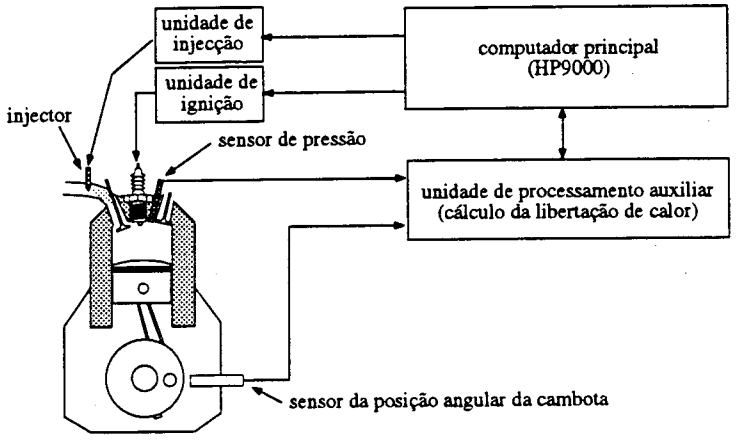


Figura 3.1: O sistema electrónico-mecânico para controlo de motores de combustão interna.

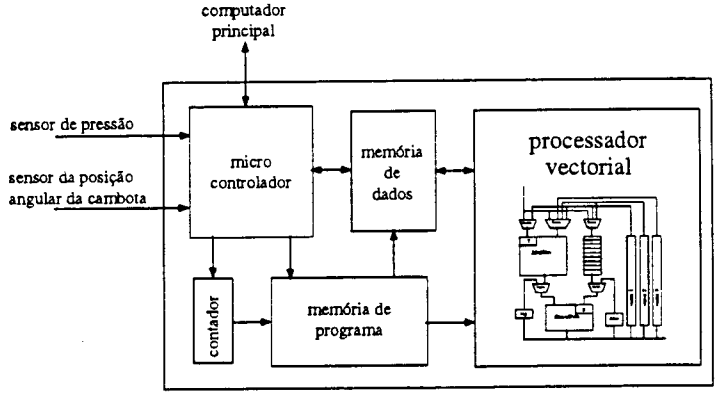


Figura 3.2: Arquitectura da unidade de processamento auxiliar.

Hochschule Darmstadt).

3.2.1 Requisitos computacionais

A caracterização termodinâmica de cada combustão é obtida de forma rigorosa se forem contabilizadas as perdas de calor através das paredes da câmara de combustão. Além disso, a equação dos gases perfeitos $P.V = N.R.T$ que modela o processo de combustão não pode ser usada nas condições do problema considerado, sem afectar a constante dos gases perfeitos R (8.314 J/mol/K) do factor de compressibilidade Z, cujo valor varia durante o processo de combustão [PC73]. Deste modo, o cálculo da libertação de calor requer a resolução de uma equação diferencial não-linear, que não podia ser resolvida, dentro das restrições de tempo real exigidas, por processadores comerciais disponíveis na altura do

desenvolvimento. Aplicações realizadas anteriormente apenas tinham conseguido calcular a libertação de calor em tempo real com algoritmos muito simples, que não entravam em conta com as características referidas.

O cálculo da libertação de calor $Q_B(\varphi)$, como uma função do ângulo da cambota do motor φ (mede de forma indirecta o volume da câmara de combustão), é realizado com base na pressão instantânea no interior da câmara de combustão $p(\varphi)$, do volume da câmara da combustão $V(\varphi)$ e dos diferenciais dessas variáveis com φ, $dp(\varphi)$ e $dV(\varphi)$. O desenvolvimento do modelo matemático completo é descrito de forma detalhada em [LGW91a] e conduz à equação diferencial que modela a libertação de calor durante uma combustão, tomando em conta as perdas de calor pelas paredes do cilindro e a variação do factor de compressibilidade Z durante a combustão.

A análise do parâmetro $Q_B(\varphi)$ durante a combustão é feita para um intervalo de 128° de rotação da cambota do motor, com intervalos de 1°, correspondente a 128 amostras da pressão $p(\varphi)$. O cálculo do calor libertado Q_B é obtido por um algoritmo iterativo que requer apenas duas iterações para atingir um erro inferior a 0.1%. O algoritmo final realiza, para cada valor de $p(\varphi)$, 232 operações aritméticas em vírgula flutuante, incluindo 14 divisões [LGW91a].

Considerando um regime máximo de funcionamento do motor de 6000 rotações por minuto, uma volta completa da cambota é realizada em 10 ms, e um grau de revolução demora aproximadamente 27 μs. Embora em cada cilindro de um motor a 4 tempos as combustões ocorram com um período de 720° da rotação da cambota (uma combustão de duas em duas voltas), a admissão da mistura é iniciada aproximadamente uma volta após o início de uma combustão. Para satisfazer os requisitos de processamento em tempo real, o cálculo do calor libertado durante a combustão deve ser feito durante um intervalo de 50° de rotação, após a aquisição das 128 amostras de pressão. Desta forma, todo o processamento é completado ao fim de meia volta da cambota, o que é necessário para ser possível repetir esse cálculo para todas as combustões que correm num motor de 4 tempos com 4 cilindros (duas por revolução). Assim, após adquirir as 128 amostras de pressão num cilindro, o tempo disponível é de 1.35 ms para executar um total de $128 \times 232 = 29696$ operações aritméticas (figura 3.3). O tempo que resta até ser iniciada a próxima admissão de mistura para o mesmo cilindro é aproximadamente $180^o \times 27\mu s/^o \approx 5$ ms, e é dispendido pelo computador principal no cálculo dos parâmetros necessários para controlar a unidade de injecção de gasolina, para o próximo ciclo no mesmo cilindro. O esforço de cálculo mínimo necessário para realizar aquelas operações em tempo real é assim de 22 MFLOPs (29696 operações/1.35 ms), o que não era atingível pelos microprocessadores com processamento em vírgula flutuante disponíveis quando foi desenvolvido o projecto.

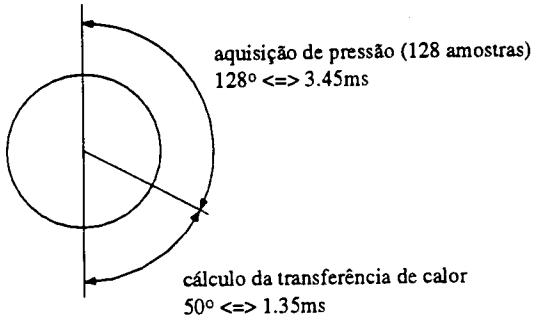


Figura 3.3: Aquisição e processamento de dados durante meia revolução da cambota.

3.2.2 Unidade de processamento auxiliar

Para além do processador vectorial, a unidade que calcula a libertação de calor é constituída por um microcontrolador que tem por função realizar a interface com os sensores de pressão e da posição angular da cambota. Para além disso, é também responsável pela comunicação com o computador principal e pelo controlo do funcionamento do processador vectorial.

O processador vectorial recebe dos sensores de um cilindro, os dados recolhidos do motor que são colocados na memória de dados partilhada com o microcontrolador, calcula a função que representa a libertação de calor e coloca os resultados na memória partilhada. O microcontrolador envia esses resultados para o computador principal que realiza algum processamento adicional e comanda as unidades de injecção e ignição para a próxima combustão nesse cilindro.

O formato para os dados foi fixado após simulações ao nível funcional, que mostraram que as gamas de valores intervenientes no processamento podiam variar várias ordens de magnitude. Optou-se por utilizar uma representação em vírgula flutuante com 20 *bits*, divididos em 12 para mantissa, 7 para expoente e 1 para sinal [LGW+91b]. Para usar uma representação em vírgula fixa conveniente seria necessário recorrer a dados de 52 *bits*. Apesar da opção tomada conduzir a unidades de cálculo mais complexas, permitiu reduzir para menos de metade a dimensão dos barramentos e com isso reduzir a área de silício necessária à implementação do processador vectorial num circuito integrado.

A análise do tipo de programa que realiza o cálculo pretendido permitiu concluir que um grande número de operações podiam ser representadas como operações em vectores de dimensão fixa, imposta pela arquitectura do sistema electrónico-mecânico (128 elementos correspondentes às 128 amostras da pressão e volume). Para além disso, as operações mais frequentes são operações combinadas do tipo multiplicação ou divisão seguidas de adição

ou subtracção, muitas delas realizadas sobre vectores de dados, pelo que resulta eficiente a utilização de operadores funcionando de forma *pipelined*. Como o tipo de programa que implementa a função a calcular pode ser traduzido numa sequência de instruções, sem saltos ou instruções condicionais, a unidade que gera os endereços para a memória de programa é constituída por um contador binário comandado pelo microcontrolador.

3.2.3 Arquitectura do processador vectorial

O processador vectorial constitui o núcleo da unidade de processamento auxiliar onde é calculada a função que representa a libertação de calor. As características do tipo de problema a resolver conduziu a uma arquitectura vectorial microprogramada que contém uma unidade de cálculo constituída por um multiplicador-divisor ligado a um somador-subtractor e 3 memórias vectoriais construídas como registos de deslocamento de 128 elementos, como se mostra na figura 3.4. O processador é controlado directamente pelos *bits* das micro-instruções que lhe são fornecidas em sequência da memória de programa, nunca sendo quebrada a execução sequencial porque não são executados saltos.

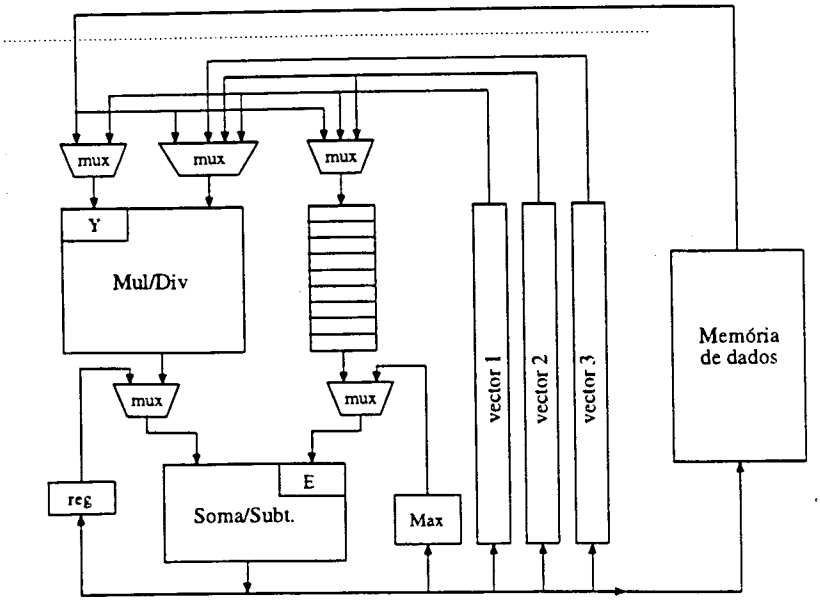


Figura 3.4: Arquitectura do processador vectorial.

Unidade de cálculo

A unidade de cálculo é composta por um multiplicador-divisor e por um somador-subtractor[3], ligados entre si da forma que se mostra na figura 3.4. As duas unidades são *pipelined* tendo respectivamente 9 e 4 estágios, com latência igual a um ciclo de relógio, e foram construídas de forma a poderem iniciar em cada ciclo de relógio uma operação de tipo diferente. Para atrasar o operando do somador que é obtido das memórias vectoriais ou da memória de dados, foi incluído um registo de deslocamento com um número de andares igual ao número de estágios de *pipeline* do multiplicador. Deste modo, os três operandos da unidade de cálculo são fornecidos no mesmo ciclo de relógio, chegando ao mesmo tempo à entrada do somador, o resultado do multiplicador e o segundo operando do somador. Os registos `Y` e `E` na entrada da unidade de cálculo permitem fixar dois dos seus operandos, o que é importante para realizar operações entre vectores e constantes.

A unidade de cálculo é por isso usada como um só operador com 13 andares de *pipeline*, realizando sempre duas operações aritméticas encadeadas. Uma vez preenchida a *pipeline*, pode ser produzido o resultado de duas operações em vírgula flutuante por cada ciclo de relógio.

Funcionando à frequência permitida de 20 MHz, o desempenho máximo possível é assim 40 MFLOPs. Convém notar que este resultado só é conseguido se em cada ciclo de relógio for consumido o resultado que aparece à saída do somador. Na aplicação considerada, as operações mais eficientes são as que processam vectores de dados de 128 elementos lidos das memórias vectoriais, gastando apenas $13+128 = 141$ ciclos de relógio. Se forem realizadas as duas operações da unidade de cálculo para produzir cada resultado, o desempenho atingido na realização de uma operação vectorial desse tipo é 36 MFLOPs. No entanto isto pressupõe que são lidos 3 operandos e é escrito um resultado no mesmo ciclo, o que apenas é possível se só um dos operandos ou o resultado residirem na memória de dados. Como no decorrer da aplicação é necessário carregar as memórias vectoriais com informação obtida da memória de dados, o desempenho global na execução de aplicações com operações vectoriais situa-se naturalmente abaixo daquele valor máximo, mas satisfaz os 22 MFLOPs requeridos pelo requisito de processamento em tempo real.

Porque a estrutura da unidade de cálculo foi optimizada para realizar operações combinadas sobre vectores de dados, o seu desempenho na execução de operações aritméticas simples sobre dados escalares é baixo. Dada a estrutura de interligação fixa entre o multiplicador e o somador, todos os resultados que a unidade de cálculo produz utilizam os

[3]Daqui em diante, o multiplicador-divisor será referido somente por multiplicador, e o somador-subtractor por somador

dois operadores. Por esse motivo, para realizar uma operação simples é necessário utilizar como um dos operandos uma constante apropriada que neutralize a operação não desejada. Por exemplo, para adicionar ou subtrair dois valores é necessário multiplicar (ou dividir) um deles por 1. De forma semelhante, multiplicar ou dividir dois números obriga a adicionar (ou subtrair) 0 ao resultado produzido pelo multiplicador. Como as constantes 0 e 1, assim como todas as variáveis escalares, são armazenadas na memória de dados, é necessário realizar 3 acessos de leitura à memória de dados para ler os operandos e uma operação de escrita para guardar o resultado. Isto traduz-se num total de 17 ciclos para realizar uma operação aritmética escalar, o que, com uma frequência de relógio de 20 MHz representa um desempenho de apenas 1.17 MFLOPs. Operações para movimento de dados entre posições de memória são também realizadas em 17 ciclos de relógio, como resultado de uma multiplicação por 1 e uma adição com 0.

Memórias vectoriais

Para além da memória de dados cujo endereço é formado por um campo da microinstrução, o processador THD contém 3 memórias vectoriais que podem ser lidas ou escritas em paralelo, sem especificar qualquer endereço. Essas memórias são construídas como registos de deslocamento com 128 estágios de 20 *bits* cada, requerendo apenas dois *bits* de controlo para o seu funcionamento. Enquanto um dos *bits* controla se é ou não realizado o deslocamento do seu conteúdo, o outro comanda um circuito selector que escolhe a origem dos dados que são carregados na sua entrada. Desta forma é possível ler os 128 elementos rodando o seu conteúdo de forma a manter os mesmo dados originais, ou ler e escrever 128 elementos em apenas 128 ciclos de relógio.

A estrutura destas memórias não permite, pelo menos de forma eficiente, usar vectores com dimensões diferentes de 128 elementos ou o acesso a posições arbitrárias no meio do vector. No entanto, isso não se traduz em qualquer inconveniente, já que para o tipo de aplicação a executar pelo processador vectorial essas memórias são apenas usadas para conter vectores de 128 elementos. Dados escalares, bem como todas as constantes usadas pelo programa, são mantidos na memória de dados partilhada com o microcontrolador.

3.3 Tradutor de *assembly* vectorial

O algoritmo que calcula a função de libertação de calor realiza um processo iterativo que opera sobre as 128 amostras de pressão e do volume da câmara de combustão. Um exemplo duma codificação desse algoritmo em linguagem C é apresentado no apêndice A.

Para conseguir um bom desempenho é necessário explorar a execução *pipelined* de operações que envolvam dados armazenados nas memórias vectoriais. Com esse objectivo foi desenvolvido um tradutor para uma linguagem *assembly* que suporta operações vectoriais [SL91]. O programa traduz operações simbólicas sobre variáveis vectoriais e escalares nas sequências de micro-instruções que conduzem à execução dessas operações. Para além de requerer a vectorização e codificação manuais do algoritmo, os segmentos de microcódigo correspondentes a cada instrução não são compactados, não sendo por isso optimizada a execução de forma *pipelined* entre operações. Apesar disso, os programas especificados nessa linguagem *assembly* e traduzidos com esse tradutor satisfazem os requisitos de execução em tempo real da aplicação, o que justifica a adopção desta forma de programação de baixo nível, obrigando à identificação manual das operações vectoriais e à sua codificação numa linguagem de baixo nível.

3.3.1 Conjunto de instruções suportadas

O tradutor suporta operações aritméticas simples e combinadas, sobre dados escalares e vectoriais. A dimensão dos vectores é fixa e imposta pelas memórias vectoriais do processador. Para além das 4 operações aritméticas simples e das 4 operações combinadas, são também suportadas instruções vectoriais específicas do tipo de aplicação a executar no processador vectorial: determinação do máximo de um vector, integração e diferenciação.

Como foi já visto antes, todas as operações aritméticas realizadas pelo processador THD são na realidade operações combinadas, compostas por multiplicações ou divisões seguidas de adições ou subtracções. Na entrada de cada operador, pode ser fixado um operando como constante, o que permite realizar as operações aritméticas simples, carregando 0 para o registo `E` do somador, ou 1 para o registo `Y` do multiplicador, consoante a operação a realizar é uma multiplicação ou divisão, ou é uma adição ou subtracção, respectivamente. Desta forma, uma operação aritmética simples é traduzida numa sequência de micro-instruções idêntica à requerida para uma operação aritmética combinada envolvendo um operando escalar, diferindo apenas o endereço da memória de dados que especifica esse operando escalar.

O resultado produzido pela unidade de cálculo pode ser carregado indistintamente para a memória de dados ou para qualquer memória vectorial. No entanto, se um dos operandos for um vector lido da memória de dados, o vector resultado só pode ser armazenado numa das memórias vectoriais, já que só é possível realizar um acesso à memória de dados em cada ciclo de relógio. Dada a estrutura da rede de circuitos selectores que alimenta a unidade de cálculo, a origem de operandos vectoriais é condicionada pela posição que

cada um ocupa na operação executada.

As instruções suportadas pelo tradutor podem ser divididas em instruções escalares e instruções vectoriais. Instruções escalares envolvem apenas variáveis escalares residentes na memória de dados; instruções vectoriais utilizam pelo menos uma variável vectorial como um dos operandos.

Instruções escalares

As instruções escalares mais simples permitem carregar um elemento da memória de dados, nos registos Y e E existentes nas entradas do multiplicador e somador, respectivamente. Cada uma destas instruções é traduzida em duas micro-instruções.

As 8 operações aritméticas escalares, bem como a operação de deslocamento de dados, são traduzidas em sequências idênticas de 17 micro-instruções, sendo apenas diferentes os endereços dos operandos e do resultado, e os *bits* que definem a operação a realizar (multiplicação ou divisão e adição ou subtracção). Como foi referido antes, as operações aritméticas simples, assim como as de deslocamento de dados, são realizadas como casos particulares das operações aritméticas combinadas, em que um ou dois operandos são as constantes 0 ou 1, obtidas também da memória de dados.

As instruções que realizam as operações aritméticas consomem todas 17 ciclos de relógio, correspondentes às 17 micro-instruções em que são traduzidas: 3 ciclos para ler os três operandos da memória de dados, 13 ciclos para propagar os dados pela *pipeline* da unidade de cálculo e 1 ciclo para escrever o resultado na memória.

Instruções vectoriais

As instruções vectoriais utilizam pelo menos um dos operandos como um vector de dados, obtido de um dos registos vectoriais ou da memória de dados. São suportadas as 8 operações aritméticas simples e combinadas considerando como operandos vectores ou vectores e escalares, deslocamento de vectores e 3 instruções que realizam tarefas específicas da aplicação: determinação do máximo, diferenciação e integração.

As instruções aritméticas vectoriais podem tomar como operandos os conteúdos das memórias vectoriais, um vector da memória de dados ou um escalar previamente carregado nos registos E ou Y. Como só pode ser feito um acesso à memória de dados em cada ciclo de relógio, o vector resultado só pode ser armazenado nessa memória se esta não for também usada como fonte de um operando. Estas operações são traduzidas em 141 micro-instruções e consomem igual número de ciclos de relógio. A instrução de deslocamento de um vector entre um registo e a memória é também implementada como um caso particular

de uma operação aritmética combinada.

Instruções vectoriais específicas da aplicação

As instruções específicas da aplicação que foram implementadas como instruções vectoriais, permitem calcular aproximações da derivada e do integral de uma função representada por um conjunto de 128 amostras e determinar o maior elemento de um vector. Estas instruções foram implementadas pelo tradutor de *assembly* porque permitem simplificar a codificação de operações deste tipo que ocorrem nos programas a executar no processador THD.

A diferenciação é realizada com 414 micro-instruções e implementa a seguinte sequência de operações, que corresponde a aproximar a derivada da função no ponto i pelo declive da recta que passa pelas amostras $i-1$ e $i+1$:

```
df[0] = f[1]-f[0]
df[i] = (f[i+1]-f[i-1])/2, i=1..126
df[127] = f[127]-f[126]
```

Esta operação é realizada em duas fases: num primeiro passo são calculadas as diferenças entre os elementos do vector operando e é armazenado temporariamente o resultado numa memória vectorial; numa segunda fase, os elementos apropriados desse resultado são divididos por 2.

A instrução que realiza a integração é implementada com 997 micro-instruções. Recebe como operando um vector contendo amostras de uma função e produz como resultado outro vector que representa o integral dessa função. Esta instrução realiza a seguinte sequência de operações que calcula uma aproximação do integral da função $df(i)$ usando a regra dos trapézios:

```
f[0] = 0
f[i] = f[i-1] + (df[i]+df[i-1])/2, i=1..127
```

Esta instrução é executada em três fases, usando também uma memória vectorial como variável temporária. Numa primeira fase é calculada a soma entre elementos adjacentes do vector origem; na segunda fase os elementos do vector calculado na primeira fase são divididos por 2; finalmente são acumulados os valores da função integral, usando para isso o registo existente na saída do somador. Como para realizar a acumulação é necessário ter disponível o resultado de uma adição antes de iniciar a próxima soma, não é possível utilizar o somador de forma *pipelined*. Por este motivo, o cálculo final do integral necessita

de 5 ciclos de relógio para produzir cada elemento do vector resultado, gastando no total 646 micro-instruções.

Foi também implementada uma instrução para determinar o máximo elemento de um vector. Esta operação utiliza o subtractor e a unidade MAX, que é constituída por um registo de deslocamento controlado pelo *bit* de sinal do resultado produzido pela unidade de cálculo, e que é carregada com o máximo valor encontrado. Como para comparar um elemento do vector com o máximo valor encontrado é necessário dispor do resultado de uma subtracção (que implementa a comparação), também não é possível utilizar o subtractor de forma *pipelined.* Por esta razão, o processamento de cada elemento do vector gasta 5 ciclos de relógio e esta instrução é traduzida num total de 659 micro-instruções.

Um exemplo em *assembly* com operações vectoriais

O algoritmo para cálculo da libertação de calor apresentado no apêndice A em linguagem C, foi codificado manualmente na linguagem *assembly* vectorial suportada por este tradutor. Considerando apenas uma chamada à função `iteration()`, o programa é traduzido em 91 instruções, das quais 71 vectoriais e 20 escalares, correspondendo a um total de 12400 micro-instruções. A título de exemplo, mostra-se em seguida uma sequência de 4 instruções desse programa, contendo operações com dados vectoriais e escalares que realizam as operações representadas pelos comentários, incluídos à direita do asterisco. Os operandos `vr1` e `vr2` representam as memórias vectoriais, e todos os outros operandos são variáveis residentes na memória de dados.

```
divadd(vr1,1E7,A;vr1)          * vr1 = vr1 / 1E7 + A
muladd(vr1,9.31,269.99;vr1)    * vr1 = vr1 * 9.31 + 269.99
add(M_LUFT,M_RESTGAS;buf)      * buf = M_LUFT + M_RESTGAS
muladd(Su,M_KRAFT,buf;vr2)     * vr2 = Su * M_KRAFT + buf
```

3.4 Adaptação de um compilador de C

Com o objectivo de tornar praticável a avaliação de diferentes algoritmos sem obrigar à codificação manual naquela linguagem *assembly*, foi criado um compilador de C para permitir programar este processador dedicado com uma linguagem de programação de alto nível.

Para isso, foi adaptado o compilador de C da GNU (gcc) para produzir código *assembly* compatível com as operações suportadas pelo processador vectorial [Alv93]. Foi posteriormente desenvolvido um programa para sequenciar as instruções produzidas pelo compilador, de forma a permitir compactar o microcódigo a executar pelo processador [AHG94].

O gcc é talvez o compilador de C mais popular para sistemas operativos UNIX e outros sistemas derivados do UNIX. A razão disso é bem conhecida: o programa é desenvolvido sob os auspícios da organização norte-americana *Free Software Foundation*, é distribuído gratuitamente incluindo o código fonte, e pode ser configurado para a generalidade dos processadores convencionais que equipam os mini e microcomputadores actuais.

O facto deste compilador ser configurável para uma gama alargada de processadores, sugeriu a sua utilização para compilar programas escritos em linguagem C para uma linguagem de baixo nível que pudesse ser posteriormente traduzida em microcódigo para o processador vectorial. Admitia-se que, dadas as características particulares do processador considerado, fosse necessário impor restrições ao tipo de construções da linguagem C suportadas no código fonte. No entanto, isso não deveria constituir qualquer limitação para o tipo de programa a que o processador se destina.

A configuração do compilador é realizada construindo um conjunto de ficheiros, onde são especificadas as características do processador destino, regras a empregar nalguns passos de optimização, e a sintaxe da linguagem *assembly* gerada pelo compilador [Sta92]. A distribuição do compilador inclui já ficheiros de configuração para os processadores mais populares. Configurar o gcc para esses processadores é um processo automático, executando um ficheiro de comandos que adapta alguns ficheiros fonte do compilador, de forma a incluir a informação específica do processador destino considerado.

O trabalho de configuração do gcc para o processador vectorial foi realizado com a versão 2.3.3 do compilador. A utilização desta versão justificou-se na altura deste desenvolvimento, por ser uma versão estável e pela disponibilidade da documentação descrevendo o processo de configuração [Sta92].

3.4.1 Funcionamento do GNU C

O gcc realiza a compilação em várias fases, que são relativamente independentes entre si e que seguem a sequência de procedimentos padrão empregues pela generalidade dos compiladores de linguagens de programação de alto nível [ASU86]. O código fonte é inicialmente lido e traduzido para estruturas de dados em memória que representam as suas instruções. Sobre essas estruturas de dados são realizadas várias optimizações e finalmente é produzido o código *assembly*, que é posteriormente traduzido para código

"objecto"[4]. É compilada uma função de cada vez, e a representação interna dessa função é libertada após a sua compilação, excepto se tiver sido declarada de modo a que o seu código seja incluído noutras funções que a invoquem.

Os passos do compilador que se descrevem em seguida são realizados para obter o grau máximo de optimização e podem ser activados ou não, dependendo de opções especificadas quando é invocado o compilador. Em cada passo é executada uma tarefa específica de optimização que se refere a seguir, identificando-se também o nome do ficheiro fonte do compilador que contém as funções que intervêm nessa tarefa.

- *Parsing*
  Este é o primeiro passo executado na compilação de um ficheiro. O texto do código fonte é lido instrução a instrução, é analisada a sintaxe da instrução e é criada uma estrutura em memória que representa cada operação elementar contida nessa instrução. A estrutura de dados construída em memória é baseada em listas com uma estrutura semelhante à da linguagem de programação Lisp, designada RTL—*Register Transfer Language*. Esta estrutura de dados é utilizada por todos os procedimentos de optimização do compilador, até à geração do código *assembly* final. Nesta tarefa intervêm funções existentes em vários ficheiros fonte, dependendo das linguagens para as quais o compilador é construído.

- Optimização de saltos (`jump.c`)
  Neste passo são simplificados alguns tipos de saltos, eliminando instruções que nunca são executadas e etiquetas (*labels*) que não são referidas.

- Eliminação de expressões comuns (`cse.c`, *common statement elimination*)
  Neste passo são identificadas e eliminadas expressões comuns. Sequências similares de operações repetidas em locais diferentes são simplificadas numa só. São também propagadas constantes e realizadas simplificações que resultem da utilização de constantes em expressões. Se instruções de salto condicionais ficam incondicionais após a simplificação de expressões, então o passo anterior é invocado novamente de forma a eliminar as instruções que deixam de ser executadas. Algumas simplificações de expressões algébricas são também realizadas nesta fase. Um exemplo é a substituição de multiplicações por constantes em combinações de instruções de deslocamento de *bits* e adições.

[4]Tradução à letra do termo inglês *object code*, usado para denominar os ficheiros produzidos por um compilador, contendo o código máquina e informação dos símbolos e das referências a símbolos.

- Optimização de ciclos (`loop.c`, `unroll.c`)
  Neste passo são optimizados ciclos, deslocando para fora do ciclo instruções cujo resultado não dependa das iterações do ciclo. Se o conjunto de instruções do ciclo satisfizerem determinados critérios, por exemplo número de instruções no ciclo inferior a um limite especificado ou número de iterações conhecido, o ciclo é expandido (desenrolado). Após este passo, a tarefa anterior (`cse.c`) pode ser ou não repetida, dependendo do grau de optimização seleccionado e do tipo de optimizações efectivamente realizadas durante esta fase.

- Análise do fluxo de dados (`flow.c`)
  Nesta fase, as instruções que compõem o programa são agrupadas em blocos básicos (conjuntos de instruções sem saltos, ciclos ou chamadas a funções) e são eliminadas instruções que produzem resultados que nunca são usados.

- Combinação de instruções (`combine.c`)
  Esta fase tenta juntar sequências de duas ou três instruções relacionadas entre si por dependências de dados, em instruções combinadas que sejam suportadas pelo processador destino. Durante este passo, são efectuadas simplificações algébricas adicionais, com o objectivo de tentar construir operações combinadas.

- Sequenciamento de instruções (`sched.c`)
  Este passo tenta juntar sequências de operações que não apresentem dependências de dados entre si, reordenando as instruções de forma a optimizar o funcionamento *pipelined* das unidades aritméticas e lógicas do processador destino.

- Atribuição de registos (`local-alloc.c`, `global.c`)
  O primeiro passo atribui registos reais do processador às variáveis temporárias que apenas existem durante um bloco básico. Em seguida, essa tarefa é completada para as variáveis que são utilizadas para além das fronteiras de blocos básicos. As variáveis temporárias que não forem atribuídas a registos reais do processador são atribuídas a registos fictícios.

- *Reloading* (`reload.c`, `reload1.c`)
  Nesta fase são atribuídas posições de memória para as variáveis temporárias que no passo anterior foram colocadas em registos fictícios. Esta tarefa pode acrescentar instruções de movimentação de dados, de forma a colocar variáveis nos tipos apropriados de registos reais do processador, esperados pelas instruções que as utilizam. As variáveis temporárias que não tenham sido até então colocadas em registos do processador, são atribuídas a variáveis criadas na pilha do processador.

- Neste ponto é repetido o sequenciamento de instruções e a optimização de saltos (`sched.c` e `jump.c`).

- Fim (`final.c, insn-output.c`)
  Este é o último passo do compilador. São realizadas algumas optimizações locais, substituindo sequências de instruções RTL por sequências de instruções *assembly* equivalentes, e é emitido o código *assembly* final da função compilada.

No final de cada um destes passos de optimização, é possível produzir uma listagem do conjunto de instruções existentes nessa fase, num formato textual utilizado para representar as estruturas internas usadas pelo `gcc`. Estas estruturas são chamadas expressões RTL (ou *rtx*) e são listas cujo primeiro elemento é um código que identifica o tipo de objecto representado. Existem códigos para representar vários tipos de entidades, como constantes, referências a memória ou operações aritméticas. O código de uma lista define os tipos de elementos que a lista pode ter; cada elemento pode, por sua vez, ser uma lista ou uma constante. Cada código tem associado o chamado "modo máquina" (*machine mode*), que identifica o tipo de dado que a expressão representa, e que é listado, na representação textual, por um código de dois caracteres (por exemplo `SI` para *Single Integer* e `SF` para *Single Float*). As instruções da função em compilação são representadas por uma lista ligada de expressões RTL com código *insn* (*instruction node*).

Como exemplo, a referência a um elemento de um vector do tipo `float`, escrito em C como a expressão `X[4]`, é representada pela seguinte expressão RTL[5]:

```
(mem:SF (const:SI (plus:SI
                    (symbol_ref:SI ("X") )
                    (const_int 16)
) ) )
```

Esta expressão RTL pode ser lida como "uma referência a um dado em memória do tipo `SF` (`(mem:SF...)`), cujo endereço é uma expressão inteira constante (`(const:SI...)`), obtida como o resultado da adição (`(plus:SI...)`) de um inteiro representado pelo símbolo `X` (`(symbol_ref:SI ("X") )`) com a constante 16 (`(const_int 16)`)". Esta leitura é semelhante ao modo como pode ser lida uma escrita alternativa daquela expressão em C, `*(X+4)`: o elemento cujo endereço é `X` mais 4 vezes o espaço ocupado por cada elemento de `X`.

[5]Neste exemplo considera-se que cada variável do tipo `float` gasta 4 endereços de memória.

3.4.2 Configuração do GNU C

A configuração do gcc é realizada construindo um ficheiro principal de configuração e ficheiros adicionais em linguagem C que contêm funções e definições de *macros*.

No ficheiro principal de configuração (*machine description*–`*.md`) representam-se modelos das instruções que o processador destino suporta (chamados "padrões de instrução"). Cada modelo define a operação realizada pela instrução, o tipo de operandos permitidos e o código *assembly* a emitir quando essa instrução é encontrada na lista de *rtx* que representa a função compilada.

Para além da geração do código *assembly*, os padrões de instrução presentes neste ficheiro são também usados para construir as expressões RTL durante a leitura e interpretação do código fonte. Alguns padrões de instrução têm que existir sempre para definir o modo como devem ser criadas as expressões RTL para certos tipos de instruções, mesmo que nunca apareçam no código fonte a compilar. Exemplos são os que representam as instruções de salto condicional e incondicional e chamadas a sub-rotinas. No ficheiro `*.md` são também definidos padrões para especificar as instruções combinadas que o processador destino suporta como instruções individuais, e regras para realizar algumas optimizações locais durante a fase final de emissão do código *assembly*.

Noutro ficheiro de configuração (`*.h`) são criados *macros* que definem características adicionais do processador destino (*target description macros*). Entre outras, são representadas as dimensões dos diferentes tipos de dados, nomes de registos e sua função, os custos relativos das operações e modos de endereçamento. São também definidos os formatos a usar na emissão do código *assembly* para os diversos tipos de operandos e modos de endereçamento suportados pela arquitectura destino.

Uma configuração do gcc pode também incluir ficheiros auxiliares em C, contendo funções requeridas pelas construções contidas nos dois ficheiros anteriores.

Padrões de instrução

Os modelos das instruções que o processador suporta são representados no ficheiro `*.md` por padrões de instrução, que são expressões RTL incompletas, com código `define_insn` (designadas *insn_template*). Em lugar dos operandos, são especificadas regras que definem os tipos de operandos que uma instrução deve ter para ser correspondida por um padrão de instrução. Durante a fase de geração do código *assembly*, as *insn* que representam a função compilada são comparadas com os padrões que representam as instruções suportadas pelo processador. Quando é encontrada uma correspondência é emitido o código *assembly*, de acordo com regras especificadas nesse padrão de instrução.

Para ilustrar os componentes de um padrão de instrução, considere-se o exemplo seguinte que representa uma operação de adição de inteiros "naturais" da máquina (tipo SI):

```
(define_insn "addsi3"
    [ ( set (match_operand:SI 0 "nonimmediate_operand" "=mr")
            (plus:SI (match_operand:SI 1 "register_operand" "r")
                     (match_operand:SI 2 "register_operand" "r")
            )
      ) ]
      ""
      "ADD %0,%1,%2"
)
```

O código `define_insn` da expressão RTL define um padrão de instrução. O nome que segue é opcional, embora os padrões para algumas operações tenham de usar nomes pré-definidos, porque são usados também na fase de construção da representação RTL. O nome pré-definido `addsi3` representa uma instrução de adição de inteiros com 3 variáveis (resultado e dois operandos). O elemento seguinte (chamado *template RTL*) é um vector de expressões RTL, delimitadas por parêntesis rectos. No exemplo apresentado esse vector contém uma só expressão com código `set` que representa uma atribuição; o primeiro operando é a variável a quem é atribuído o valor (*lvalue* ou *left-value*), e o segundo representa a expressão cujo valor é atribuído (expressão *RHS*, de *Right Hand Side*). O lugar de cada operando é especificado por uma expressão RTL com código `match_operand`, com um modo-máquina que define o tipo de elemento que esse operando representa. Os elementos seguintes dessa expressão são a posição do operando (0 é o primeiro), o predicado e a restrição de operando[6].

O predicado é o nome de uma função que é invocada quando é tentada a correspondência de um operando de uma *insn* com a posição correspondente no padrão de instrução, e que serve para aceitar ou rejeitar um tipo de operando. No exemplo apresentado, `nonimmediate_operand` é o nome de uma função que apenas aceita operandos não imediatos (do tipo *lvalue*, a quem possa ser atribuído um valor). No ficheiro fonte do compilador `recog.c` existem várias funções para identificar tipos de operandos variados, e podem ser construídas funções adicionais que definam outros predicados.

A restrição de operando é um código que especifica o tipo de elemento de memória onde esse operando deve residir para ser aceite no padrão de instrução, durante a fase

[6]Os dois últimos termos são a tradução à letras das designações originais para estes campos: *predicate* e *operand constraint*.

de atribuição de registos do processador e geração do código *assembly*. Neste exemplo, `r` representa um registo do processador, e `=mr` significa um operando em memória ou num registo, a quem possa ser atribuído um valor; a restrição `r` dos operandos 1 e 2 é redundante, porque o predicado `register_operand` apenas permite operandos que sejam registos. Num mesmo padrão podem ser especificados vários conjuntos de restrições para cada operando, de forma a diferenciar o código *assembly* a emitir em função do tipo de operando (por exemplo, uma instrução pode ter diferentes códigos consoante os seus operandos são registos, referências a memória ou valores imediatos). Durante a fase de *reload*, as *insn* que representam a função compilada são comparadas com os padrões de instrução, de forma a verificar se são ou não satisfeitas as restrições de cada operando. Se necessário, são acrescentadas instruções adicionais, de forma a mover os operandos que não satisfaçam essas restrições para os elementos de memória adequados.

O campo seguinte (vazio no exemplo apresentado) pode conter uma expressão condicional escrita em C, que tem de ser avaliada como verdadeira para uma *insn* ser correspondida pelo padrão de instrução.

Finalmente, no último campo é especificado o código *assembly* a emitir quando esse padrão é correspondido. A forma mais simples, como é mostrado no exemplo anterior, consiste numa cadeia de caracteres onde se representa o código da instrução *assembly*, especificando a posição dos operandos com %0, %1 e %2. Em alternativa, pode ser usado um segmento de código C para emitir o código *assembly* de uma instrução. No contexto em que esse código é compilado, tem acesso a um conjunto de funções, estruturas de dados e *macros* para manipular as expressões RTL que representam a instrução. Mostra-se em seguida um padrão de instrução semelhante ao anterior, onde é usado um bloco de código C que permite distinguir dois códigos de instrução diferentes para a mesma operação de adição, em função de serem ou não iguais o primeiro e o segundo operando:

```
(define_insn "addsi3"
  [ ( set (match_operand:SI 0 "nonimmediate_operand" "=mr")
          (plus:SI (match_operand:SI 1 "register_operand" "r")
                   (match_operand:SI 2 "register_operand" "r")
          )
    ) ]
    ""
    "*
    { if ( REGNO(operands[0]) == REGNO(operands[1]) )
         return \"ADA %0, %2\";
      else
```

```
        return \"ADD %0, %1, %2\";
    }"
)
```

Combinação de instruções

Num dos passos do compilador (`combine.c`) é tentada a combinação de operações relacionadas entre si por dependências de dados, em instruções suportadas pelo processador. A especificação das combinações possíveis é feita criando padrões de que representem as sequências de operações admissíveis. O mesmo padrão de instrução é usado posteriormente para a emissão de código *assembly*.

O exemplo seguinte mostra o padrão de instrução usado na configuração construída para o processador THD, para combinar sequências do tipo `y=a*b; z=y+c` numa única operação combinada `z=a*b+c`. A função `output_madd()` é responsável pela emissão do código *assembly* em que é traduzida esta instrução.

```
(define_insn ""
  [(set (match_operand:SF 0 "general_operand" "")
        (plus:SF (match_operand:SF 3 "general_operand" "")
                 (mult:SF (match_operand:SF 1 "general_operand" "")
                          (match_operand:SF 2 "general_operand" ""))
                 )
        )
   )]
  ""
  "*
  { output_madd(operands, insn);
    return \"\";
  }")
```

Expansão e separação de *insn*

No ficheiro de configuração podem ainda existir dois tipos de padrões de instrução, cuja função é a inversa da referida antes: expansão ou separação de uma instrução em várias *insn*.

Os padrões criados com o código `define_expand` permitem definir uma sequência de *insn* em que um dos padrões pré-definidos deve ser expandido durante a fase de leitura e interpretação do código fonte para construir a representação interna RTL. Por exemplo,

uma operação de adição de 32 *bits* num processador de 16 *bits* pode requerer a realização de duas adições de 16 *bits*.

O código `define_split` define outro tipo de padrão que permite dividir uma *insn*. De forma semelhante ao anterior, é definida uma sequência de instruções individuais em que uma *insn* pode ser expandida, mas estes padrões são utilizados durante a fase de optimização, quando são tentadas combinações de operações e simplificações de expressões (`combine.c`).

Optimizações locais

Na fase final de geração de código *assembly* é realizado um último passo de optimização, seguindo regras especificadas no ficheiro de configuração. Esse tipo de optimização local (*peephole optimization*) é definido com padrões de instrução que contêm sequências de *insn* e o código *assembly* a emitir quando essa sequência é encontrada no conjunto de instruções que representam a função compilada. Uma optimização deste tipo permite, por exemplo, transformar a sequência de operações `R1=R2; R1=X[R1]` em `R1=X[R2]`.

Definição de *macros*

Para além do ficheiro `*.md`, outro ficheiro de configuração contém um conjunto de definições de *macros* que representam características variadas do processador destino: custos relativos de instruções, número, nome e função dos registos, dimensão dos tipos de dados e segmentos de código C para formatar os operandos das instruções no ficheiro de *assembly* produzido. Podem ser criados mais de 400 *macros* neste ficheiro, embora muitos deles sejam definidos com valores por omissão, dispersos pelos vários ficheiros fonte do compilador.

Um conjunto de definições especifica a forma como os dados que o processador manipula são armazenados em memória: número de *bits* em cada unidade endereçável de memória (*byte*), alinhamento de *bytes*, dimensão dos tipos dos dados declarados em C e formato de dados em vírgula flutuante, entre outros. O tipo "natural" do processador é definido como uma *word*, e as dimensões dos tipos de dados de C são definidas como múltiplos inteiros de *words* (excepto dados do tipo `char` que são sempre metade de uma *word*).

Outro conjunto de *macros* define as características dos registos do processador destino: número, nome, tipo de operações a que está dedicado (apontador de pilha ou apontador de programa, por exemplo) e ordem de prioridade a usar na fase de atribuição de variáveis temporárias a registos. São também especificados os tipos de dados que cada registo pode

armazenar, bem como classes que agrupam registos em função do tipo de operações em que podem ser usados. Isto permite classificar os registos do processador destino, facilitando a construção de restrições de operandos para os padrões de instrução (como exemplo, em certas arquitecturas só alguns registos podem ser usados para endereçamento indexado a memória).

Os custos relativos das instruções, constantes e modos de endereçamento são representados por outro conjunto de definições. Esses custos são usados para quantificar os resultados conseguidos pelas simplificações realizadas nas várias fases de optimização.

A especificação da forma como é emitido o código *assembly* é feita principalmente no ficheiro `*.md` ou em funções em C incluídas em ficheiros adicionais de configuração. Para operandos, constantes e etiquetas (*labels*) devem ser criados *macros* que estabelecem o formato para aqueles elementos no ficheiro de *assembly*. Outro conjunto de definições especifica as sequências de instruções a executar na entrada e na saída de uma função, por exemplo guardar e recuperar da pilha os registos que a função utiliza ou reservar espaço na pilha para variáveis temporárias.

O grande número de *macros* que podem ser definidos neste ficheiro e certas interdependências que são assumidas entre algumas definições torna difícil a construção de raiz de um ficheiro deste tipo. A violação de certas regras (por exemplo, definir um *byte* ou *word* com um número de *bits* que não seja uma potência inteira de dois) faz com que o compilador falhe de forma catastrófica, obrigando a processos morosos de detecção e depuração de erros para identificar a definição responsável por essa falha. Por este motivo, a configuração que se construiu para o processador THD tomou como base o conjunto de ficheiros de configuração já existente para um processador suportado na distribuição do `gcc` (AT&T WE32000, com os ficheiros de configuração `we32k.*`).

3.4.3 Configuração para o processador THD

Com o objectivo de construir um compilador de C para o processador THD, foram estudadas as capacidades de adaptação do `gcc` e foi construída uma configuração para gerar código *assembly* contendo operações suportadas pelo processador dedicado. Foram encontradas várias dificuldades, resultantes de diferenças importantes entre esta arquitectura e o tipo de processador convencional que o `gcc` considera como alvo. Com efeito, algumas características do processador alvo são consideradas de forma rígida pelo `gcc`, e são difíceis ou mesmo impossíveis de ultrapassar apenas com as construções de configuração oferecidas, sem modificar o código fonte do compilador.

Limitações do modo de configuração do GNU C

Um exemplo do tipo de diferenças referidas diz respeito ao facto de o gcc considerar sempre que o processador destino suporta instruções para manipular a pilha em memória e o registo apontador de pilha, como acontece na generalidade dos processadores convencionais. Uma das utilizações correntes dessa estrutura de memória serve para criar variáveis temporárias utilizadas localmente a uma função. O gcc considera a existência dessa estrutura de memória e cria automaticamente variáveis temporárias sempre que são excedidos os registos disponíveis do processador. Isto pode ser contornado se for definido um conjunto com um número suficiente de registos, que permita conter todas as variáveis temporárias criadas durante a compilação de uma função, sendo posteriormente traduzidos em referências a variáveis em memória aqueles que não correspondam a registos reais do processador.

Outra diferença importante entre o processador THD e um processador convencional resulta do facto de o primeiro não permitir carregar o registo apontador de programa para executar saltos. Para o tipo de programa que se pretende executar nesse processador o resultado da compilação pode ser descrito por um único bloco básico de instruções que não contém instruções de salto. No entanto, a configuração do gcc obriga a definir um registo como apontador de programa e a incluir no ficheiro de configuração os padrões de instrução que representam instruções de salto, mesmo que elas não ocorram no código fonte a compilar, ou que sejam eliminadas como resultado das várias tarefas de optimização.

O compilador assume que os processadores destino suportam instruções de deslocamento de *bits*, e que estas são sempre mais eficientes do que multiplicações e divisões. Num dos passos de optimização (`cse.c`), multiplicações ou divisões inteiras por constantes iguais a potências inteiras de dois são substituídas por instruções de deslocamento de *bits* equivalentes, independentemente dos custos relativos que lhes tenham sido atribuídos. Como o processador alvo apenas realiza operações sobre dados em vírgula flutuante, estas simplificações nunca são realizadas.

Uma característica do processador vectorial que não foi possível modelar relaciona-se com a utilização das memórias vectoriais. Como o acesso a essas memórias só pode ser feito da forma sequencial descrita atrás, não é possível considerá-las como bancos de registos que permitam ler ou escrever de forma directa um elemento, sem obrigar ao deslocamento sequencial de todo o seu conteúdo. As construções para configuração suportadas pelo compilador não permitem modelar esse modo de funcionamento e por este motivo não foi possível considerar a utilização das memórias vectoriais. Esta opção surgiu como uma limitação importante do compilador construído, já que a utilização daquelas memórias é fundamental para o funcionamento vectorial do processador, sendo a chave

para o elevado desempenho conseguido com a codificação manual na linguagem *assembly* vectorial apresentada antes. Por esta razão, o código *assembly* produzido pelo compilador contém apenas instruções que executam operações escalares, com dados e resultados residentes na única memória de dados. Como o processador só tem um barramento para acesso a essa memória, a execução de cada instrução requer tantos acessos à memória quantos os operandos consumidos e os resultados produzidos. Isto condiciona inexoravelmente a rapidez que pode ser conseguida executando o código produzido pelo `gcc`, não sendo possível, pelas razões apresentadas, competir com o desempenho de codificações manuais que explorem a utilização daquelas memórias.

Apesar das limitações referidas, o compilador construído permite traduzir automaticamente um algoritmo descrito numa linguagem de programação de alto nível em instruções suportadas pelo processador THD, sem obrigar à sua codificação manual em linguagem *assembly*. Isto veio possibilitar a comparação e avaliação dos resultados conseguidos por diferentes abordagens para aquela aplicação de controlo, antes de proceder à optimização e codificação manuais, que mostraram ser necessárias para satisfazer os requisitos de tempo real da aplicação.

Especificação da arquitectura e do conjunto de instruções

Uma parte importante da especificação da arquitectura do processador destino define o conjunto de registos suportados, dimensões e funções especiais associadas a cada um. Esta parte da configuração é feita no ficheiro de descrição de *macros*.

Para evitar a criação automática de variáveis temporárias na pilha, foi definido um conjunto de 64 registos de 32 *bits*, alguns dos quais são reservados a funções específicas requeridas pelo `gcc` (apontador de programa e apontador de pilha, entre outras). Excepto os registos `Y` e `E` considerados por um de dois conjuntos de instruções experimentados, os restantes não têm existência real no processador e devem ser posteriormente traduzidos em referências à memória de dados, durante a fase de tradução do código *assembly* para microcódigo.

Como nos programas que se pretende compilar para esta arquitectura todas as variáveis envolvidas são do mesmo tipo e representam números em vírgula flutuante com 20 *bits* (12 para mantissa, 7 para expoente e 1 para sinal), este deveria ser o formato estabelecido para representar dados em vírgula flutuante considerados pelo compilador. Tal não é possível, porque a dimensão para todos os tipos de dados que o `gcc` considera deve ser igual a múltiplos inteiros da unidade mínima endereçável, e esta só pode ser definida com 8 ou 16 *bits*. Por esta razão, foram definidos os tipos `float` e `double` com 32 *bits*, ocupando assim 4 endereços de memória consecutivos. Os tipos `int` e `long` foram também definidos

com 32 *bits*, embora não sejam suportados pelas instruções *assembly* consideradas para o processador. Deste modo, qualquer registo de uso geral pode conter qualquer um dos tipos de dados permitidos no programa fonte, `int` ou `float`, o que permite simplificar os predicados e as restrições de operandos usados na construção dos padrões de instrução.

A construção de uma configuração requer a especificação do conjunto de instruções suportado pelo processador destino. Isso é feito principalmente com os padrões de instrução definidos no ficheiro de descrição da máquina (`*.md`). O `gcc` produz código *assembly* que tem de ser posteriormente convertido no microcódigo executável pelo processador THD. Essa operação pode ser feita pelo tradutor de *assembly* referido na secção 3.3, embora esse programa não realize qualquer compactação do microcódigo que produz. Como complemento ao compilador construído, foi desenvolvido o programa que se apresenta na secção 3.4.4 e que permite atribuir a ciclos de relógio as instruções *assembly* produzidas, de forma a compactar o microcódigo em que são traduzidas.

No sentido de avaliar os resultados das simplificações realizadas pelos vários passos de optimização, foram experimentadas configurações que consideram 2 conjuntos de instruções diferentes, referidos nas secções seguintes por conjunto de instruções 1 e conjunto de instruções 2. O primeiro considera que os registos à entrada do multiplicador e do somador podem ser carregados por instruções apropriadas, o que permite evitar a repetição dessa operação para sequências de valores iguais. Por exemplo, para executar uma sequência de adições basta colocar uma vez a constante 1 no registo `Y` à entrada do multiplicador, não sendo necessário repetir essa operação para as restantes adições realizadas. O segundo conjunto de instruções vê esses registos como parte integrante da unidade de cálculo, de forma que todas as instruções consideradas correspondem a operações combinadas consumindo sempre três operandos da memória de dados.

Para além das instruções *assembly* produzidas como resultado da compilação e correspondentes às operações aritméticas simples e combinadas realizadas pela unidade de cálculo, foi também necessário incluir no ficheiro de configuração certos padrões de instrução exigidos pelo `gcc` para vários tipos de operações, mesmo que nunca apareçam no código fonte a compilar nem no código resultante da compilação. É o caso dos saltos condicionais e incondicionais, chamadas a sub-rotinas, comparações e operações aritméticas com inteiros. Se essas operações forem encontradas durante a fase de geração do código *assembly*, é emitida uma mensagem de erro apropriada.

Conjunto de instruções 1

No primeiro conjunto de instruções experimentado, as operações suportadas pelo processador vectorial resumem-se às operações aritméticas combinadas, sendo as operações

aritméticas simples e de deslocamento de dados realizadas como casos particulares daquelas, colocando as constantes 0 ou 1 nos registos `Y` ou `E`. Estes registos são vistos como registos de dados especiais e as variáveis que armazenam apenas podem ser usadas como operandos das operações aritméticas, em posições determinadas das operações. Isto permite evitar a repetição do carregamento desses registos, quando são executadas sequências de operações em que os seus conteúdos são mantidos constantes.

A consideração deste conjunto de instruções permite fazer uso da fase de optimização do compilador em que são atribuídos os registos do processador às variáveis. Definindo aqueles dois registos como pertencendo a classes diferentes, pode-se especificar nos padrões de instrução que os operandos das operações aritméticas têm de residir nos registos apropriados. Construindo também os padrões de instrução que representam as operações de deslocamento de dados da memória para esses registos, durante as fases de atribuição de variáveis a registos e de *reload*, são acrescentadas as instruções necessárias para mover os dados para os registos `E` ou `Y` apropriados.

Como não existe nenhum registo à saída do multiplicador onde possa ser mantido o seu resultado todas as operações aritméticas têm de ser traduzidas em operações combinadas, envolvendo sempre os dois operadores da unidade de cálculo e necessitando de três operandos: dois colocados previamente nos registos `E` e `Y` e o terceiro obtido da memória de dados. Para o funcionamento correcto desta configuração, na fase de optimização em que são tentadas as combinações de instruções (`combine.c`) deveria ser possível conseguir agrupar todas as operações aritméticas em operações combinadas. Isto deveria acontecer mesmo para as operações aritméticas simples, sendo nesse caso carregadas as constantes 0 ou 1 apropriadas para os registos `Y` ou `E`. No entanto, devido à forma como o compilador realiza as várias fases de optimização, não é possível garantir que sejam apenas produzidas instruções que correspondam a operações combinadas.

Durante a interpretação do código fonte e construção da representação interna RTL, são criadas estruturas que representam as operações aritméticas simples, sendo só tentada a combinação de operações após a realização de alguns passos de optimização, como foi explicado na secção 3.4.1. Na tentativa de conseguir o resultado acima referido, foram criados padrões (`define_expand`) que definem como devem ser expandidas as operações aritméticas simples em expressões RTL. Por exemplo, uma adição `x=a+b` é expandida na sequência seguinte de operações elementares:

```
Y=1
E=b
tmp=a*Y
x=tmp+E
```

esperando que na fase de combinação a sequência multiplicação-adição seja reunida numa única instrução, após eliminar as atribuições desnecessárias aos registos `Y` e `E`. No entanto, como antes de ser tentada a combinação de operações são realizadas simplificações algébricas e propagação de constantes, que não podem ser evitadas porque são importantes para a optimização global do programa, aquela sequência de 4 operações resultante da expansão da adição é de novo simplificada para a mesma operação adição original: como é atribuída a constante `1` a `Y`, então a multiplicação é eliminada e o operando `tmp` da adição é substituído por `a`. Simplificações semelhantes acontecem para as outras operações aritméticas e para a instrução de deslocamento de dados.

Considerando este conjunto de instruções não foi possível garantir como resultado da compilação, apenas instruções que representem as operações aritméticas combinadas e as operações de carregamento dos registos `E` e `Y`. Por esse motivo, foi decidido não os considerar como registos do processador, vendo a unidade de cálculo como um único operador capaz de executar todas as operações aritméticas simples e combinadas. Isto conduziu à construção do conjunto de instruções 2 apresentado a seguir, que permitiu traduzir com sucesso código C para um conjunto de instruções *assembly* representando operações suportadas pelo processador THD.

Conjunto de instruções 2

No segundo conjunto de instruções considerado, os registos da unidade de cálculo não são vistos como registos do processador, sendo o seu carregamento feito sempre que é realizada uma operação aritmética simples ou combinada na unidade de cálculo.

Foram definidos padrões para representar as instruções aritméticas simples e combinadas, deslocamento de dados e determinação de máximo. No ficheiro de *assembly* gerado, apenas são suportadas as instruções que representam operações sobre dados do tipo `float`, que corresponde ao modo `SF` nos padrões de instrução. Como foi já dito antes, são também incluídos outros padrões de instrução que são requeridos pelo `gcc` em qualquer ficheiro de configuração, mas que produzem uma mensagem de erro se forem encontrados na lista de instruções resultante da compilação. Isso é feito invocando as funções `Invalidmachmode()` se for encontrada uma operação sobre um tipo de dado diferente de `SF` e `InvalidOpcode()` se for encontrada uma operação não suportada.

São consideradas 10 instruções *assembly*, que aceitam como operandos constantes, referências a variáveis em memória e "registos" do processador. A tradução posterior dessas instruções para microcódigo deve transformar essas referências em endereços físicos na memória de dados. As 10 instruções consideradas são listadas a seguir, representando do lado direito a operação realizada por cada uma

```
MOV  dest, orig              ; dest <- orig
MAX  dest, opr1, opr2        ; dest <- max(opr1,opr2)
ADD  dest, opr1, opr2        ; dest <- opr1 + opr2
SUB  dest, opr1, opr2        ; dest <- opr1 - opr2
MUL  dest, opr1, opr2        ; dest <- opr1 * opr2
DIV  dest, opr1, opr2        ; dest <- opr1 / opr2
MADD dest, opr1, opr2, opr3  ; dest <- opr1 * opr2 + opr3
MSUB dest, opr1, opr2, opr3  ; dest <- opr1 * opr2 - opr3
DADD dest, opr1, opr2, opr3  ; dest <- opr1 / opr2 + opr3
DSUB dest, opr1, opr2, opr3  ; dest <- opr1 / opr2 - opr3
```

Para além do ficheiro de configuração `*.md`, foi também criado o ficheiro com as definições de *macros* onde são descritas características adicionais do processador destino não incluídas no ficheiro `*.md`. Como existem certas restrições não documentadas na forma como alguns *macros* devem ser construídos, optou-se por usar como ponto de partida um ficheiro disponível na distribuição do compilador (`we32K.h`), tendo sido modificadas apenas as definições que caracterizam os registos do processador e as que intervêm na geração do código e controlo de algumas optimizações. As principais alterações no ficheiro de *macros* foram as seguintes:

- Registos do processador: foi definido um conjunto de 64 registos (`R0` a `R63`), que são posteriormente traduzidos em referências a endereços físicos na memória de dados. Desses registos, 4 são dedicados a funções especiais (apontador de programa e apontador de pilha, por exemplo), pelo que nunca podem ser usados para armazenar variáveis temporárias.

- Sequências de entrada e saída de funções: durante a emissão do código *assembly* de uma função podem ser definidas sequências padrão de instruções, a executar antes e depois das que constituem o corpo da função. Na configuração construída, todo o código produzido pelo compilador é reunido num único bloco básico e as chamadas a funções que possam ocorrer são substituídas localmente pelo seu código. Por esse motivo não é produzida qualquer sequência adicional de instruções no início de uma função, e no final é apenas emitida a directiva `END` que serve para representar o fim do programa.

- Sintaxe de escrita dos operandos no ficheiro de *assembly*: constantes são escritas em decimal, precedidas de #; referências a registos ou variáveis são traduzidas no nome respectivo; referências a elementos de um vector são representadas pelo nome do vector adicionado do índice que identifica o elemento.

- Custo relativo das operações: foi atribuído o mesmo custo a todas as operações suportadas pelo processador, e um custo 50 vezes superior a instruções de salto.

No apêndice B é listado o ficheiro de descrição da máquina (`procthd.md`) que contém as definições dos padrões de instrução, e a especificação do modo como devem ser tentadas as combinações das operações aritméticas. O código *assembly* a emitir para cada instrução e as mensagens de erro geradas quando são encontradas operações não suportadas, é feito por funções que são incluídas num ficheiro auxiliar em C. Como foi já dito antes, o ficheiro de descrição de *macros*, `procthd.h`, foi obtido à custa de um número reduzido de modificações num dos ficheiros já existente na distribuição do `gcc`.

Opções do compilador

O `gcc` construído com esta configuração deve ser invocado com as seguintes opções na linha de comando:

- `-S` : termina após gerar o ficheiro de *assembly*, sem invocar o programa que compõe o código executável (*linker*).
- `-O2` : activa todas as optimizações, excepto o desenrolamento de ciclos. Isto é necessário para obter o código final com as características acima referidas.
- `-funroll-loops` : executa o desenrolamento de ciclos. Após esta operação são repetidos alguns passos de optimização.

Restrições do código fonte em C

Como o processador destino suporta apenas o conjunto de 10 instruções referido acima, o código fonte a compilar está sujeito a restrições, embora sejam satisfeitas pelo tipo de programa que se pretende compilar para esta arquitectura. Um programa que satisfaz as restrições aqui enunciadas e que foi usado para experimentar a configuração construída é uma codificação em C do algoritmo para cálculo da função de libertação de calor, listado no apêndice A. Dada a sua extensão, não se apresenta o código *assembly* produzido pela compilação desse programa com o compilador construído. Um segmento desse código é listado no apêndice D.4, e corresponde a uma iteração do ciclo `for(...)` da função `iteration()` daquele programa.

As variáveis temporárias do tipo `float` devem ser declaradas globais e estáticas. As que forem declaradas locais a funções podem eventualmente ser eliminadas durante os passos de optimização, mas se não existirem disponíveis registos do processador em número

suficiente são traduzidas em referências na pilha e o compilador falha porque não são definidas instruções que suportem a manipulação dessa estrutura de memória.

Variáveis inteiras podem ser usadas no código fonte mas devem ser declaradas locais, pois só nesse caso podem ser eliminadas pelas optimizações. Uma aplicação possível de variáveis inteiras é no controlo de ciclos com número constante de iterações. Como os ciclos têm que ser desenrolados na sequência de todas as suas iterações, essas variáveis são eliminadas e traduzidas em constantes que correspondem ao valor assumido em cada iteração do ciclo.

Como o processador destino não tem instruções de salto, o código final tem de ser um único bloco básico de instruções. Para que isso seja garantido têm de ser verificadas as regras seguintes:

- Ciclos (`for`, `while`, `do...while`): podem ser usados desde que seja possível expandi-los nas sequências de instruções que representam cada iteração. Isto é possível se o número de iterações do ciclo for conhecido durante a compilação, por exemplo:

```
for(i=0;i<100;i++)
{...}
```

- Instruções condicionais (`if`, `case`): podem aparecer no código fonte, desde que o resultado da expressão condicional possa ser avaliado durante a compilação, como se mostra no seguinte exemplo:

```
for(i=0;i<100;i++)
{...
  if ( i>0 && i<99 ) {...} else {...}
}
```

- Chamadas a funções: podem ser incluídas no código fonte, desde que as funções chamadas sejam declaradas com a directiva `inline`. Isto faz com que a chamada à função seja substituída pelo seu código, não existindo restrições quanto à passagem de argumentos ou retorno de resultados.

- Referências a vectores em memória: como o processador não suporta endereçamento indexado à memória, todos os índices de vectores têm de poder ser traduzidos para constantes inteiras, como se mostra no seguinte exemplo:

```
for(i=0;i<100;i++)
{...
  if ( i>0 && i<99 )
  {... dMU[i-1]=MU[i]-MU[i-2] ...}
}
```

Se após as optimizações realizadas pelo gcc persistirem instruções para além das 10 suportadas pelo processador, o compilador falha e é emitida uma mensagem de erro que identifica a construção não suportada que foi encontrada no código compilado.

3.4.4 Sequenciamento e compactação de microcódigo

O código *assembly* gerado pelo compilador tem de ser traduzido para o microcódigo que constitui o programa que o processador THD executa. Como a unidade de cálculo funciona de forma *pipelined* com latência igual a um ciclo de relógio e com 13 ciclos de duração, então as 17 micro-instruções em que cada instrução *assembly* é traduzida podem ser parcialmente sobrepostas no tempo. Para minimizar o número de ciclos de relógio necessários à execução de um programa (que é igual ao número de micro-instruções), as instruções produzidas pelo compilador podem ser reordenadas, preservando no entanto as dependências de dados existentes entre as operações que representam.

Para automatizar esta operação, foi desenvolvido um programa que implementa o algoritmo de sequenciamento por lista de prioridades (*list scheduling*), tomando em consideração as restrições impostas pela execução *pipelined* de operações na unidade de cálculo. Esta técnica de sequenciamento mostrou conseguir bons resultados na optimização de microcódigo para processadores *pipelined* [DLSM81, SP89], e em aplicações de síntese de alto nível de circuitos digitais em que os recursos de *hardware* são fixos e conhecidos [GWDL92a]. O programa que foi desenvolvido interpreta o código *assembly* gerado pelo gcc, constrói um grafo dirigido representando as dependências de dados entre as operações e produz como resultado uma atribuição das instruções a ciclos de relógio que conduz à compactação do microcódigo em que o programa é traduzido.

Dadas as características de latência da unidade de cálculo, em cada ciclo de relógio poderia ser iniciada uma operação de tipo diferente. No entanto, como os 3 operandos necessários a cada operação são lidos da memória de dados em 3 ciclos consecutivos, só pode ser iniciada uma nova operação em cada 3 ciclos de relógio, consumindo cada uma 17 ciclos de relógio: 3 para ler os operandos, 13 para produzir o resultado e um para escrever o resultado na memória de dados. Embora a função realizada por instrução *assembly* seja especificada por apenas 4 micro-instruções, entre as 3 primeiras e a última

tem de decorrer exactamente 13 ciclos de relógio, porque não há nenhum registo à saída do somador que possa armazenar temporáriamente esse resultado. Durante os 13 ciclos em que os dados são propagados pela *pipeline*, um máximo de 4 novas operações podem ser iniciadas, desde que sejam naturalmente satisfeitas as dependências de dados que possam existir entre elas.

Na figura 3.5 exemplifica-se a forma como é representado o microcódigo de 3 operações, e como elas podem ser executadas de forma *pipelined*. Cada operação é representada pela lista de 17 micro-instruções em que cada instrução *assembly* é traduzida. As micro-instruções são divididas em dois tipos: `m` são micro-instruções que incluem acessos à memória de dados e a especificação da operação a realizar pela unidade de cálculo e `x` representam micro-instruções que não têm interferência na operação realizada e que correspondem aos ciclos de relógio em que os dados são propagados pela *pipeline* da unidade de cálculo. A regra para compactar as sequências de micro-instruções representadas desta forma é apenas uma e resume-se a não poderem ser atribuídas ao mesmo ciclo de relógio duas micro-instruções do tipo `m`.

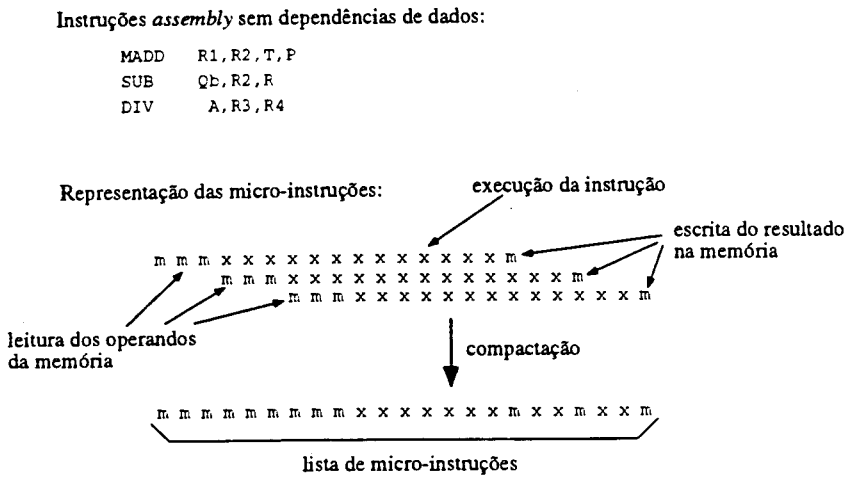


Figura 3.5: Compactação das micro-instruções de 3 operações sem dependências de dados.

O algoritmo de sequenciamento mantém uma lista de micro-instruções, onde são colocados os padrões de caracteres `m` e `x` que representam os dois tipos diferentes de micro-instruções em que uma instrução *assembly* é traduzida. Esta lista serve para determinar em que ciclo podem ser colocadas micro-instruções, sem que isso conduza a conflitos no acesso à memória de dados. Assim, sempre que uma operação é atribuída a um ciclo de relógio é tentada a sua colocação nesse ciclo da lista de micro-instruções. Se tal não for possível, então a operação é atrasada sucessivamente até ser encontrada uma posição nessa lista onde possa ser colocada.

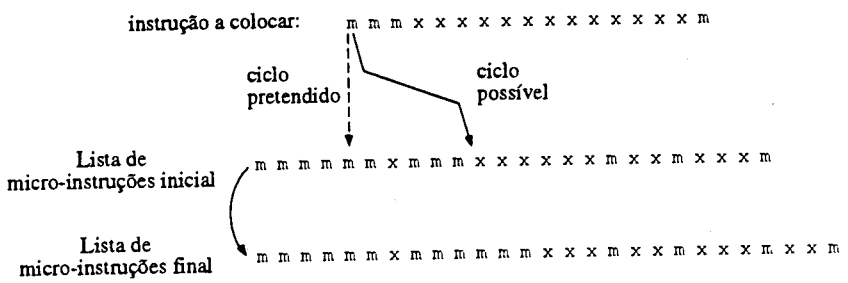


Figura 3.6: Colocação de uma instrução na lista de micro-instruções.

Na figura 3.6 representa-se o conteúdo da lista de micro-instruções antes e depois da colocação de uma nova instrução. Embora pelas restrições resultantes das relações de precedência entre as operações a instrução pudesse ser colocada no ciclo indicado pela seta a tracejado, é necessário atrasá-la 6 ciclos para que não ocorra sobreposição com outras micro-instruções do tipo `m` já presentes na lista de micro-instruções, o que representaria conflitos no acesso à memória de dados.

O algoritmo de sequenciamento com lista de prioridades é descrito pelo pseudo-código apresentado na figura 3.7. É mantida uma lista com as operações prontas a sequenciar (*ready list*), que são todas aquelas cujos antecessores foram já atribuídos a ciclos de relógio. A lista é ordenada segundo uma função de prioridade calculada para cada operação e é inicialmente criada com todas as operações cujos operandos não são produzidos por outras operações (as que não têm antecessores). É seleccionada a operação com prioridade mais elevada e é atribuída ao primeiro ciclo de relógio, permitido pelas restrições de precedência entre as operações e pela ocupação da lista de micro-instruções. A operação escolhida é retirada da lista de operações prontas, são acrescentadas todas as operações que ficaram prontas a sequenciar e a lista é reordenada segundo a função de prioridade calculada para cada operação.

```
LISTA= operações sem antecessores;
ENQUANTO LISTA não vazia
    Op = operação em LISTA com prioridade máxima;
    coloca Op;
    remove Op da lista;
    LISTA = LISTA + (sucessores de Op com todos os antecessores colocados)
```

Figura 3.7: Pseudo-código do algoritmo de sequenciamento com lista de prioridades.

A função de prioridade usada foi inspirada na que é proposta na aplicação de sequenciamento de microcódigo apresentada em [SP89], e é uma soma pesada de parâmetros extraídos do grafo de dependências de dados e da lista de micro-instruções: o número de

operações no caminho mais longo desde a operação corrente até ao nó do grafo que representa o fim do problema, o número de operações que usam o resultado produzido pela operação corrente (sucessores), o número de operações que produzem os operandos consumidos pela operação corrente (antecessores) e o primeiro ciclo em que a operação pode ser colocada. Enquanto que os três primeiros parâmetros podem ser calculados uma vez construído o grafo de dependências de dados que representa o programa, a determinação do primeiro ciclo em que pode ser colocada cada operação tem de ser feita dinamicamente, já que depende do estado da lista de micro-instruções no decorrer do algoritmo de sequenciamento.

3.4.5 Resultados

A compilação do programa de teste listado no apêndice A conduziu a um programa *assembly* com 21980 instruções escalares, das quais 10729 (49%) são operações aritméticas combinadas identificadas pelo gcc. Com as instruções *assembly* consideradas lendo 3 operandos e escrevendo um resultado na memória de dados, o número mínimo de ciclos de relógio que esse programa necessitaria é imposto pelo número de acessos à memória, igual a $4 \times 21980 = 87920$. Com o processador a trabalhar com a frequência de 20 MHz, isto traduzir-se-ia num tempo igual a 4.4 ms, só para efectuar todos os acessos à memória de dados. Este resultado permitiu concluir desde cedo que a execução do programa assim obtido, contendo apenas instruções escalares, não conseguiria satisfazer os requisitos de tempo real da aplicação, que admitia um tempo de execução máximo de 1.35 ms. No entanto, o compilador de C construído torna praticável a experimentação de diferentes algoritmos para a aplicação de controlo, possibilitando a sua comparação e avaliação sem obrigar à tradução manual de programas complexos para linguagem *assembly*.

Tabela 3.1: Pesos usados para a função de prioridade do algoritmo de sequenciamento.

caminho mais longo	sucessores	antecessores	ciclo mais cedo
5	3	1	4

O algoritmo de sequenciamento foi implementado em C e compilado com o gcc numa estação de trabalho SUN Sparc10. Após várias experiências usando diferentes combinações de pesos para a função de prioridade, foram utilizados os pesos apresentados na tabela 3.1. Com esta técnica de sequenciamento, as 21980 instruções do programa de teste foram compactadas num mínimo de 213540 micro-instruções, o que representa 57% do número de ciclos necessários para executar as 21980 operações de forma não *pipelined*,

sem compactar o microcódigo ($21980 \times 17 = 373660$ ciclos). A execução deste programa no processador vectorial requer igual número de ciclos de relógio, que representa um tempo de execução igual a 10.7 ms (20 MHz).

Sem se recorrer às memórias vectoriais, não é utilizada parte importante da característica vectorial do processador e o desempenho máximo que é possível atingir é reduzido significativamente. Para satisfazer os requisitos de processamento em tempo real da aplicação a que o processador dedicado se destina, é necessário e justificável a codificação em instruções de baixo nível, recorrendo às operações vectoriais específicas suportadas por esta arquitectura.

3.5 Conclusões

O processador vectorial apresentado neste capítulo é um exemplo de um sistema digital que foi desenvolvido como um circuito integrado de aplicação específica, tendo em vista as necessidades de processamento de uma aplicação bem definida. Apesar de ser programável, a estrutura fixa do *hardware* faz com que não seja possível a sua utilização eficiente noutras aplicações, ainda que difiram pouco daquela para que o circuito foi inicialmente concebido. Por exemplo, em aplicações semelhantes que utilizem um número diferente de amostras de pressão e volume, já não é possível utilizar este processador com a mesma eficácia que é conseguida para as 128 amostras, devido à dimensão fixa das memórias vectoriais.

O problema da construção de programas para o processador THD foi abordado a dois níveis diferentes. O primeiro, baseado num tradutor de instruções de baixo nível vectoriais e escalares, permitiu obter microprogramas que utilizam de forma muito eficiente os recursos do processador e que satisfazem os requisitos de tempo real da aplicação para a qual o sistema foi desenvolvido. No entanto, obriga a identificar manualmente as operações vectoriais e traduzir o algoritmo a executar no conjunto de instruções de baixo nível considerado por esse tradutor. Para ultrapassar essa limitação e possibilitar a programação deste processador dedicado numa linguagem de alto nível, foi adaptado um compilador de C configurável, tornando possível a tradução de programas na linguagem de alto nível C para instruções suportadas pelo processador dedicado. O compilador produz uma sequência de instruções *assembly*, que posteriormente são traduzidas em microcódigo compactado e executadas pelo processador dedicado. Porque não foi possível utilizar as memórias vectoriais que são componentes fundamentais desta arquitectura, os programas traduzidos por este processo não podem competir com codificações optimizadas manualmente de maneira a explorar a execução de operações vectoriais, que são a chave para o

elevado desempenho que este processador consegue atingir.

O compilador construído constitui uma ferramenta para programação em alto nível do processador THD, tornando possível comparar diferentes alternativas de algoritmos, sem exigir a sua codificação em código de baixo nível, tarefa demorada e sujeita a erros. Apesar de o compilador usado não oferecer a flexibilidade que seria desejável para contemplar características específicas encontradas em processadores não convencionais, permite aliviar uma parte significativa do trabalho de construção de ferramentas de suporte para programação em alto nível de processadores dedicados. Optimizações posteriores do código produzido pelo compilador são necessárias para explorar convenientemente as particularidades que cada arquitectura alvo apresenta. O programa que foi desenvolvido para sequenciar e compactar o microcódigo é disso um exemplo, assim como a aplicação apresentada no capítulo 5 para optimizar a configuração para uma arquitectura reconfigurável. No caso concreto desta aplicação, uma solução para tornar possível a execução em tempo real do código produzido pelo compilador de C exigiria a construção de ferramentas que permitissem identificar automaticamente operações em vectores, de forma a explorar a utilização das memórias vectoriais.

Capítulo 4

PUFR3 - Um processador reconfigurável

4.1 Introdução

Neste capítulo é proposta uma arquitectura para um processador reconfigurável, construído em torno de operadores implementados em circuitos lógicos programáveis FPGA. A arquitectura proposta tem por objectivo executar sequências de operações extraídas de uma especificação de alto nível de um algoritmo, explorando a realização de tarefas seleccionadas em operadores construídos como circuitos lógicos desenhados expressamente para as necessidades específicas de diferentes aplicações.

A arquitectura reconfigurável que aqui se propõe foi idealizada para constituir um processador auxiliar de um computador de uso geral, acelerando tarefas seleccionadas pelo utilizador. A sua estrutura foi inspirada na do processador THD e integra um núcleo reconfigurável construído com 3 circuitos lógicos programáveis FPGA que constituem as chamadas unidades funcionais. Estas permitem implementar, para cada aplicação, um conjunto de operadores seleccionados de acordo com as características específicas de cada aplicação. A estrutura de interligações flexíveis entre as unidades funcionais permite também encadear as funções realizadas por cada uma, de forma semelhante às sequências de multiplicações e adições realizadas pela unidade de cálculo do processador THD.

Neste capítulo discutem-se detalhes de implementação da arquitectura proposta, com destaque para a selecção e construção de operadores para aplicações específicas a incluir nas unidades funcionais. Apresentam-se exemplos seleccionados de tarefas para as quais se desenvolveram operadores dedicados, realizados como circuitos lógicos desenhados à medida e implementados em circuitos FPGA. São comparados os desempenhos dessas

implementações com realizações optimizadas em *software* e é apresentado um estudo detalhado de uma aplicação concreta, na qual foi identificado e implementado um operador dedicado para uma função específica.

O problema da configuração desta arquitectura de forma optimizada para uma aplicação específica é abordado no capítulo seguinte. Consideram-se identificadas as operações do problema que devem ser realizadas como operadores dedicados, e que esses operadores residem numa biblioteca formada por configurações para os circuitos FPGA que constituem as unidades funcionais. O trabalho que se apresenta no capítulo seguinte foca o problema da escolha das configurações para as unidades funcionais, ordenação temporal (sequenciamento) das operações do problema e afectação dessas operações aos operadores realizados nas unidades funcionais. Este é um dos problemas centrais no processo de optimização de um algoritmo para um processador reconfigurável com as características consideradas, para o qual é apresentada uma solução desenvolvida com base numa técnica de optimização combinatória (*Simulated Annealing*).

4.2 Motivação - o processador THD

A arquitectura do processador vectorial THD apresentado no capítulo 3 foi estabelecida pelas características do tipo de problema que se pretendia resolver. A existência de um grande número de operações com vectores de dimensão fixa e conhecida e a identificação de operações combinadas, constituídas por multiplicações ou divisões seguidas de adições ou subtracções, conduziu à realização de uma arquitectura de processamento vectorial com uma unidade de cálculo composta por um multiplicador/divisor seguida de um somador/subtractor.

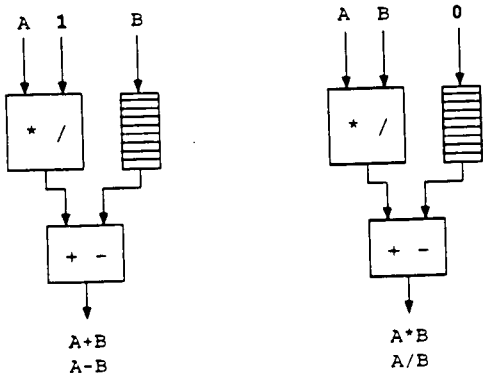


Figura 4.1: Execução de operações aritméticas simples no processador THD.

A rigidez resultante da interligação fixa entre aqueles dois operadores conduz a que todas as operações que a unidade de cálculo executa sejam sempre daquele tipo. Como consequência, operações simples (multiplicações ou adições, por exemplo) têm de ser realizadas como casos particulares daquelas operações combinadas, exigindo um número superior de ciclos de relógio do que seria necessário para executar só uma operação. Por exemplo, para efectuar uma multiplicação é necessário adicionar 0 ao valor que sai do multiplicador, obtendo o resultado à saída do somador/subtractor. De forma semelhante, realizar uma adição obriga a multiplicar um dos operandos por 1 (figura 4.1).

Para o tipo de aplicações que se pretende executar naquele processador, esta característica não se traduz numa degradação de desempenho significativa, já que o tempo de execução é dominado pela realização de operações combinadas daquele tipo. Além disso, o facto dessas operações serem executadas pela unidade de cálculo de forma *pipelined* com uma latência de um ciclo de relógio, faz com que o acréscimo de tempo devido à execução de uma operação adicional seja diluído no tempo total de processamento de sequências longas de operações, como acontece naquela aplicação com a execução de operações vectoriais.

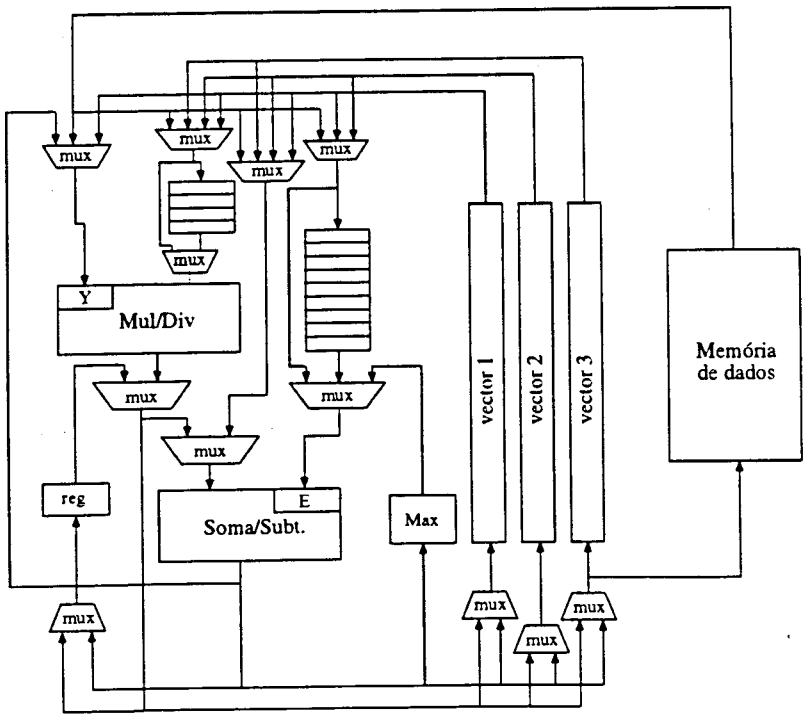


Figura 4.2: Proposta de uma arquitectura flexível para o processador THD.

É claro, no entanto, que a introdução de configurabilidade ao nível da interligação entre os dois operadores que compõem aquela unidade de cálculo aumentaria a sua fle-

xibilidade e possibilitaria a sua utilização noutras aplicações com diferentes requisitos de operações combinadas, tornando possível, por exemplo, a execução de somas ou subtracções em paralelo com multiplicações ou divisões. Na figura 4.2 mostra-se uma sugestão de alterações a introduzir no processador THD de maneira a incrementar a sua flexibilidade, embora mantendo os mesmos operadores originais. Essas alterações consistem essencialmente na introdução de circuitos selectores que permitem reconfigurar os caminhos de dados entre os operadores e as unidades de armazenamento de dados. Com esta arquitectura pode afirmar-se que é acrescentada capacidade de reconfiguração estrutural ao processador original, alargando com isso o âmbito da flexibilidade já oferecida pela execução de instruções. Na figura 4.3 apresentam-se as operações combinadas suportadas pela arquitectura modificada.

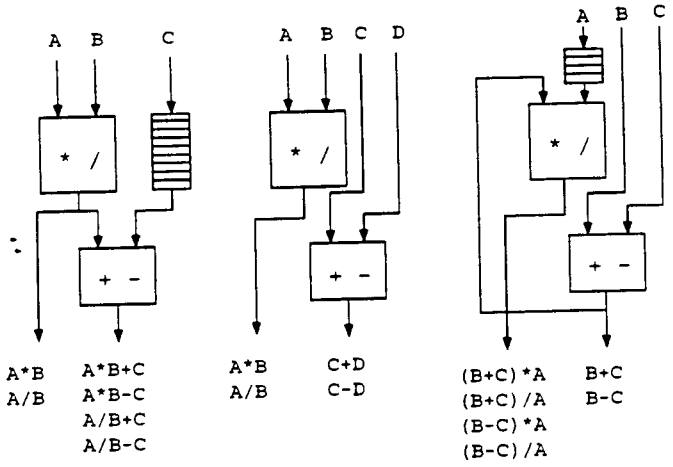


Figura 4.3: Operações combinadas suportadas pelo processador THD modificado.

A flexibilidade conseguida com as interligações configuráveis entre os operadores daquele processador, sugeriu a concepção de uma arquitectura para um processador reconfigurável com operadores construídos sobre circuitos lógicos programáveis do tipo FPGA. Isso permite levar mais longe a reconfigurabilidade estrutural exibida pela modificação proposta para o processador THD, substituindo os seus operadores originais por um conjunto de operadores alargado e adaptado às características próprias de cada aplicação.

4.3 Arquitectura do PUFR3

Conforme foi explicado antes, o processador reconfigurável que se propõe é construído em torno de um núcleo de processamento que pode ser visto como uma extensão do processador vectorial apresentado no capítulo anterior. É constituído por um número

fixo de unidades funcionais realizadas como circuitos lógicos programáveis FPGA, que podem ser configurados de forma a implementarem um conjunto alargado de operadores, adequados para as características de cada aplicação. O recurso a uma rede de interligações flexível entre as unidades funcionais permite combinar as operações realizadas por cada unidade, criando desse modo funções complexas construídas como composições desses operadores. Na figura 4.4 apresenta-se a proposta de arquitectura para o processador PUFR3.

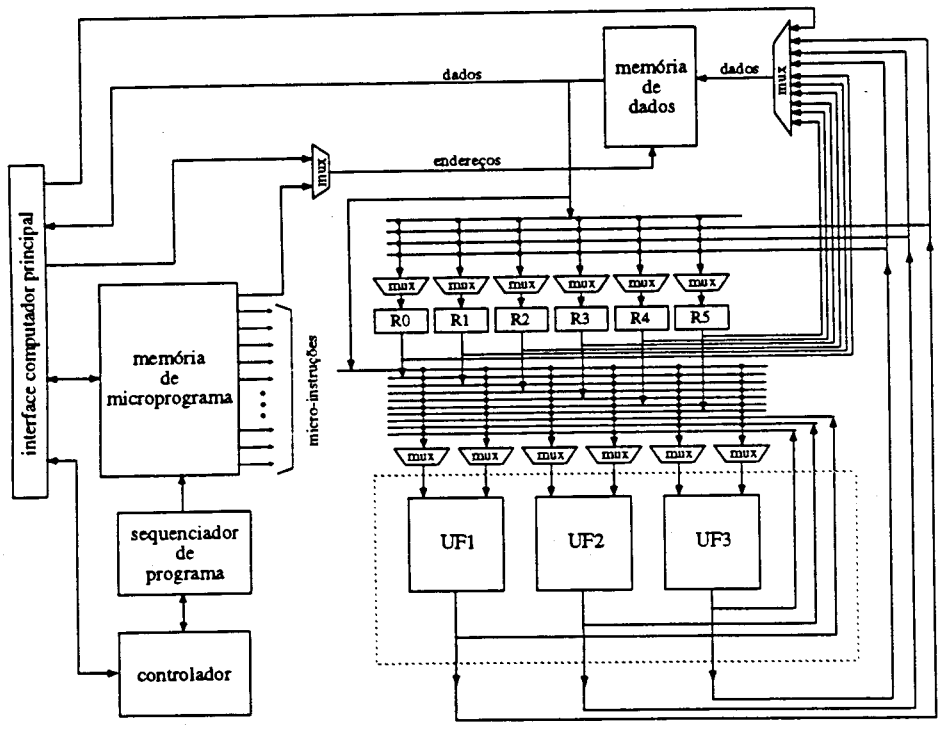


Figura 4.4: Arquitectura do processador reconfigurável PUFR3

Nesta proposta são acrescentados dois níveis de reconfiguração estrutural em relação ao processador THD em que foi inspirado, para além do facto de executar de igual modo programas formados por sequências de micro-instruções horizontais. O primeiro consiste na possibilidade de reconfiguração das unidades funcionais de forma a construir um conjunto de operadores específicos para diferentes aplicações. O segundo nível é introduzido pela capacidade de modificar as interligações entre as unidades funcionais, de maneira a criar operadores complexos construídos como combinações das funções realizadas por cada uma. A configuração das interligações entre as unidades funcionais é definida por campos das micro-instruções e pode por isso ser modificada por cada instrução no decorrer da execução de um programa. No que diz respeito à forma como é feita a reconfiguração dos

FPGAs que constituem as unidades funcionais, são considerados dois cenários diferentes: reconfiguração estática e reconfiguração dinâmica. Usando reconfiguração estática, o circuito implementado em cada unidade funcional é fixado de acordo com as características da aplicação, e é mantido durante a sua execução; considerando que as unidades funcionais podem ser reconfiguradas dinamicamente, a função realizada por cada uma pode ser modificada durante a execução da aplicação, de forma a implementar diferentes funcionalidades que vão sendo necessárias ao longo da execução de um programa. Estes dois cenários requerem diferentes abordagens no processo de optimização de uma configuração para esta arquitectura que são tratadas no capítulo seguinte.

O número de unidades funcionais foi fixado em 3 e de forma semelhante ao processador THD, também se considera que é executada uma sequência de operações, sem saltos ou instruções condicionais (bloco básico). Nas secções seguintes são abordados alguns detalhes de implementação dos principais blocos funcionais: interface com o processador principal, banco de registos e unidade de cálculo.

4.3.1 Interface com o processador principal

A arquitectura que se propõe para o processador reconfigurável foi idealizada para acelerar partes seleccionadas de programas, sob controlo do processador principal de um computador convencional. Dada a crescente vulgarização dos chamados PC-compatíveis e a tendência de este tipo de computador continuar a ser, pelo menos num futuro próximo, o padrão do computador pessoal com elevada relação desempenho/custo, considerou-se esta plataforma computacional como o destinatário da arquitectura reconfigurável proposta.

O processador reconfigurável é proposto para funcionar como um periférico do processador principal, comunicando com este pelo barramento de expansão do sistema. O conjunto de dados a processar é transferido para a memória do processador reconfigurável e é iniciada a execução do microprograma previamente carregado na memória de programa. Quando o programa termina, o processador principal é notificado, recuperando então da memória os resultados do processamento efectuado.

Como foi já visto no capítulo 2, um factor que contribui para amortecer ou mesmo anular os ganhos em rapidez conseguidos pela utilização de processadores auxiliares dedicados, é a taxa limitada de transferência de dados com o processador principal de um computador, recorrendo aos barramentos de uso geral disponíveis para expansão. Esse tipo de ligação é feita através dos circuitos de interface associados a esses barramentos, reduzindo a rapidez de comunicação em relação àquilo que seria possível atingir se fosse usada uma ligação directa ao processador. No entanto, em computadores comerciais não

é geralmente possível realizar esse tipo de ligação.

Para os computadores pessoais baseados em processadores Intel 80x86, genericamente designados por PCs, dois barramentos comuns disponíveis para expansão são o ISA (*Industry Standard Architecture*) [SA96] e o PCI (*Peripheral Component Interconnect*) [SA95]. O barramento ISA de 8 *bits* foi introduzido com o IBM PC nos finais dos anos 70, e a extensão para 16 *bits* foi acrescentada poucos anos mais tarde com o PC-AT. Actualmente é ainda incluído na generalidade dos PCs, embora venha a ser progressivamente substituído pelo barramento PCI, que permite taxas de transferência de informação muito mais elevadas. Enquanto que a comunicação através do barramento ISA é sincronizada com um relógio de 8 MHz e permite uma taxa máxima de transferência de informação igual a 8 M*bytes*/s, a revisão 2.1 do barramento PCI [SA95] pode trabalhar com frequências máximas de 66 MHz e atingir taxas de transferência de dados de 528 M*bytes*/s, o que representa 66 vezes mais do que o conseguido com o barramento ISA.

A realização de uma interface para o barramento PCI requer a satisfação de restrições físicas de implementação previstas na norma [PCI92]. Estas impõem limites apertados nos tempos de atraso dos circuitos que realizam a interface e até nas dimensões físicas das pistas da carta de circuito impresso onde é montado o sistema, o que indirectamente condiciona também as características físicas dos tipos de caixas dos circuitos integrados a utilizar (dimensões e disposição dos terminais).

A implementação física de um interface para o barramento ISA não apresenta grandes dificuldades técnicas, dada a relativa baixa frequência dos sinais envolvidos. No entanto, apenas são permitidas transferências de dados de 16 *bits*, e a rapidez real conseguida com transferências realizadas sob controlo do processador principal fica sempre aquém daquele valor máximo (8 M*bytes*/s). Isto representa uma limitação importante, já que na avaliação do ganho de desempenho conseguido por um processador auxiliar é naturalmente necessário contabilizar também o tempo consumido para transferir informação entre os dois sistemas. A experiência adquirida com outras implementações realizadas pelo autor com circuitos FPGA para este barramento de expansão, levou a considerar a sua utilização numa realização prática da arquitectura que aqui se propõe.

No entanto, dentro dos limites aceites para este desenvolvimento, a arquitectura do processador reconfigurável é independente da forma como é realizada a comunicação com o computador principal, embora disso dependa em grande parte o desempenho global que possa ser atingido.

4.3.2 Memória de dados e banco de registos

A memória de dados é usada como memória principal do processador reconfigurável e serve também para transferir dados e resultados com o processador principal. No decorrer da execução de um programa, a memória de dados pode ser escrita com o resultado presente à saída de qualquer uma das três unidades funcionais ou com o conteúdo de qualquer um dos bancos de registos. Os dados lidos da memória podem ser encaminhados para qualquer banco de registos ou para qualquer entrada das unidades funcionais.

Os 6 bancos de registos representados na figura (R0 a R5) são unidades de memória destinadas ao armazenamento temporário de informação durante a execução de um programa. Com a estrutura de interligações proposta, podem ser carregados em paralelo com informação lida da memória de dados ou com os resultados presentes nas saídas de qualquer uma das unidades funcionais. Por outro lado, as saídas dos bancos de registos podem alimentar qualquer uma das duas entrada das 3 unidades funcionais, permitindo fornecer em cada ciclo de relógio um conjunto de dois novos operandos a cada uma.

A arquitectura para os 6 bancos de registos considerados pode assumir variadas formas, conduzindo a diferentes capacidades de armazenamento, modos de acesso e sinais de controlo necessários a incluir nas micro-instruções. Uma sua realização física com circuitos FPGA permite seleccionar para estas unidades as arquitecturas que melhor se adaptem às exigências de diferentes aplicações.

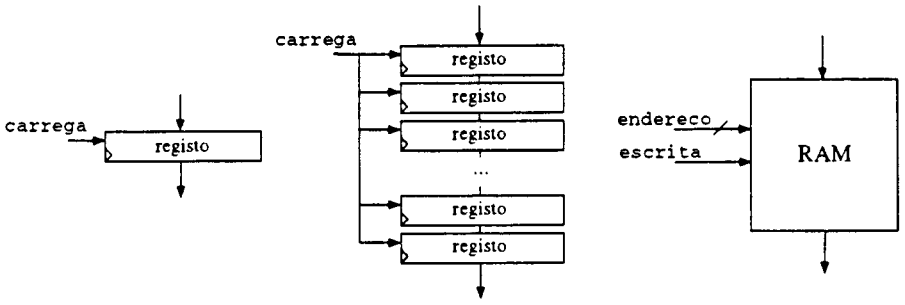


Figura 4.5: Alternativas para a realização física dos bancos de registos do PUFR3.

Na figura 4.5 mostram-se 3 alternativas possíveis para a construção dos bancos de registos. Na sua configuração mais simples, cada um é formado por um único registo e pode armazenar um único dado. Pode também ser construído como um conjunto de registos endereçáveis de forma directa (ou uma memória RAM) ou como uma memória vectorial construída como um registo de deslocamento, semelhante às usadas no processador THD. Enquanto uma memória vectorial desse tipo não necessita de mais linhas de controlo do

que as que são necessárias para um registo escalar, o uso de um conjunto de registos ou de uma memória RAM obriga à inclusão de campos nas micro-instruções que especifiquem os endereços dos registos seleccionados para cada operação.

4.3.3 Unidade de cálculo

A unidade de cálculo é formada por 3 unidades funcionais implementadas como circuitos FPGA, e são configuradas para cada aplicação de forma a oferecerem operadores adequados às suas necessidades. Além disso, podem também ser interligadas entre si de forma a realizar diferentes combinações dos operadores implementados em cada uma. Na figura 4.6 mostram-se as 5 configurações de interligação possíveis com três unidades funcionais. Para cada uma dessas configurações, há ainda a considerar as diferentes funções que resultam do facto de cada unidade poder oferecer vários operadores para funções distintas. Embora tenham sido consideradas 3 unidades funcionais, a arquitectura proposta pode ser expandida para um número superior de unidades, naturalmente à custa de um aumento da complexidade dos elementos de interligação e da largura das micro-instruções.

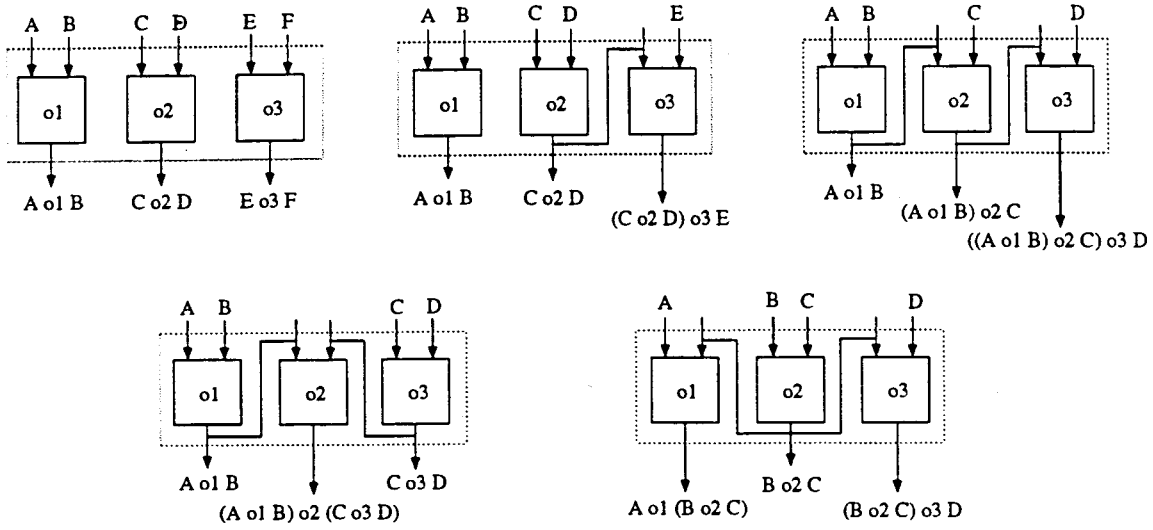


Figura 4.6: Configurações possíveis para a unidade de cálculo do PUFR3 com 3 unidades funcionais. Os operandos de cada configuração, referidos na figura como A, B, C, D, E, F, podem ser provenientes de qualquer registo ou da memória de dados, assim como as saídas de cada unidade funcional podem ser dirigidas para qualquer registo ou para a memória de dados.

A possibilidade de configurar as unidades funcionais para implementar um conjunto de operadores seleccionados expressamente para cada aplicação, permite encarar o processo de configuração desta plataforma como um problema de síntese de alto nível de circuitos digitais. Dada uma representação algorítmica de um problema a executar neste processador, a informação necessária para o configurar é formada por um conjunto de operadores a implementar nas unidades funcionais e por uma sequência de palavras de controlo, que constituem o programa a executar para realizar a tarefa pretendida. Optimizar este processo de tradução, de forma a explorar as características dos operadores disponíveis e o paralelismo exibido pelas operações do problema, envolve as mesmas tarefas nucleares requeridas em síntese de alto nível de circuitos digitais: escolha de um conjunto de operadores, sequenciamento das operações do problema e atribuição das operações aos operadores seleccionados.

A realização física de circuitos lógicos em FPGAs resulta necessáriamente com piores desempenhos do que recorrendo a circuitos programados por máscara ou fabricados expressamente para uma aplicação. Por este motivo, a utilização da arquitectura proposta implementando nas unidades funcionais apenas operadores para operações do mesmo tipo que as realizadas por processadores convencionais, dificilmente conseguirá competir, em termos de rapidez, com programas executados por processadores de uso geral. Nessa situação, possíveis ganhos de rapidez só poderão resultar do funcionamento em paralelo dos operadores realizados nas unidades funcionais, ou da exploração de operadores que trabalhem sobre dados com dimensões não suportadas por processadores convencionais. Um exemplo disso é o processador dedicado que se apresenta no capítulo 6: apesar de os circuitos lógicos implementados nos FPGAs realizarem apenas operações do mesmo tipo das suportadas por instruções de processadores convencionais (somas, multiplicações e comparações em variáveis inteiras), a exploração do paralelismo exibido pelas operações do problema conduziu a ganhos importantes de rapidez em relação a implementações optimizadas em *software*.

O uso de circuitos programáveis FPGA é atraente porque permite usar operadores desenhados propositadamente para as necessidades específicas de diferentes aplicações. Como foi já referido, existem situações em que com circuitos lógicos de complexidade reduzida, é possível construir operadores específicos que apresentem desempenhos muito melhores do que os conseguidos por implementações em *software*, por estas requererem a execução de um número elevado de instruções de processadores convencionais. Exemplos são operações que envolvam manipulação de *bits* ou trabalhem com dados cujas dimensões não se ajustem às fronteiras dos múltiplos de 8 *bits*, considerada pela generalidade dos computadores convencionais.

4.4 Exemplos de operadores dedicados

Nesta secção apresentam-se exemplos seleccionados de operadores dedicados que requerem pequena complexidade de *hardware* tirando partido das características da operação que realizam e que conduzem a ganhos de desempenho importantes em relação a implementações optimizadas em *software*. Não se pretende naturalmente abranger todas as áreas possíveis com as aplicações que se apresentam e muitas outras podem ser encontradas nos vários trabalhos referidos no capítulo 2.

As versões em *hardware* foram implementadas em FPGAs XILINX XC4010-4 e experimentadas no sistema ISAX apresentado na secção 2.3.5 do capítulo 2. Os circuitos lógicos foram desenhados usando as ferramentas para captura esquemática da ViewLogic e o sistema de desenvolvimento da XILINX XACT Step 6.0. Os valores apresentados para a complexidade dos circuitos obtidos representa o número de blocos lógicos configuráveis (CLB—*Configurable Logic Block*) ocupados pela implementação naquela família de FPGA. Os tempos de atraso e frequência máxima de relógio foram obtidos com o programa XDELAY do sistema de desenvolvimento da XILINX, considerando que os operandos residem em registos e que os resultados são também colocados em registos.

Para obter medidas comparativas de rapidez das implementações em *hardware* com versões em *software*, foram comparados os tempos de realização dessas operações nos operadores construídos com os tempos de execução de versões em *software*. Pela sua simplicidade, as implementações em *software* foram escritas e optimizadas directamente em código *assembly* do 80286 e foram executadas num PC-Pentium a 120 MHz. Os tempos apresentados são tempos médios de execução do núcleo de instruções *assembly* que implementam as operações, tendo sido obtidos executando a rotina um número elevado de vezes e subtraindo o tempo consumido com a passagem de argumentos e chamada à função.

O espaço necessários à realização em *software* é apresentado em número de *bytes* ocupados pelo código máquina da rotina. Se se considerar que em cada CLB daquela família de FPGA pode ser realizada uma memória RAM com uma capacidade igual a 4 *bytes*, pode-se estabelecer um critério de comparação em termos de espaço ocupado entre aqueles dois domínios de realização. Esta comparação deve ser no entanto considerada com algumas reservas, já que FPGAs e os circuitos de memória de um PC são estruturalmente e funcionalmente muito distintos, e a área física de silício ocupada por um CLB é seguramente muito maior do que o espaço físico ocupado por 4 *bytes* numa memória dinâmica.

4.4.1 Contagem de *bits*

Dado um vector de N *bits*, pretende-se um operador que conte o número de *bits* que são iguais a 1. Esta operação é usada, por exemplo, para determinar a distância de Hamming entre dois números inteiros, calculada como o número de *bits* iguais a 1 do resultado da operação lógica OU-EXCLUSIVO entre os dois operandos (exemplo retirado de [AS93]). A implementação em *software* de uma rotina que realize esta função pode ser feita segundo duas abordagens, com diferentes desempenhos e requisitos de memória: utilizando uma tabela onde o vector dado como argumento funciona como apontador e o seu conteúdo representa o número de *bits* iguais a 1 para cada valor assumido por esse argumento, ou analisando em sequência todos os *bits* do operando, contando os que são iguais a 1. Enquanto a primeira solução é rápida mas obriga a um espaço de memória para manter uma tabela com 2^N inteiros, a segunda solução é mais lenta mas pode ser realizada usando apenas registos do processador. Na figura 4.7 é listado o código *assembly* das duas versões acima descritas, considerando operandos de 16 *bits*. Podem ser construídas outras versões entre estes dois extremos que combinem a utilização de tabelas de consulta para obter o número de *bits* de partes do operando, com a adição sequencial do número de *bits* iguais a 1 de cada uma das partes em que o operando é dividido.

```
; conta o numero de bits iguais a um
; AX contem o parametro de entrada
; retorna em DL o numero de bits iguais a um
            xor   DL,DL         ; contador
ciclo:      test  AX,01h        ; testa LSbit
            jz    naosoma
            inc   DL            ; incrementa se for 1
naosoma:    shr   AX,1          ; desloca para a direita
            and   AX,AX
            jnz   ciclo         ; repete ate AX = 0
```

(a)

```
; conta o numero de bits iguais a um
; AX contem o parametro de entrada
; retorna em DL o numero de bits iguais a um
; TABELA tem 65536 elementos e cada um   igual ao
; numero de bits iguais a 1 do indice
TABELA:     DB    0,1,1,2,1,...,15,16

            mov   SI,AX             ; indice na tabela
            mov   DL,TABELA[SI]     ; recupera o valor
```

(b)

Figura 4.7: Código *assembly* de uma função para contar *bits* iguais a 1 de uma palavra de 16 *bits*, analisando os *bits* em sequência (a) e usando uma tabela de consulta (b).

Uma implementação desta operação em *hardware* pode ser realizada como um circuito paralelo ou série, dependendo de ser ou não conhecida previamente a dimensão N do

operando. Conhecido N, pode ser construído um circuito paralelo como uma árvore com $log_2(N)$ andares de circuitos somadores, produzindo como resultado $log_2(N) + 1$ *bits*. O primeiro andar é constituído por $N/2$ meios-somadores (somadores de números de um *bit*) e soma *bits* dois a dois. O segundo andar tem $N/4$ somadores de 2 *bits*, o terceiro $N/8$ somadores de 3 *bits* e o k-ésimo andar tem $N/(2^k)$ somadores de k *bits*. Na figura 4.8-(a) mostra-se o circuito lógico deste operador, para operandos de 8 *bits*.

Uma realização série pode ser obtida com um contador e um registo de deslocamento com saída em série, onde é colocado o operando. Deslocando os *bits* sincronamente com um sinal de relógio e usando a saída do registo de deslocamento como sinal de inibição de um contador actuado pelo mesmo sinal de relógio, obtém-se um circuito série que calcula o número de *bits* iguais a 1 em N ciclos de relógio (fig. 4.8-(b)).

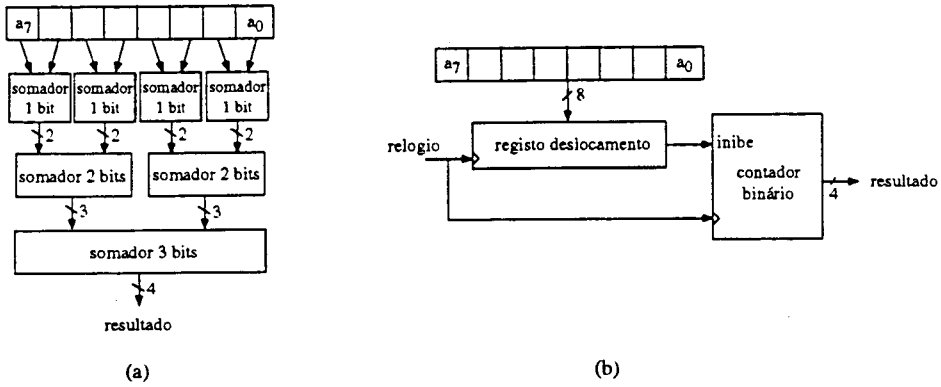


Figura 4.8: Circuito lógico paralelo (a) e série (b) de um operador dedicado para contar *bits* de uma palavra de 8 *bits*.

Na tabela 4.1 comparam-se os resultados obtidos para cada implementação, considerando operandos de 16 *bits*. Pode ver-se que a implementação mais rápida desta função em *software*, recorrendo a uma tabela de consulta, gasta aproximadamente metade do tempo necessário ao operador mais rápido em *hardware* paralelo, embora à custa da utilização de um número elevado de posições de memória. Por outro lado, o operador em *hardware* série tem de trabalhar com uma frequência de relógio de 58 MHz (1/17 ns) para atingir uma rapidez dupla da conseguida pela versão em *software* iterativo. Pode ver-se também que a implementação do operador em *hardware* série nesta família de FPGA apenas reduz em cerca de 24% o espaço ocupado pela versão em *hardware* paralelo.

Tabela 4.1: Comparação das implementações em *software* e em *hardware* de operadores para contar o número de *bits* iguais a 1 em palavras de 16 *bits*.

Operador	Espaço	Tempo (ns)	Rapidez ($\times 10^6$ operações/seg)
SW iterativo	15 *bytes*	550	1.8
SW tabela	65536+6 *bytes*	20	50
HW paralelo	41 CLBs	38	26
HW série	31 CLBs	272 (17 ns/*bit*)	3.7

4.4.2 Localização de uma sequência de *bits* num vector

Dado um vector de N *bits*, pretende-se determinar a posição em que nela ocorre uma sequência de M *bits*(com $M \leq N$). Por exemplo, pesquisar a sequência 1011 no vector 1101000010111001 deve dar como resultado 8, contando posições desde a esquerda e a partir de zero. Uma implementação em *software* requer, no pior dos casos, $N - M + 1$ iterações para comparar a sequência a pesquisar com todas as posições que ela pode ocupar no vector pesquisado. Na figura 4.9 apresenta-se um segmento de código *assembly* que realiza essa função, considerando como operandos um vector com 16 *bits* e a sequência a pesquisar com 4 *bits*.

```
; pesquisa uma palavra de 4 bits (4 LSbits de BX)
; num vector de 16 bits dado em AX
; retorna em DX a posicao da sequencia no vector
; ou -1 se nao encontrar
            mov   si,ax         ; guarda vector
            mov   cx,000Fh      ; CX = 0000000000001111
            mov   dx,12         ; posicao inicial
ciclo:      mov   ax,si         ; recupera vector
            and   ax,cx         ; mascara
            xor   ax,bx         ; compara vector com
            jz    fim           ; se igual, termina
            dec   dx            ;
            test  dx,08000h     ; se dx != -1, continua
            jnz   fim
            shl   cx,1          ; desloca mascara
            shl   bx,1          ; desloca sequencia
            jmp   ciclo
fim:
```

Figura 4.9: Código *assembly* de uma função para pesquisar uma sequência de 4 *bits* num vector de 16 *bits*. O valor retornado é -1 se a sequência pesquisada não existir, ou se for encontrada, a posição em que ocorre.

Uma implementação em *hardware* pode ser obtida como uma função combinacional dos dois operandos, usando em paralelo $N - M + 1$ comparadores de igualdade de M *bits* e um codificador de prioridade para calcular a posição da sequência encontrada. Na figura 4.10-(a) mostra-se o circuito lógico paralelo que implementa esse operador para

pesquisar uma sequência de 4 *bits* num vector de 8 *bits*. Uma realização série requer um só comparador de igualdade de M *bits* e um contador, mas realiza a operação em $N - M + 1$ ciclos de relógio (fig. 4.10-(b)). Na tabela 4.2 comparam-se as complexidades e desempenhos dos 3 operadores referidos. Pode ver-se que a implementação em *software* necessita de 1.5 vezes o tempo gasto pela versão mais lenta do operador em *hardware* e 10 vezes mais do que a realização mais rápida em *hardware* paralelo.

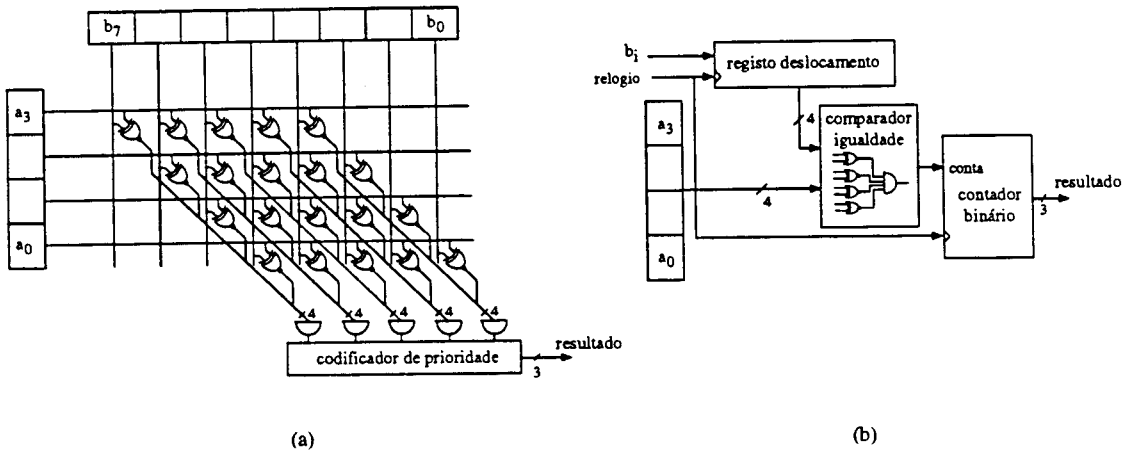


Figura 4.10: Circuitos lógicos de operadores para pesquisar uma sequência de 4 *bits* $(a_3 \ldots a_0)$ num vector de 8 *bits* $(b_7 \ldots b_0)$: circuito paralelo (a) e circuito série (b)

Tabela 4.2: Comparação das implementações em *software* e em *hardware* de operadores para pesquisar uma sequência de 4 *bits* num vector de 16 *bits*.

Operador	Espaço	Tempo (ns)	Rapidez ($\times 10^6$ operações/seg)
SW	29 *bytes*	400	2.5
HW paralelo	41 CLBs	40	25
HW série	20 CLBs	$12 \times 22 = 264$	3.8

4.4.3 Gerador de assinaturas

Um método usado para compactar a resposta de um circuito digital num ambiente de teste com vista à detecção de faltas, consiste em calcular uma assinatura da resposta do circuito a um conjunto de estímulos seleccionados para esse fim. Uma forma de obter a assinatura de uma sequência de *bits* baseia-se na determinação de códigos CRC (*Cyclic*

Redundancy Check), calculados com um LFSR (*Linear Feedback Shift-Register*) sobre o vector de *bits* que representa a resposta do circuito sob teste.

```
C0 = 0;
C1 = 0;
for(i=M-1; i>=0; i++)
{
      T = C1 ^ A[i];
      C1 = C0 ^ F1[T];
      C0 = F0[T];
}
```

(a)

```
; Calcula em CX a assinatura de 16 bits de um vector de Mx8 bits
; lidos do vector A[]. As tabelas de consulta usadas pelo
; algoritmo sao F0[] e F1[]. BX conta o numero de bytes do
; vector de entrada.
            mov   CX,0          ; LSFR = 0 (C1 = CH e C0 = CL)
            xor   AH,AH         ; limpa AH
            mov   BX,M          ; numero de bytes do vector de entrada
            mov   SI,BX         ; SI = indice no vector de entrada
ciclo:      dec   SI            ; SI = M-1,...,0
            mov   AL,CH
            xor   AL,A[SI]      ; T = CH xor A[i] (T=AX)
            mov   DI,AX         ; DI = T
            mov   CH,CL
            xor   CH,F1[DI]     ; CH = CL xor F1[T]
            mov   CL,F0[DI]     ; CL = F0[T]
            dec   BX            ; repete ate BX=0 (M vezes)
            jnz   ciclo
```

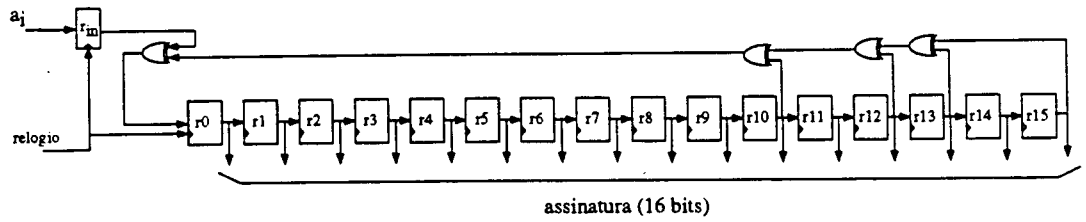
(b)

Figura 4.11: Uma função para calcular uma assinatura de 16 *bits* recorrendo a tabelas de consulta. O vector de entrada é dado *byte* a *byte* no vector `A[]` e as tabelas de consulta são `F0[]` e `F1[]`. Em (a) mostra-se uma função em C e em (b) apresenta-se uma versão escrita directamente em *assembly*.

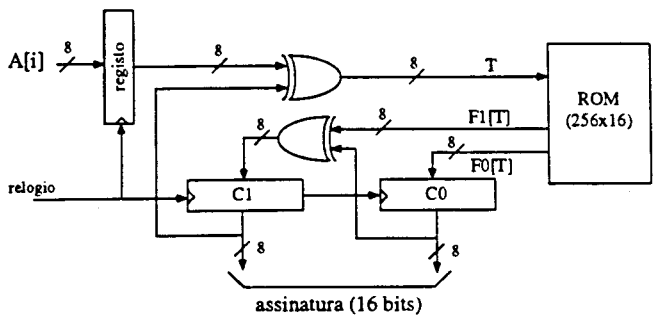
A implementação em *software* de um LFSR resulta ineficiente porque obriga à utilização de operações sobre os *bits* individuais do vector. Em [SS92] são propostas técnicas para o cálculo eficiente de assinaturas, usando tabelas de consulta que podem ser construídas sem conhecimento prévio dos dados. No entanto, esses algoritmos resultam ineficientes para calcular assinaturas com um número de *bits* que não seja um múltiplo de 8, porque obrigam também à execução de operações para trabalhar individualmente com *bits*. Na figura 4.11 mostra-se uma função escrita em C (a) e em *assembly* (b) que implementa um LFSR de 16 *bits* segundo o algoritmo proposto em [SS92]. O vector de entrada é constituído por M *bytes*, dados no vector `A[i]`, e as tabelas de consulta são `F0[]` e `F1[]`, tendo cada uma 256 *bytes*. O código da figura 4.11-(b) foi escrito directamente em *assembly* e não foi obtido por compilação do código C mostrado na mesma figura. É interessante notar que para o mesmo vector de dados de entrada, a função em *assembly* gasta apenas 56% do tempo de execução da mesma função codificada em C e compilada

com o compilador de C da Microsoft (MSC 6.0), com optimização para rapidez.

A realização em *hardware* de um gerador de assinatura pode ser feita usando um LFSR (*Linear-Feedback Shift Register*), construído com um registo de deslocamento realimentado em pontos determinados com portas lógicas OU-EXCLUSIVO, como é mostrado na figura 4.12 (a). Este operador pode ser expandido para calcular assinaturas com qualquer número de *bits*, sem que isso se traduza numa degradação da sua rapidez de funcionamento, medida pelo número máximo de *bits* que podem ser processados por unidade de tempo.



(a)



(b)

Figura 4.12: Circuitos lógicos de um detector de assinatura de 16 *bits*. O circuito (a) é um LFSR (*Linear-Feedback Shift Register*) de 16 *bits*. O circuito (b) foi obtido por tradução para *hardware* da implementação em *software* apresentada na figura 4.11.

Uma implementação mais eficiente pode ser obtida traduzindo para *hardware* a função mostrada na figura 4.11. O operador usa tabelas de consulta previamente construídas e armazenadas em memória ROM, processando um *byte* do vector de entrada em cada ciclo de relógio. Para assinaturas mais longas, embora com um número de *bits* múltiplo de 8, o circuito pode ser expandido à custa de um acréscimo da complexidade do circuito lógico que varia de forma linear com o número de *bytes* da assinatura. Por exemplo, um LFSR de 32 *bits* necessita aproximadamente do dobro dos recursos utilizados pelo gerador de

assinatura de 16 *bits* mostrado na figura 4.12-(b). Além disso, pode também processar um *byte* num ciclo de relógio aproximadamente igual ao requerido pela implementação de 16 *bits*[1].

Na tabela 4.3 são comparados os desempenhos das realizações em *software* e em *hardware* de um gerador de assinatura de 16 *bits*, operando sobre um vector de 128 *bytes* (1024 *bits*). O espaço ocupado pela versão em *hardware* paralelo (*byte* a *byte*) inclui a memória ROM que contém as duas tabelas de consulta. Pode ver-se que a realização em *hardware* série (*bit* a *bit*) gasta aproximadamente o dobro do tempo necessário à implementação em *software* optimizada em *assembly*, e requer uma frequência de relógio igual a 52 MHz. O operador em *hardware* paralelo funcionando com a frequência máxima de 22 MHz consegue reduzir para metade o tempo consumido pela versão optimizada em *software*.

Tabela 4.3: Comparação das implementações em *software* e em *hardware* de operadores para calcular assinaturas de 16 *bits* de um vector com 1024 *bits*.

Operador	Espaço	Tempo (ns)	Tempo/*bit* (ns)	$\times 10^6$ *bits*/seg.
SW	2x256+33 *bytes*	10870	10.6	94
HW série	26 CLBs	19456	19.0 (52 MHz)	51
HW paralelo	179 CLBs	5837	5.7 (22 MHz)	171

4.4.4 Operações com constantes

Operações que envolvam operandos constantes podem ser realizadas de forma eficiente com operadores dedicados, obtidos à custa de simplificações resultantes do conhecimento prévio dessas constantes. Isto permite obter circuitos lógicos para operadores dedicados que sejam mais rápidos e menos complexos do que operadores genéricos.

Transformações semelhantes são realizadas por compiladores optimizadores de linguagens de alto nível, traduzindo certos tipos de operações por constantes em sequências de outras instruções de um processador, que conduzem a uma execução mais rápida. Um exemplo de transformações deste tipo ocorre para multiplicações ou divisões de números

[1] Ciclos de relógio iguais ocorrem se for considerado apenas o atraso dos elementos lógicos do circuito. Como num circuito FPGA não são desprezáveis os atrasos introduzidos pelas interligações programáveis. um operador para assinaturas de 32 *bits* necessitará, em princípio, de um ciclo de relógio mais longo já que envolve um número superior de elementos de interligação programáveis entre os seus elementos.

inteiros por potências inteiras de 2, normalmente substituídas por operações de deslocamento de *bits* que são mais rápidas na generalidade dos processadores convencionais.

Comparação de caracteres

Uma aplicação onde são exploradas simplificações resultantes da propagação de constantes para *hardware* realizado em FPGAs é apresentada em [Fou93]. Neste trabalho foi usado um sistema de prototipagem baseado em FPGAs da Algotronix (CHS2x4 [KB92]) para implementar em *hardware* operadores dedicados para pesquisar sequências de caracteres em textos. Uma vez definida a sequência a pesquisar, podem ser construídos circuitos comparadores dedicados, que são mais simples e mais rápidos do que comparadores genéricos. Uma realização de um sistema para detectar uma sequência de 8 palavras quaisquer de 16 caracteres cada, acrescida de caracteres adicionais para representar um contexto (fim de linha, parágrafo ou secção) necessita de 5 circuitos CAL1024 (4505 células). Se for conhecida a sequência de caracteres a pesquisar e forem propagadas para *hardware* as constantes que os representam, é possível realizar o mesmo sistema com apenas 2 circuitos daquele tipo (1911 células). A utilização dos circuitos programáveis CAL1024 possibilita a construção de um sistema inicial que implementa apenas comparadores com zeros, sendo modificada parcialmente a sua configuração quando é especificada a sequência de caracteres a pesquisar, reconfigurando as células apropriadas dos FPGAs.

Na figura 4.13 comparam-se os circuitos lógicos que implementam a comparação de dois *bytes*, com 8 portas lógicas NÃO-OU-EXCLUSIVO e uma porta E de 8 entradas (a), e em (b) o circuito necessário para comparar um *byte* com a constante 10010100, que é constituído por uma porta E de 8 entradas, algumas das quais são negadas. Naturalmente que o ganho real em complexidade depende da tecnologia de implementação que for empregue na realização daqueles circuitos.

Um simplificação deste tipo foi aplicada ao operador apresentado na secção 4.4.2 para pesquisar o padrão 1010 num vector com 16 *bits*. Para isso, os 12 comparadores daquele operador foram substituídos por comparadores por aquela constante, resultando num circuito mais pequeno e mais rápido, como mostram os resultados apresentados na tabela tabela 4.4. Fixar o padrão de 4 *bits* a pesquisar permite reduzir para metade o número dos recursos configuráveis do FPGA necessários à implementação dos 12 comparadores, o que representa uma redução máxima de 6 CLBs[2]. No entanto esse número apenas foi reduzido em 5, o que resulta do facto de implementações em FPGA nem sempre usarem a totalidade dos recursos configuráveis disponíveis nos CLBs ocupados.

[2]Em cada CLB desta família de FPGA pode ser implementado um comparador de igualdade de dois números de 4 *bits* ou dois comparadores de igualdade de um número de 4 *bits* por uma constante conhecida

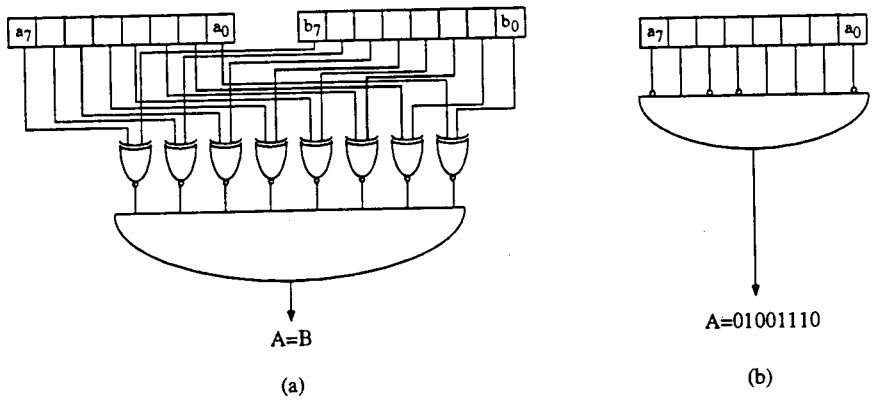


Figura 4.13: Circuito lógico para comparação de dois caracteres (a) e para comparação de um carácter com uma constante (b).

Tabela 4.4: Comparação entre as implementações do operador em *hardware* paralelo para pesquisar uma sequência genérica de 4 *bits* e para pesquisar o padrão 1010 num vector de 16 *bits*.

Operador	Espaço	Tempo (ns)	Rapidez ($\times 10^6$ operações/seg)
HW paralelo (original)	41 CLBs	39	26
HW paralelo (constante)	36 CLBs	36	28

Multiplicação por constantes

Um tipo de operação onde podem ser conseguidos ganhos importantes de rapidez e de redução de complexidade são multiplicações inteiras por constantes que possam ser transformadas em combinações de deslocamentos de *bits*, adições e subtracções. Dependendo da constante envolvida, podem ser conseguidos multiplicadores dedicados substancialmente mais simples e rápidos do que multiplicadores genéricos, embora tenham que ser construídos especificamente para cada uma das constantes. Para constantes inteiras inferiores ou iguais a 42, a multiplicação é simplificada para um máximo de duas adições ou subtracções, não contando com os deslocamentos de *bits* uma vez que não requerem qualquer elemento lógico. Na figura 4.14 mostra-se como podem ser decompostas as multiplicações por cada uma daquelas constantes, indicando-se também o número de andares de circuitos somadores ou subtractores necessários para a construção desses operadores.

Na figura 4.15 exemplificam-se as decomposições da multiplicação pelas constantes 25 e 39, identificadas na figura 4.14 a carregado e os respectivos circuitos lógicos, considerando um operando de 8 *bits*. Os resultados das realizações destes multiplicadores em FPGA são

C	decomposição x * C	andares soma/subt
1	x	0
2	x << 1	0
3	x << 1 + x	N
4	x << 2	0
5	x << 2 + x	N
6	x << 2 + x << 1	N
7	x << 3 - x	N+3
8	x << 3	0
9	x << 3 + x	N
10	x << 3 + x << 1	N
11	x << 3 + x << 2 - x	N+N+4
12	x << 3 + x << 2	N
13	x << 3 + x << 2 + x	N+N+2
14	x << 4 - x << 1	N+3
15	x << 4 - x	N+4
16	x << 4	0
17	x << 4 + x	N
18	x << 4 + x << 1	N
19	x << 4 + x << 1 + x	N+N+4
20	x << 4 + x << 2	N
21	x << 4 + x << 2 + x	N+N+3

C	decomposição x * C	andares soma/subt
22	x << 4 + x << 2 + x << 1	N+N+2
23	x << 4 + x << 3 - x	N+N+5
24	x << 4 + x << 3	N
25	**x << 4 + x << 3 + x**	**N+N+2**
26	x << 4 + x << 3 + x << 1	N+N+2
27	x << 5 - x << 2 - x	N+3+N+6
28	x << 5 - x << 2	N+3
29	x << 5 - x << 1 - x	N+4+N+6
30	x << 5 - x << 1	N+4
31	x << 5 - x	N+5
32	x << 5	0
33	x << 5 + x	N
34	x << 5 + x << 1	N
35	x << 5 + x << 1 + x	N+N+5
36	x << 5 + x << 2	N
37	x << 5 + x << 2 + x	N+N+4
38	x << 5 + x << 2 + x << 1	N+N+4
39	**x << 5 + x << 3 - x**	**N+N+6**
40	x << 5 + x << 3	N
41	x << 5 + x << 3 + x	N+N+3
42	x << 5 + x << 3 + x << 1	N+N+3

Figura 4.14: Decomposição de multiplicações por constantes positivas inferiores a 42 em deslocamentos de *bits*, (representado pelo operador <<), adições e subtracções. Para cada decomposição indica-se o número de andares somadores ou subtractores necessários, em função do número de *bits* do operando, N.

apresentados na tabela 4.5, e comparam-se com uma implementação de um multiplicador paralelo para operandos de 8 *bits*, produzindo um resultado de 16 *bits*. Os resultados apresentados mostram os ganhos importantes em rapidez e espaço ocupado que foram conseguidos pelos operadores dedicados a operandos constantes.

Tabela 4.5: Comparação da implementação de multiplicadores de números de 8 *bits* pelas constantes 25 e 39 com um multiplicador paralelo para operandos de 8 *bits*, produzindo um resultado de 16 *bits*.

Multiplicador	Resultado	Espaço (CLBs)	Atraso (ns)	Frequência (MHz)
8 *bits* por 25	13 *bits*	20	31	32
8 *bits* por 39	14 *bits*	24	35	28
8 × 8 *bits*	16 *bits*	75	82	12

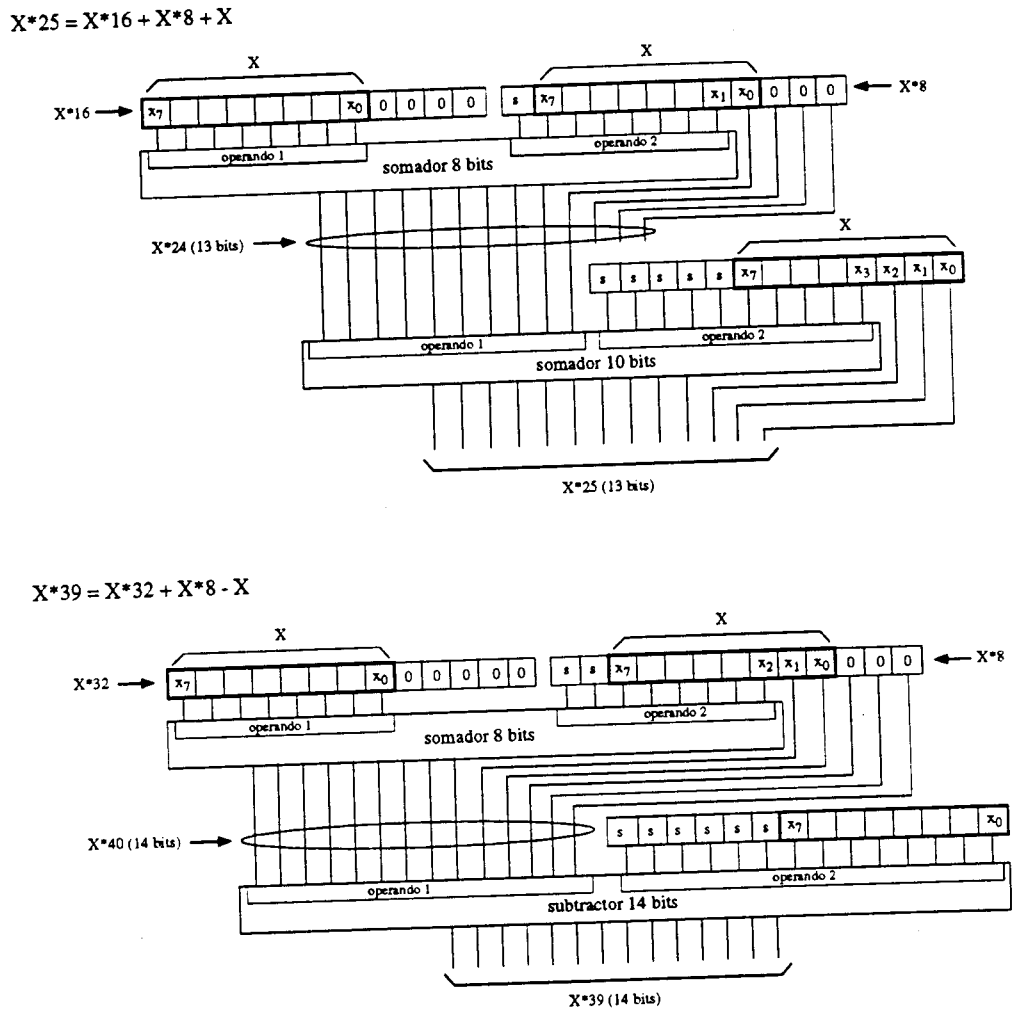


Figura 4.15: Decomposição de multiplicações pelas constantes 25 e 39 e respectivos circuitos lógicos combinacionais.

4.4.5 Um estudo detalhado—aceleração de rotinas gráficas

Um exemplo de uma sequência de operações que sugeriu a implementação num operador dedicado foi extraído do sistema gráfico Atega para PCs desenvolvido pelo autor [Alv89]. O Atega é uma biblioteca de rotinas gráficas eficientes para controladores gráficos EGA e VGA, que foram em grande parte desenvolvidas em linguagem *assembly* para conseguir atingir a máxima eficiência. De forma a satisfazer esse objectivo, as várias rotinas incluídas nesta biblioteca não utilizam as funções disponíveis no BIOS (*Basic Input Output System*) do sistema operativo MSDOS para interface com o controlador gráfico, e funcionam antes

com programação directa dos seus registos e escrita na área de memória vídeo.

Este trabalho foi motivado, na altura do seu desenvolvimento, pela falta de bibliotecas de rotinas gráficas de fácil utilização para programação em PCs. Este sistema foi usado com sucesso no desenvolvimento de várias ferramentas de *software* para edição e manutenção de desenhos de cartas de redes eléctricas de média e baixa tensão, desenvolvidos para a EDP—Electricidade de Portugal [MMF+90]. Embora actualmente a programação de aplicações gráficas em PCs seja feita, na generalidade das situações, usando ambientes de programação integrados no sistema operativo Windows e que permitem a construção interactiva de interfaces gráficos, O Atega permite ainda construir aplicações gráficas eficientes para aquele tipo de carta gráfica, que é ainda suportada por um grande número dos controladores gráficos actuais para PCs.

Identificação de um operador dedicado

Um segmento de código que é intensivamente utilizado pela generalidade das rotinas gráficas serve para calcular o endereço físico na memória vídeo e uma máscara a usar nas operações de escrita nessa memória, para desenhar o ponto de imagem (*pixel*) com coordenadas (x, y). Para os modos gráficos com 640 × 350 e 640 × 480 *pixel* a memória vídeo é organizada da forma apresentada na figura 4.16: cada *pixel* corresponde a um *bit* de um *byte* nesse espaço de memória e para o desenhar é necessário conhecer o endereço físico desse *byte* e a posição do *bit* correspondente dentro desse *byte*.

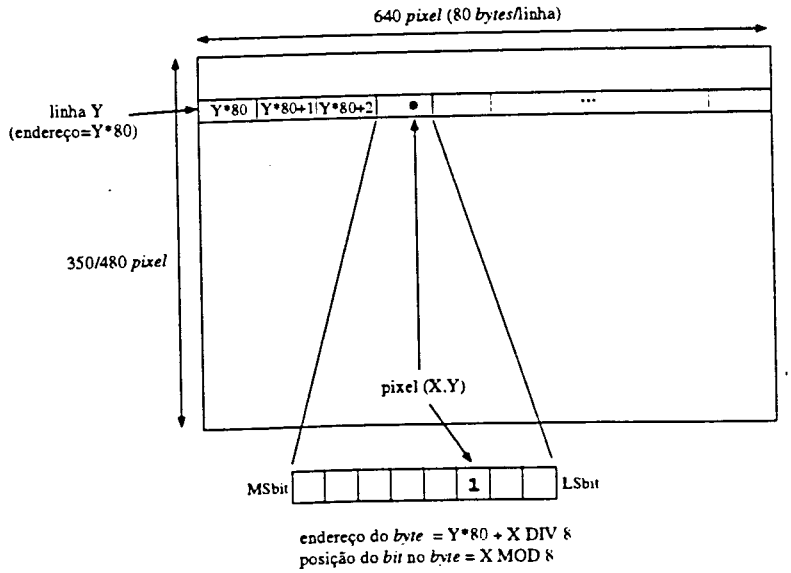


Figura 4.16: Organização da memória vídeo nos controladores gráficos EGA e VGA.

Dadas as coordenadas (x, y) de um ponto de imagem, o endereço do *byte* a escrever, relativo ao início do espaço de endereçamento ocupado pela memória vídeo, é calculado como:

```
addr = y*80 + x DIV 8 = y*64 + y*16 + x DIV 8
```

e a máscara que selecciona o *bit* desse *byte* que corresponde ao *pixel* (x, y) é obtida por:

```
mask = 080h >> ( x MOD 8 )
```

onde `DIV` e `MOD` representam o quociente e o resto da divisão inteira, respectivamente, e o operador `>>` representa o deslocamento para a direita dos *bits* do primeiro operando, de um número de posições igual ao valor do segundo operando.

Embora esta operação ocorra na generalidade das funções gráficas deste sistema, para desenvolver e experimentar um operador dedicado para esta operação elegeu-se a função `Plot(x,y,cor)` que desenha um ponto de imagem, por ser a mais simples que envolve aquela sequência de operações. Nesta rotina, aquelas operações são feitas pelo conjunto das 12 instruções máquina escritas a carregado na listagem da função original que se apresenta na figura 4.17.

Para realizar aquela sequência de operações num operador dedicado que não use variáveis na memória do PC, é necessário reescrever a secção de código referida de forma a obter um único bloco básico de instruções que utilize apenas registos do processador (fig. 4.18). O bloco resultante contém 14 instruções máquina, recebe os argumentos nos registos `AX` e `BX` e retorna o resultado nos registos `DI` (endereço na memória vídeo) e `AX` (`AH`=máscara, `AL`=08h).

Aquela sequência de instruções sugere ser adequada para ser realizada num operador dedicado, já que o tipo de operações envolvidas (adições e deslocamentos de *bits*) podem ser efectuadas de forma eficiente e requerendo reduzida complexidade de *hardware*. As duas funções que representam os cálculos realizados por aquele bloco de instruções podem ser avaliadas em paralelo, uma vez que não há qualquer dependência entre elas. Além disso, todas as multiplicações e divisões são feitas sobre constantes iguais a potências inteiras de 2, o que permite implementá-las como operações de deslocamento de *bits* que não consomem recursos de *hardware*.

Realização prática do operador dedicado

O circuito lógico do operador desenvolvido apresenta-se na figura 4.19. A máscara é obtida ligando os 3 *bits* menos significativos de `x` (o resto da divisão inteira de `x` por 8)

```
_Plot   PROC FAR
        PUSH BP
        MOV  BP,SP
        PUSH DI
        MOV  BH,[BP+10]          ; get color
        MOV  AH,BH
        SHR  AH,1
        AND  AH,00011000B        ; program function bits
        MOV  AL,DATROT
        MOV  DX,GRAPH
        CALL OUTDX
        MOV  AH,BH               ; program pixel color
        MOV  AL,SRES
        CALL OUTDX

        MOV  AX,[BP+6]           ; get X
        MOV  BL,08H
        DIV  BL                  ; AH=bit position, AL=byte offset
        MOV  CL,AH
        MOV  AH,80H
        SHR  AH,CL               ; AH=bitmask
        MOV  CL,AL
        MOV  AL,BITMASK
        CALL OUTDX               ; program bitmask register

        MOV  AX,[BP+8]           ; get Y
        MOV  BX,80
        MUL  BX
        MOV  DI,AX               ; AX=Y*80
        MOV  CH,00H
        ADD  DI,CX               ; DI=byte offset in vram buffer

        MOV  AX,BMAP
        MOV  ES,AX               ; bitmap segment
        MOV  AL,ES:[DI]          ; latch data
        MOV  ES:[DI],AL          ; write data
        POP  DI
        POP  BP
        RET
_Plot   ENDP
```

Figura 4.17: Listagem da função `Plot(x,y,cor)` do sistema Atega para desenho de um ponto de imagem. As instruções que efectuam os cálculos referidos no texto estão escritas a carregado. A rotina `OUTDX` envia os conteúdos de `AL` e `AH` para portas do espaço de endereçamento de entrada e saída com endereços dados em `DX` e `DX+1`.

às entradas de um circuito descodificador de 3 para 8. Transformando a multiplicação de y pela constante 80 em somas e deslocamentos de *bits*, obtém-se o endereço da área de memória vídeo com apenas dois circuitos somadores, um para calcular y*80 e o outro para adicionar esse valor ao quociente da divisão inteira de x por 8. Como alguns dos *bits* de entrada dos somadores são zero, são apenas necessários 10 andares somadores completos (FA) sendo os restantes 11 meios-somadores (HA).

Este operador foi implementado num FPGA XILINX XC4010-4 do sistema reconfigurável ISAX (secção 2.3.5 do capítulo 2). O interface com o processador principal é realizado através de instruções de entrada e saída de 16 *bits*, sendo por isso necessário executar duas instruções `OUT` para enviar os dois operandos (x e y) e duas instruções `IN` para recuperar os resultados. Para utilizar o mesmo endereço de entrada e saída para estas 4 operações, foi construído um circuito que, por cada instrução de escrita ou de leitura selecciona as portas apropriadas do operador.

```
...
MOV   AX,[BP+6]        ; get X
MOV   BX,[BP+8]        ; get Y

MOV   CL,08H
DIV   CL               ; AH = X MOD 8, AL = X DIV 8
MOV   CL,AH
MOV   AH,80H
SHR   AH,CL            ; AH = bitmask
MOV   CH,AH            ;
MOV   CL,AL            ; CL = byte offset in scan line
MOV   AX,80
MUL   BX               ; AX = Y * 80
MOV   DI,AX
MOV   AH,CH            ; bitmask
MOV   AL,08h           ; bitmask register code
MOV   CH,00h           ; CX = byte offset in scan line
ADD   DI,CX            ; DI = offset in vram buffer

CALL  OUTDX            ; program bit mask register
MOV   AX,BMAP
MOV   ES,AX            ; bitmap segment
MOV   AL,ES:[DI]       ; latch data from vram
MOV   ES:[DI],AL       ; write data
...
```

Figura 4.18: Função `Plot(x,y,cor)` modificada para isolar o bloco básico de instruções que efectua a sequência de operações seleccionada.

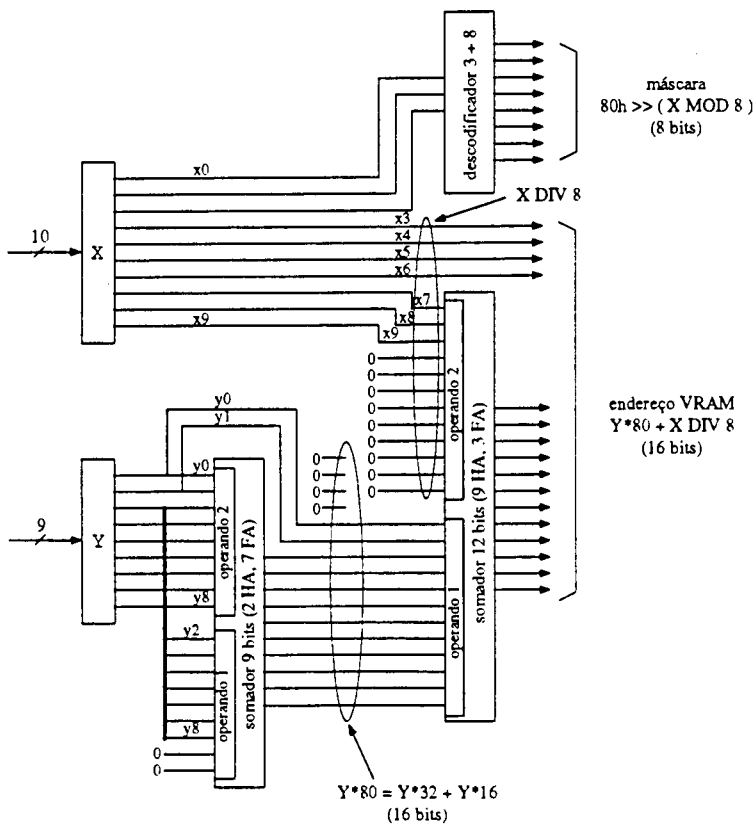


Figura 4.19: Uma implementação do operador dedicado construído para cálculo de endereços da memória vídeo para o sistema Atega.

O circuito completo, incluindo a interface com o barramento ISA, utiliza menos de 12% dos blocos lógicos configuráveis do FPGA XC4010-4 (47 de um total de 400 CLBs) e apresenta um tempo de atraso da lógica combinacional entre registos inferior a 43 ns. Este tempo estabelece assim o número máximo de operações daquele tipo que o operador pode realizar por unidade de tempo: 23.2×10^6 operações por segundo. Considerando que essa função equivale a 4 operações aritméticas (uma multiplicação, uma adição, uma divisão e um deslocamento de *bits*), pode afirmar-se que o desempenho máximo atingido por esta implementação em FPGA é 92.8 MOPs. Finalmente, se esse tempo for comparado com os 320 ns necessários para um PC-Pentium a 120MHz executar as 14 instruções do segmento de código da figura 4.18 (tomando a média dos tempos consumidos por 10^7 execuções daquele bloco de código), pode traduzir-se esse aumento de rapidez em cerca de 7.4 vezes. No entanto, este ganho só seria possível se o bloco original de 14 instruções máquina fosse substituído por uma única instrução que invocasse aquele operador e que gastasse apenas os 43 ns correspondentes ao tempo de atraso do circuito lógico. Para tal seria necessário que o operador dedicado fosse interno ao processador, de modo a ter acesso directo aos registos que contêm os operandos e onde são colocados os resultados. Numa realização como um circuito externo é naturalmente impossível conseguir aquele valor.

Numa avaliação prévia do desempenho daquele operador na função em que é utilizado é também preciso avaliar que percentagem do tempo de execução da rotina é gasto pelo bloco de instruções seleccionado para realizar no operador dedicado. Apesar do número referido antes obtido por comparação dos tempos de execução das duas alternativas de implementação, a medida que realmente interessa é a redução do tempo total de execução na função que desenha um ponto de imagem. Medindo o tempo médio gasto pela função original obteve-se $5.0\mu s$, e eliminando aquele segmento de 14 instruções o tempo de execução é apenas reduzido em $0.32\mu s$, o que representa só 6.4% do tempo de execução original. Isto quer dizer que, à custa de realizar em *hardware* aquela sequência de operações, é impossível reduzir em mais de 6.4% o tempo total de execução da função `Plot()`, o que conduz a um ganho máximo em rapidez que não poderá ultrapassar 1.07 vezes.

Naturalmente que para se poder avaliar o ganho de desempenho real tem se contabilizar também o tempo necessário para passar os argumentos e resultados entre o processador principal e o processador auxiliar. Como foi já referido antes, esta operação pode requerer um tempo de tal forma longo que anule o ganho resultante da utilização do operador dedicado. Esse tempo tem que ser medido considerando a sequência real de operações necessárias para efectuar a passagem de dados. É necessário ter em conta que o processador principal não está apenas a correr a aplicação do utilizador, mas está periodicamente sujeito a interrupções pelo sistema operativo, que consomem tempo e alteram os conteúdos

das memórias *cache*. Medidas realistas podem ser obtidas calculando a média dos tempos necessários a um número elevado de execuções, em condições semelhantes àquelas em que se pretende que a aplicação venha a ser utilizada.

Na aplicação estudada, um cálculo preliminar permite concluir que, devido ao fraco desempenho do barramento ISA, não é possível conseguir qualquer ganho de rapidez naquela rotina, recorrendo ao operador dedicado implementado naquele sistema reconfigurável. O tempo mínimo necessário para transferir os 4 valores de 16 *bits* que representam os operandos e os resultados, com instruções de entrada e saída pelo barramento ISA equivale a 12 ciclos do relógio de 8 MHz do barramento, o que é igual a $1.5\mu s$ e representa cerca de 4.7 vezes o tempo gasto pela execução da sequência original de 14 instruções máquina. Como estas transferências resultam da execução de instruções de entrada e saída no processador principal, é necessário acrescentar ainda o seu tempo de execução, assim como o de instruções adicionais que foi necessário incluir para mover dados e resultados para os registos apropriados esperados pelas instruções de entrada e saída.

Para invocar o operador construído, a função `Plot()` da figura 4.18 foi modificada de forma a escrever os dados e ler os resultados calculados pelo operador (figura 4.20). Com esta versão da função, o número de instruções máquina dedicadas àquela operação foi reduzido de 14 para 8, embora se tenha com isso eliminado uma divisão e uma multiplicação. Como o tempo de atraso do operador é francamente inferior ao ciclo de acesso do barramento, não foi preciso incluir estados de espera entre a última instrução de escrita dos operandos e a primeira de leitura dos resultados.

```
...
MOV   AX,[BP+6]        ; get X
MOV   BX,[BP+8]        ; get Y

MOV   DX,0300h         ; board base address
OUT   DX,AX            ; send Y
MOV   AX,BX
OUT   DX,AX            ; send X
IN    AX,DX            ; read vram address
MOV   DI,AX
IN    AX,DX            ; AL = bitmask register code,  AH = data
MOV   DX,GRAPH

CALL  OUTDX            ; program bitmask register
MOV   AX,BMAP
MOV   ES,AX            ; bitmap segment
MOV   AL,ES:[DI]       ; latch data from vram
MOV   ES:[DI],AL       ; write data
...
```

Figura 4.20: Função `Plot()` modificada para invocar o operador dedicado.

Resultados

Apesar de ser sabido antecipadamente que não seria conseguido qualquer ganho de rapidez com a realização do operador dedicado naquele sistema, foram comparados os tempos gastos entre a versão original daquela rotina e a versão modificada invocando o operador dedicado. Verificou-se que, recorrendo ao operador em *hardware* era necessário aproximadamente o dobro do tempo gasto pela função original, e que a quase totalidade desse acréscimo de tempo era gasto na execução das 4 instruções `IN` e `OUT` para comunicar com o operador dedicado. Na tabela 4.6 são resumidos os resultados obtidos com esta aplicação. Os tempos apresentados foram medidos num PC-Pentium a 120 MHz, com recurso à função `clock()` da biblioteca do compilador de C (Microsoft C 6.0) e são a média de 10 corridas de uma função que preenche o monitor com 307200 *pixels*, correspondente ao ecran completo em modo VGA (640 × 480 *pixel*). Na mesma tabela mostram-se também os ganhos que foram conseguidos nos tempos de execução daquela função, resultantes de transformações simples ao seu código original. Trocando a multiplicação e a divisão por operações de deslocamento de *bits* o tempo de execução é reduzido em 3.4%, e a substituição da chamada à subrotina `OUTDX` pelo seu código consegue uma redução de 10.7%. É de notar que a segunda modificação referida conduz a uma redução superior aos 6.4% do tempo de execução gastos pelo bloco de instruções que foi seleccionado para ser realizado pelo operador em *hardware*.

Tabela 4.6: Comparação dos tempos de execução para diferentes implementações da função `Plot()` do sistema Atega.

	tempo (μs)	tempo/tempo função original
função `Plot()` original	5.0	1.0
invocando operador em *hardware*	10.4	2.1
Modificações à função original		
trocando `MUL` e `DIV` por *shifts*	4.8	0.96
substituindo `OUTDX` pelo seu código	4.5	0.90

O estudo apresentado torna oportuno tecer algumas considerações finais. Em primeiro lugar não é naturalmente trivial a escolha das partes críticas de um programa para as quais se consigam ganhos significativos de rapidez à custa de recorrer a operadores em *hardware* dedicado. Aquilo que pode sugerir, à primeira vista, ser uma escolha interessante pela natureza das operações envolvidas, pode traduzir-se numa degradação importante de desempenho, como aconteceu no exemplo estudado. Para além do tempo gasto com a

execução das instruções necessárias para a transferência de dados e resultados, é também importante quantificar o impacto que o tempo de execução do segmento de código elegido tem, no contexto das funções em que é usado. No caso estudado, a função `Plot()` é a função mais simples que contém o conjunto de operações seleccionado, mas onde essa tarefa representa apenas 6.4% do seu tempo total de execução. Por essa razão, em qualquer outra função deste sistema gráfico que envolva aquele conjunto de cálculos será naturalmente inferior o impacto do tempo de execução daquela sequência de operações. Além disso, a decisão de construir operadores dedicados para acelerar partes de um programa deve ser tomada sem pôr de lado possíveis optimizações realizadas sobre implementações em *software*. Neste exemplo, uma transformação simples no código da função conduziu a uma redução no tempo de execução que mostrou ser superior ao tempo gasto pela sequência de instruções que foi seleccionada para construir um operador dedicado.

4.5 Identificação automática de operadores dedicados

A identificação e construção de operadores dedicados para funções específicas de um problema são tarefas que requerem o conhecimento detalhado desse problema e do tipo de plataforma de *hardware* onde são construídos. Os exemplos de operadores que se apresentaram na secção anterior ilustram funções bem definidas que sugerem implementações eficientes como circuitos lógicos dedicados. A realização de operadores dedicados a funções nessas condições é geralmente mais bem conseguida por projectistas de *hardware* com experiência na tecnologia destino, do que ferramentas para síntese automática de *hardware* partindo de descrições de alto nível das operações a realizar. Ao contrário do desenho manual onde as melhores soluções são muitas vezes obtidas de forma *ad-hoc*, ferramentas de síntese automática consideram normalmente arquitecturas padrão para os circuitos que produzem e que na generalidade das situações não constituem a melhor realização em termos do compromisso conseguido entre espaço ocupado e desempenho atingido. No entanto, o recurso a sistemas de síntese automática é uma opção que, por um lado, permite desligar o utilizador dos detalhes associados à construção de *hardware* e por outro lado possibilita a obtenção de implementações em tempos reduzidos.

Uma abordagem já explorada por outros autores [AS93, RS94] no sentido de automatizar o processo de utilização de plataformas reconfiguráveis para acelerar programas, consiste em integrar no processo de compilação de um programa a identificação automática de sequências de instruções, para as quais sejam conseguidos operadores dedicados que

consigam reduzir o tempo global de execução do programa. A automatização deste processo e a integração no percurso normal de compilação de uma linguagem de programação de alto nível permite que um sistema reconfigurável possa ser usado de forma transparente por projectistas de *software*. No entanto, um programador geralmente escreve programas numa linguagem de alto nível tendo em vista a sua compilação e execução em processadores convencionais, e da forma como o programa é estruturado e codificado muito depende a facilidade ou dificuldade em identificar automaticamente blocos de instruções que sejam potenciais candidatos a implementações como operadores realizados em *hardware* dedicado. O problema é ainda mais complicado se for considerado um programa já compilado para um processador convencional realizando a análise ao nível do código *assembly*. Nesse caso, as eventuais optimizações levadas a cabo pelo compilador podem fazer desaparecer qualquer estruturação idealizada pelo programador na construção do código de alto nível.

Um sistema para particionamento automático de um programa escrito numa linguagem de programação de alto nível (C ou C++) é apresentado em [JEÖH94]. O sistema identifica sequências de instruções que possam ser realizadas num processador auxiliar dedicado construído sobre uma plataforma reconfigurável, sendo excluídos aqueles blocos que contenham operações em vírgula flutuante e chamadas a funções. As partes candidatas são assim constituídas essencialmente por blocos básicos de instruções, ciclos e funções terminais (que não invoquem outras funções). Para cada bloco de instruções candidato é estimado o ganho de desempenho que pode ser atingido realizando-os em *hardware*, contabilizando também o tempo necessário para mover dados entre a memória ou registos do processador principal e o processador dedicado. Os resultados obtidos não mostram ganhos de rapidez significativos, sendo principalmente a consequência do acesso lento do processador auxiliar à memória principal do processador principal, quando comparado com a rapidez que este consegue à custa do uso de memórias *cache*. De 8 aplicações estudadas, foram seleccionados 3500 blocos candidatos para realizar no processador auxiliar, dos quais 97% são responsáveis por menos de 1% do tempo total de execução da aplicação. Considerando o acesso à memória principal pelo processador auxiliar 5 vezes mais lento do que pelo processador principal, apenas em 3 aplicações que não envolvem acessos à memória principal foram conseguidos ganhos de desempenho que variaram entre 1.18 e 16 vezes. Uma conclusão deste trabalho é que para ser eficaz a selecção automática de blocos de instruções a realizar numa plataforma de *hardware* dedicado, é necessário que essa plataforma tenha acesso à memória principal do computador principal com tempos de acesso semelhantes aos conseguidos pelo processador principal. Isso pode ser conseguido integrando no mesmo circuito integrado do processador principal uma área reconfigurável para implementar operadores dedicados, proposta que foi já sugerida por vários autores,

como foi revisto no capítulo 2.

Uma área onde é também abordado o problema da divisão de funcionalidade entre *hardware* e *software* é designada por projecto combinado *hardware/software*(*hardware/software-codesign*). Em vários trabalhos são propostas metodologias para definir de forma automática a parte de um sistema a realizar num circuito de aplicação específica e a que pode ser realizada por programas executados em processadores convencionais [EHB93, GJM94, IJ95]. Os trabalhos desenvolvidos nesta área são focados na implementação de sistemas digitais complexos como soluções conjuntas de *hardware* e *software* e não têm só por objectivo acelerar programas explorando a utilização de processadores dedicados realizados sobre plataformas reconfiguráveis já construídas. Uma diferença importante diz respeito à forma como é especificado o problema a resolver. No contexto de *hardware/software-codesign*, o sistema digital pretendido só em casos particulares pode ser especificado de modo correcto numa linguagem de programação de alto nível convencional, dada a natureza sequencial das operações representadas. Normalmente são usadas linguagens de descrição de *hardware* que permitem modelar correctamente o paralelismo que é característico de sistemas digitais. Na identificação automática de secções de programas para realizar em processadores auxiliares dedicados, o objectivo é partir de especificações em linguagens de programação convencionais e escolher automaticamente as partes a realizar num processador auxiliar dedicado. Neste caso, é naturalmente importante ter em consideração condicionamentos que possam ser impostos pela arquitectura da plataforma de realização física usada na implementação de processadores dedicados.

No contexto da automatização do processo de configuração de plataformas reconfiguráveis para acelerar partes determinadas de uma aplicação, é hoje bem aceite que a selecção das secções a realizar como *hardware* dedicado em plataformas reconfiguráveis deve ser feita pelo projectista. Esta tem demonstrado ser a forma eficaz de conseguir ganhos de desempenho importantes, apesar de requerer da parte do utilizador as competências necessárias, que incluem um bom envolvimento com a aplicação e o conhecimento das características da plataforma de *hardware* disponível. A tendência já proposta por vários autores de incluir em processadores convencionais áreas reconfiguráveis com ligação directa aos recursos internos do processador obrigará necessáriamente a incluir em compiladores para processadores desse tipo, procedimentos que permitam explorar essas plataformas reconfiguráveis de forma transparente a um programador. Isso será facilitado pela proximidade física entre os elementos do processador convencional e do processador dedicado, o que permitirá ultrapassar um dos estrangulamentos importantes que resulta do tempo gasto com a comunicação de dados entre processador principal e processador auxiliar.

4.6 Conclusões

Neste capítulo apresentou-se uma proposta de arquitectura para um processador reconfigurável, construído em torno de unidades funcionais implementadas por circuitos lógicos programáveis FPGA. A arquitectura proposta foi inspirada na do processador THD apresentado no capítulo 3 e destina-se a funcionar como um processador auxiliar de um computador pessoal, para acelerar tarefas seleccionadas de uma aplicação executada pelo seu processador principal. Apresentaram-se exemplos de funções para as quais se desenvolveram operadores dedicados em *hardware*, e concluíu-se com um estudo detalhado de uma aplicação, extraída de uma biblioteca de rotinas gráficas desenvolvida pelo autor.

O recurso a um processador auxiliar dedicado para executar partes seleccionadas de um programa justifica-se pelo aumento global de desempenho que pode ser conseguido com a sua utilização. Para a arquitectura aqui proposta, são identificados dois níveis de intervenção no processo de selecção dessas partes. Ao nível da aplicação executada pelo computador principal, devem ser seleccionadas tarefas com granularidade elevada (funções ou subrotinas) que são traduzidas com um compilador adequado em sequências de operações suportadas por operadores realizados nas unidades funcionais do processador reconfigurável. Nessas tarefas são identificadas operações de granularidade inferior, para as quais possam ser construídos circuitos lógicos que implementem operadores dedicados a essas operações. Uma vez seleccionados esses dois níveis de tarefas a realizar no processador auxiliar, o processo de tradução de uma especificação de alto nível de um problema para código optimizado para a arquitectura proposta envolve problemas semelhantes aos encontrados em síntese de alto nível de circuitos digitais, que são abordados no capítulo seguinte.

A selecção e avaliação das tarefas a executar pelo processador reconfigurável deve considerar os tempos necessários à transferência de dados entre os dois processadores. Por este motivo, o uso de interfaces de expansão com baixo desempenho, como é o caso do barramento ISA, acrescenta tempos para comunicação de dados que podem comprometer os ganhos obtidos no tempo de execução pela utilização de arquitecturas adaptadas a problemas específicos. No entanto, isso é dependente das características da aplicação, bem como da quantidade de informação que é necessário transferir entre o processador principal e o processador auxiliar. Como exemplo, refira-se a aplicação que se apresenta no capítulo 6, onde o tempo gasto para passar dados e resultados usando operações de entrada e saída pelo barramento ISA representa uma pequena parcela do tempo gasto pelo processamento efectuado pelo processador auxiliar. Como os operadores dedicados a implementar nas unidades funcionais operam sobre dados residentes em elementos de

memória locais ao processador auxiliar, o seu desempenho individual não depende do modo como é transferida informação entre os dois sistemas.

Naturalmente que uma arquitectura reconfigurável com as características que foram apresentadas não se destina apenas à utilização como processador auxiliar de PCs, com o tipo de interface que foi considerado. Tempos de comunicação muito inferiores podem ser conseguidos recorrendo a outros tipos de interfaces normalizadas com o computador principal, ou mesmo com ligação directa aos barramentos do processador principal. Embora isto não possa geralmente ser feito em computadores comerciais, pode ser realizado em sistemas construídos de raíz em torno de processadores de usos gerais, e permite optimizar a taxa de transferência de dados entre os dois sistemas. Como foi já proposto por outros autores, a associação ideal entre um processador convencional e um sistema lógico reconfigurável consiste na integração das duas plataformas no mesmo circuito integrado, de modo a possibilitar o acesso aos recursos internos de um processador de usos gerais por operadores construídos como circuitos lógicos desenhados expressamente para aplicações específicas.

Uma consideração importante a ter em conta na decisão de recorrer ou não a um processador auxiliar como meio de acelerar uma aplicação prende-se com o grau de optimização que é possível conseguir para implementações em *software* dessa aplicação. Como se mostra com a aplicação apresentada na secção 4.4.3, podem ser conseguidos ganhos significativos de rapidez optimizando partes críticas de uma aplicação ao nível do código *assembly*. Claro que optimizações realizadas a esse nível são praticáveis para funções com complexidade reduzida e só servem para processadores que suportem o mesmo conjunto de instruções.

Capítulo 5

Configuração optimizada para o PUFR3

5.1 Introdução

No capítulo anterior apresentou-se uma arquitectura para um processador programável construído em torno de unidades funcionais reconfiguráveis, PUFR3. Abordou-se o problema da selecção de tarefas para serem realizadas em operadores dedicados de forma mais eficiente do que num processador convencional e foram apresentados exemplos seleccionados de operadores construídos para aplicações específicas. Neste capítulo é tratado o problema geral de optimização da configuração daquela arquitectura para uma aplicação específica, assumindo identificadas as operações a realizar em operadores especializados já construídos. Esses operadores específicos, bem como os necessários para as operações aritméticas e lógicas convencionais, são mantidos numa biblioteca de configurações para as unidades funcionais, que incluem implementações com diferentes características de execução, satisfazendo variados compromissos entre tempo de execução e espaço ocupado nos FPGAs que constituem as unidades funcionais. Numa primeira abordagem é considerado que as unidades funcionais são configuradas inicialmente, mantendo um conjunto de operadores necessários à aplicação durante a sua execução. Em cada unidade funcional coexistem operadores para diferentes funções que podem ser seleccionados sem requerer para isso qualquer gasto de tempo. Numa segunda abordagem, considera-se que as unidades funcionais são circuitos FPGA com capacidade de reconfiguração parcial. Neste contexto, os diferentes operadores seleccionados para uma unidade funcional são implementados à custa da reconfiguração dinâmica dos FPGA, o que obriga a considerar tempos de inactividade sempre que ocorre uma troca do operador utilizado na unidade

funcional.

O trabalho desenvolvido e que se apresenta neste capítulo foi focado na optimização simultânea dos problemas de selecção de configurações para as unidades funcionais, sequenciamento das operações e atribuição das operações do problema aos operadores implementados nas unidades funcionais, de forma a minimizar o tempo de execução e satisfazer as restrições de funcionamento impostas pelas características das unidades funcionais. É assumido que o problema a optimizar é uma sequência de operações representadas como segmento de instruções do tipo *assembly*, que pode ser obtido de especificações em linguagens de programação de alto nível recorrendo ao uso de compiladores configuráveis, tal como foi feito no trabalho apresentado no capítulo 3. O objectivo a atingir como resultado do processo de optimização conjunta dos 3 problemas referidos é formado por soluções para esses problemas: um conjunto de configurações para as unidades funcionais que incluam os operadores necessários para realizar as operações do problema, um sequenciamento no tempo das operações e uma atribuição das operações aos operadores realizados nas unidades funcionais. De uma solução conjunta assim obtida pode ser composta uma configuração para a arquitectura reconfigurável considerada. Esta é constituída por configurações para as unidades funcionais e pela sequência de micro-instruções que correspondem a cada operação, de forma a serem realizadas nos operadores a que foram atribuídas e nos ciclos de controlo determinados pelo sequenciamento.

Estes problemas constituem as tarefas principais do problema geral de optimização em síntese de alto nível [GWDL92b]. São conhecidas várias técnicas de optimização para este problema, mas as características que aqui se consideram para as unidades funcionais, em especial a sua capacidade de reconfiguração dinâmica, requerem uma nova abordagem para optimização conjunta daqueles 3 problemas, que é desenvolvida e apresentada neste capítulo.

Um método de optimização global aplicável à resolução daqueles problemas baseia-se no recurso a modelos de Programação Linear Inteira (PLI) [HL90]. O grande atractivo desta abordagem, é conseguir atingir uma solução do problema que é óptima segundo o critério estabelecido por uma função objectivo. No entanto, o tempo necessário à sua resolução cresce exponencialmente com a complexidade dos modelos que é função do número de variáveis de decisão e do número de restrições que o problema contém. Além disso, pelo facto de os modelos matemáticos serem construídos com base em equações lineares é difícil e por vezes impossível modelar certos tipos de restrições ou de funções objectivo. No contexto da aplicação desta técnica ao problema de optimização estudado. são revistos trabalhos propostos por outros autores que apresentam modelos de PLI para este problema, embora não satisfaçam as características genéricas consideradas para a

arquitectura alvo. Pela dificuldade encontrada em adaptar ou construir novos modelos que permitissem contemplar essas características, não foi explorada esta técnica de optimização.

Outras soluções propostas por diferentes autores para a optimização conjunta daqueles problemas baseiam-se na técnica de optimização combinatória conhecida por *Simulated Annealing* [LA89][1]. A generalidade deste método de optimização torna-o aplicável a praticamente qualquer problema de optimização discreta e permite incluir restrições complexas que são difíceis ou mesmo impossíveis de construir com modelos lineares de programação matemática. Tendo por base trabalhos propostos na literatura, foi desenvolvida uma solução para a optimização conjunta dos 3 problemas considerados, baseada numa aplicação do algoritmo *Simulated Annealing*. A aplicação desenvolvida permitiu considerar modelos de unidades funcionais com características de funcionamento muito gerais, incluindo a reconfiguração dinâmica dos operadores implementados em cada uma. Tanto quanto é conhecido, não existe nenhuma abordagem que contemple essas características de funcionamento tal como são consideradas pela solução que aqui se propõe.

A aplicação desenvolvida trabalha de forma interactiva guiada pelo utilizador e oferece variadas possibilidades de configuração, o que permite a sua utilização na optimização de problemas para diferentes arquitecturas alvo ou mesmo a problemas similares identificados noutros domínios de aplicação. Sem pretensões de oferecer um processo de optimização automático, constitui antes uma ferramenta que permite, de forma interactiva, explorar diferentes compromissos para a solução de um problema e também avaliar o desempenho de variadas estratégias para a condução da técnica de optimização implementada.

5.2 Partição do problema

Partindo de uma representação de um problema dada como uma especificação em código de alto nível, o processo de construção de uma configuração para a arquitectura considerada é dividido nas tarefas que se enumeram a seguir.

1. Selecção das tarefas do problema a realizar como operadores dedicados. Pelas razões que foram já referidas antes, considera-se que essa selecção é feita manualmente, requerendo um bom conhecimento do problema e das suas partes críticas. Essa escolha é realizada sobre a especificação de alto nível do problema, substituindo as secções de código correspondentes a essas tarefas por chamadas a funções, que representam a invocação de operadores dedicados.

[1] Este termo pode ser traduzido para "arrefecimento simulado", "solidificação simulada" ou "recozimento simulado", embora se tenha optado por manter ao longo do texto a designação original em Inglês.

2. Construção dos operadores dedicados. Este processo pode ser feito de forma automática, traduzindo as tarefas seleccionadas para linguagens de descrição de *hardware* e recorrendo a ferramentas de síntese automática de *hardware*. No entanto, o desenho manual pode conduzir à exploração eficaz de características próprias dos circuitos FPGA que constituem as unidades funcionais. Por exemplo, com pequeno acréscimo de custo (traduzido em aumento da complexidade do *hardware* ou do tempo de execução) pode ser possível partilhar blocos funcionais usados por diferentes operadores, de forma a alargar a diversidade de operações disponibilizadas por uma configuração de uma unidade funcional.

3. Tradução da especificação de alto nível numa linguagem de baixo nível do tipo *assembly*, incluindo instruções que invocam os operadores dedicados seleccionados. Como foi mostrado no capítulo 3, o processo de construir um compilador de uma linguagem de alto nível para um conjunto específico de instruções *assembly* pode ser simplificado recorrendo ao uso de compiladores configuráveis, como é o caso do gcc. Nesta aplicação assume-se que a especificação de alto nível do problema a executar pelo processador reconfigurável satisfaz restrições semelhantes às que foram consideradas para o compilador construído para o processador THD. Deste modo, o código *assembly* produzido é igualmente formado por um único bloco básico de instruções.

4. Uma vez seleccionados e construídos os operadores requeridos pela aplicação, obter uma configuração para a arquitectura considerada requer a optimização conjunta de três problemas fundamentais, que são as tarefas nucleares de optimização em síntese de alto nível de circuitos digitais: selecção do conjunto de operadores a realizar nas unidades funcionais, sequenciamento das operações, e atribuição das operações aos operadores implementados nas unidades funcionais.

5. Conhecida a ordem pela qual são executadas as operações do problema, o conjunto de operadores a implementar em cada unidade funcional e a atribuição das operações a esses operadores, o microprograma para controlo do processador pode ser obtido reunindo as sequências de micro-instruções que conduzem à execução de cada operação, nos ciclos de relógio determinados pelo sequenciamento.

Entre estas tarefas, o processo de optimização referido no passo 4 é o que apresenta maior importância no contexto do problema geral de optimização de uma configuração para a arquitectura considerada e para o qual se propõe e apresenta neste capítulo uma

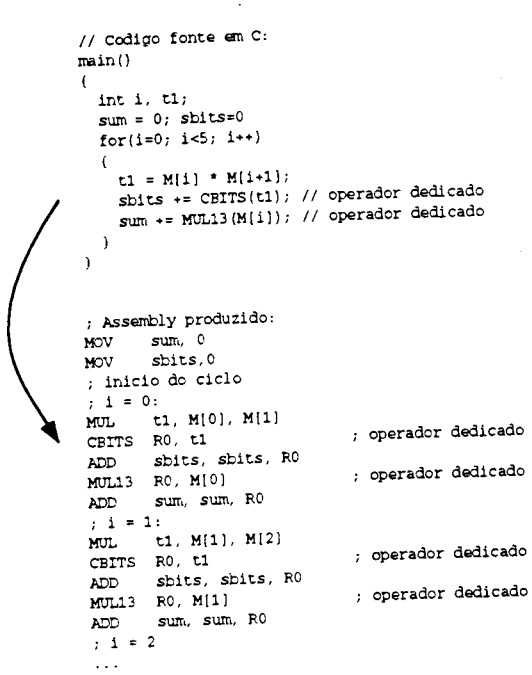
abordagem de resolução conjunta baseada numa aplicação do algoritmo *Simulated Annealing*.

5.3 Tradução da linguagem de alto nível

Para a aplicação desenvolvida, considera-se que o problema a optimizar é especificado por uma sequência de operações, representadas como instruções numa linguagem do tipo *assembly*. Esta pode ser obtida de uma especificação de alto nível do problema seleccionado para executar no processador reconfigurável, com um compilador adaptado para considerar um conjunto determinado de instruções de baixo nível. Fazendo uso de compiladores configuráveis permite-se explorar as fases de interpretação e optimização da especificação de alto nível do problema, o que constitui uma parte importante do processo de compilação de uma linguagem de alto nível. A aplicação desenvolvida e apresentada no capítulo 3 mostrou o processo de adaptação de um compilador de C para produzir um conjunto de instruções específico de um processador dedicado.

Na tradução de uma especificação de alto nível de um problema para o processador reconfigurável, pode não ser desejável explorar certos tipos de optimizações efectuadas por um compilador de uma linguagem de alto nível, por poder conduzir à transformação de operações ou sequências de operações passíveis de serem posteriormente realizadas de forma eficiente em operadores dedicados construídos nas unidades funcionais. Um exemplo de optimização não desejada são multiplicações por constantes inteiras para as quais podem ser construídos operadores dedicados eficientes, mas que um compilador pode simplificar para sequências de adições, subtracções ou deslocamentos de *bits*, naturalmente dependendo do tipo de operações que se considera suportadas pela arquitectura alvo. Para evitar situações desse tipo, as tarefas seleccionadas para realizar em operadores dedicados são substituídas no código fonte por chamadas a funções, de forma a ser mantida a sua integridade até à geração de código *assembly*.

Considera-se assim que o ponto de entrada para a aplicação que aqui se apresenta é um segmento de código de baixo nível representando um bloco básico de instruções de 2 ou 3 operandos, contendo operações aritméticas e lógicas convencionais, movimentação de dados e também as operações específicas correspondentes a tarefas implementadas em operadores dedicados. Na figura 5.1 mostra-se um exemplo de código C incluindo chamadas às funções `CBITS()` e `MUL13` que representam operadores dedicados, e a tradução para o código de baixo nível considerado.



```
// Codigo fonte em C:
main()
{
  int i, t1;
  sum = 0; sbits=0
  for(i=0; i<5; i++)
  {
    t1 = M[i] * M[i+1];
    sbits += CBITS(t1); // operador dedicado
    sum += MUL13(M[i]); // operador dedicado
  }
}

; Assembly produzido:
MOV     sum, 0
MOV     sbits,0
; inicio do ciclo
; i = 0:
MUL     t1, M[0], M[1]
CBITS   R0, t1                  ; operador dedicado
ADD     sbits, sbits, R0
MUL13   R0, M[0]                ; operador dedicado
ADD     sum, sum, R0
; i = 1:
MUL     t1, M[1], M[2]
CBITS   R0, t1                  ; operador dedicado
ADD     sbits, sbits, R0
MUL13   R0, M[1]                ; operador dedicado
ADD     sum, sum, R0
; i = 2
...
```

Figura 5.1: Um segmento de código fonte em C e o código de baixo nível correspondente.

5.4 Modelo das unidades funcionais

A adaptação da arquitectura reconfigurável considerada para uma aplicação específica requer a configuração dos FPGAs que constituem as unidades funcionais, de modo a criar o conjunto de operadores seleccionados para essa aplicação. No sentido de simplificar a linguagem empregue, será usado o termo unidade funcional para referir a unidade funcional física (um circuito FPGA ou eventualmente outra plataforma reconfigurável que o substitua), e também para designar uma configuração para a unidade funcional física, caracterizada pelo conjunto de operações que é implementado com essa configuração. No contexto em que este termo é empregue ao longo do texto, é claro o significado que lhe é atribuído embora em situações de ambiguidade seja explicitado o sentido que se lhe pretende atribuir.

Para cada configuração de uma unidade funcional é considerado um modelo que define o conjunto de operadores que ela implementa, as características de funcionamento de cada operador e um custo atribuído à unidade funcional. Como foi já referido antes, são considerados dois cenários no que diz respeito à forma como são configuradas as unidades funcionais físicas para implementar diferentes operadores: configuração inicial (estática) das unidades funcionais com um conjunto de operadores, que é mantida durante a execução do programa e reconfiguração dinâmica onde os vários operadores realizados por uma unidade funcional (física) são implementados à custa de reconfiguração parcial do

circuito FPGA que a constitui. No cenário em que se considera reconfiguração dinâmica, o modelo para uma unidade funcional é acrescido dos chamados custos de reconfiguração, que caracterizam os tempos associados à trocas dos operadores realizados por cada uma.

No contexto da arquitectura alvo considerada, é atribuído um custo a cada unidade funcional que pode traduzir a complexidade (em número de blocos configuráveis, por exemplo) do menor FPGA que permita implementar essa configuração. Num cenário diferente de realização física desta arquitectura, em que os operadores fossem implementados num circuito integrado construído expressamente para uma aplicação específica, então esse parâmetro deveria traduzir o custo efectivo da sua implementação física (por exemplo, área de silício, número de portas lógicas ou número de transistores).

5.4.1 Caracterização dos operadores

Uma unidade funcional agrega um conjunto de operadores que implementam operações de diferentes tipos e com diferentes modos de funcionamento. Cada operador produz um resultado e é caracterizado pelo número de operandos, número de ciclos de duração, latência (para operadores não *pipelined* a latência é igual à duração da operação) e os ciclos em que cada um dos seus operandos é consumido. O conceito de ciclo deve aqui ser entendido como ciclo de controlo ou fase de tempo em que é discretizada a execução das operações, e não necessáriamente o ciclo do relógio do sistema.

Sempre que é executada numa unidade funcional uma operação de um tipo diferente da última que nela foi realizada, o operador só pode ser usado depois de ser completada a operação anterior. Assim, o funcionamento de forma *pipelined* apenas pode ser explorado quando são executadas sequências de operações do mesmo tipo, tendo de ser concluída a operação anterior antes de ser iniciada uma operação de um tipo diferente, de forma a "esvaziar" a *pipeline*. Nos exemplos apresentados nas figuras deste capítulo é usada uma representação gráfica para ilustrar o modelo de diferentes operadores, onde se representa a duração da operação, os ciclos em que cada operando é consumido e o ciclo em que é produzido o resultado. Nessa representação não é mostrada a latência, que se considera igual a 1 ciclo para todos os operadores exemplificados. Considera-se que os operadores são circuitos síncronos com um relógio comum e que funcionam de forma *pipelined*. Na figura 5.2 exemplificam-se as representações usadas para 3 operadores com diferentes características de execução.

Uma característica importante a considerar na modelação do funcionamento *pipelined* de unidades funcionais que realizem diferentes operações diz respeito à forma como esse funcionamento é possível quando são efectuadas sequências de operações de diferentes

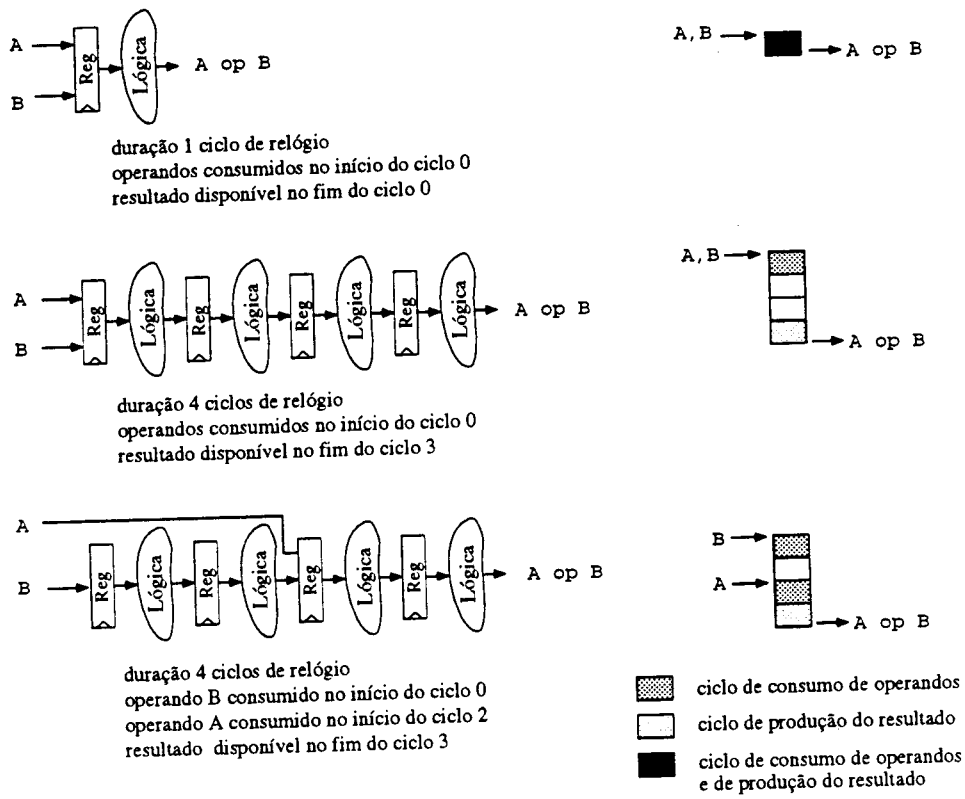


Figura 5.2: Representação gráfica dos modelos para 3 operadores com diferentes características.

tipos. Supondo que uma unidade funcional inclui operadores com números diferentes de andares de *pipeline*, ciclos de duração e de latência, é necessário definir de que forma essa unidade pode executar sequências de operações de tipos diferentes. A abordagem mais simples e que é assumida pelas técnicas conhecidas em síntese de alto nível, considera que, se uma unidade funcional realiza uma operação de forma *pipelined* com uma latência L, então na mesma unidade funcional pode ser iniciada a execução de uma nova operação de um tipo diferente após os L ciclos de latência do operador que está (ainda) a executar a operação anterior.

Esta situação nem sempre se verifica e depende da forma como são implementados os diferentes operadores que a unidade funcional suporta. Se existirem simultâneamente disponíveis numa unidade funcional como elementos individuais que não partilhem recursos entre si, então essa unidade pode executar em paralelo diferentes operações de forma *pipelined*, sem ocorrer qualquer interferência entre elas (figura 5.3-(a)). No entanto, uma limitação desta paralelização que não se relaciona com os operadores em si, diz respeito ao número de portas para entrada de operandos e saída de resultados, fisicamente dis-

poníveis na unidade funcional. Por exemplo, se apenas existem duas portas para receber os operandos e uma porta para saída do resultados sincronizadas por um relógio comum, então no mesmo ciclo não podem ser consumidos mais do que dois operandos nem pode ser produzido mais do que um resultado.

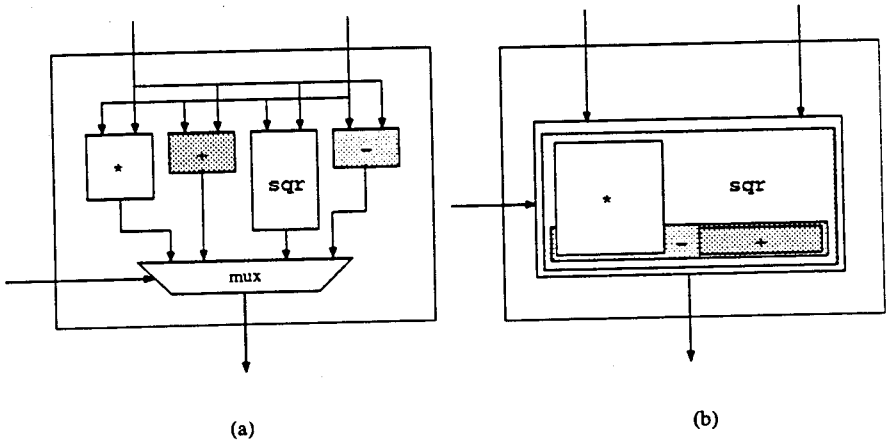


Figura 5.3: Unidades funcionais multi-operação: (a) coexistência de operadores; (b) partilha de recursos por diferentes operadores.

Se, por outro lado, uma unidade funcional implementar operadores de diferentes tipos partilhando recursos entre eles, então não é geralmente possível ter a trabalhar em simultâneo dois operadores que usem recursos comuns (figura 5.3-(b)), apesar de cada um dos operadores poder funcionar de forma *pipelined*. Nesse caso, é possível executar sequências de operações do mesmo tipo de forma *pipelined*, mas quando a mesma unidade funcional é invocada para realizar uma operação de outro tipo é necessário esperar pela conclusão da operação anterior. Naturalmente que isso depende da forma como são partilhados recursos de um circuito lógico para implementar operadores diferentes. Por exemplo, no processador THD os operadores multiplicador-divisor e somador-subtractor foram construídos de forma a poderem iniciar em cada ciclo de relógio operações de tipos diferentes.

A partilha de recursos numa unidade funcional entre operadores de tipos diferentes pode ser conseguida à custa da re-utilização de blocos funcionais ao nível da transferência entre registos, ou a um nível mais elementar, reprogramando os recursos configuráveis de um circuito FPGA. Com esta segunda abordagem, a selecção de diferentes operadores obriga a reconfigurar dinamicamente o FPGA, mas permite alargar o número de operadores diferentes que é possível implementar numa unidade funcional ao longo do tempo, para além daquele que é permitido pela capacidade do circuito físico que a implementa. Enquanto que partilhando blocos funcionais de grande granularidade, ao nível da trans-

ferência entre registos, a transformação de um operador noutro pode ser feita em tempos curtos, a reconfiguração parcial ou total de um circuito FPGA requer tempos mais longos durante os quais os elementos que são reconfigurados não podem ser utilizados.

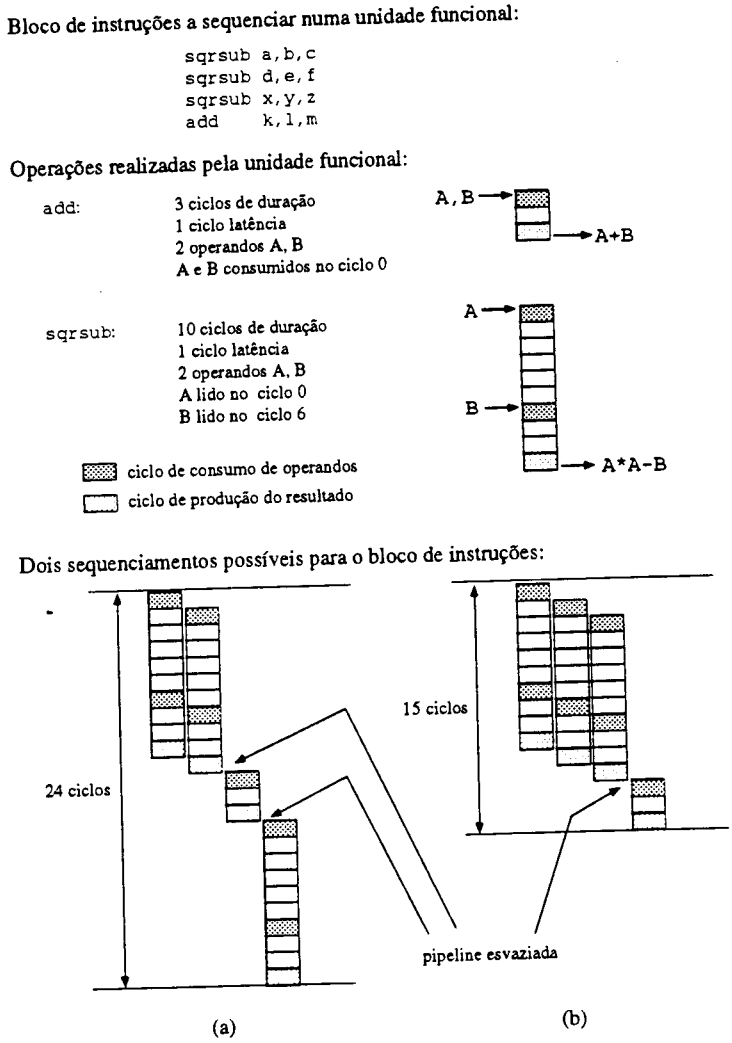


Figura 5.4: Execução *pipelined* de uma sequência de operações numa unidade funcional multi-operação.

Na figura 5.4 mostram-se dois sequenciamentos possíveis para um bloco de 4 operações de dois tipos diferentes (3 operações `sqrsub` e uma operação `add`) sem dependências de dados entre si, numa unidade funcional que implementa operadores *pipelined* para essas operações. Em (a), a realização da operação `add` entre duas operações `sqrsub` obriga a esvaziar a *pipeline* duas vezes, necessitando assim de mais 9 ciclos de relógio do que o sequenciamento conseguido em (b). Neste caso, a operação `add` é atrasada apenas um ciclo em relação a (a), o que é suficiente para realizar antes a terceira operação `sqrsub`.

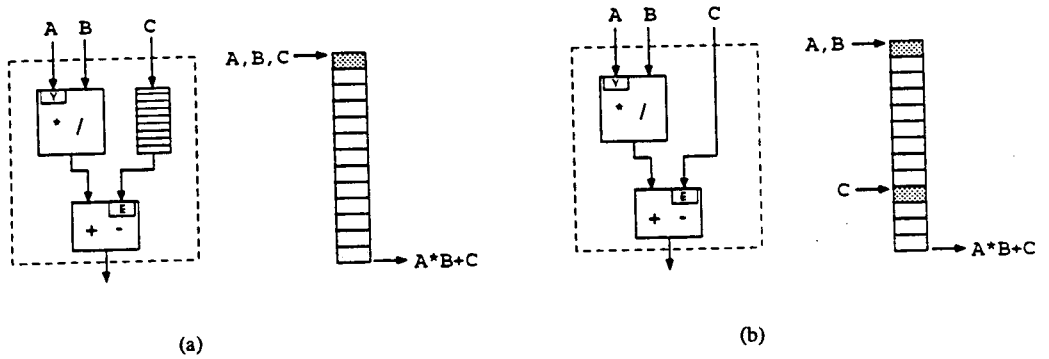


Figura 5.5: Modelo do operador multiplicador-somador do processador THD, com (a) e sem (b) o registo de deslocamento à entrada do somador.

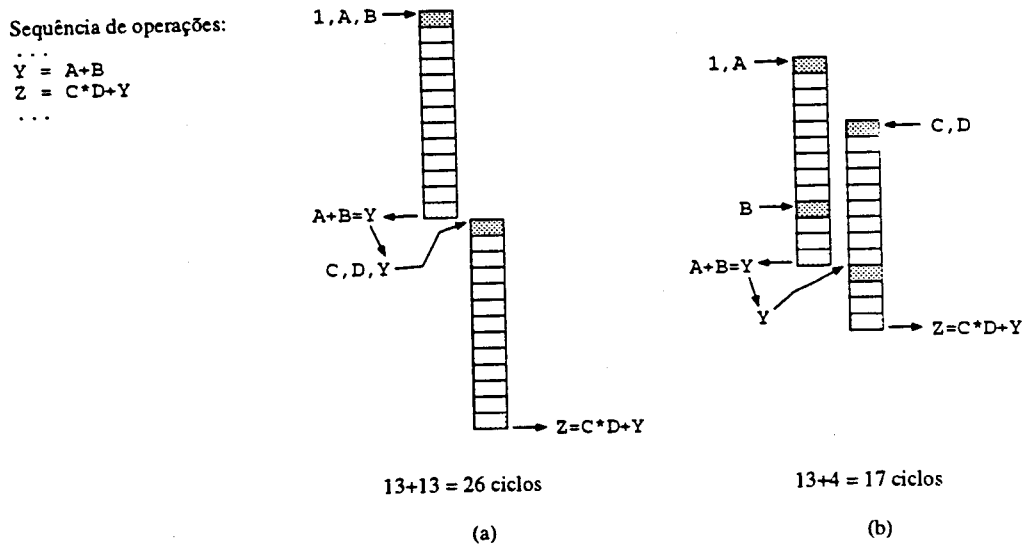


Figura 5.6: Comparação do sequenciamento de duas operações com dependência de dados nos dois operadores considerados na figura 5.5.

A especificação dos ciclos em que cada operando é consumido permite modelar de forma mais realista o comportamento de operadores complexos que não necessitem simultâneamente de todos os seus operandos. Para ilustrar isto recorde-se a unidade de cálculo do processador THD. Para atrasar o operando que entra no somador, de forma a ser possível fornecê-lo ao mesmo tempo dos operandos do multiplicador, foi usado um registo de deslocamento à entrada do somador, com um número de andares igual ao da *pipeline* do multiplicador. É possível eliminar esse registo de deslocamento, desde que seja fornecido o segundo operando do somador no mesmo ciclo em que é produzido o resultado na saída do multiplicador. Além disso, como o segundo operando do somador

só necessita estar disponível mais tarde, é possível reduzir o número de ciclos necessários à execução de certas sequências de operações. Na figura 5.5 apresentam-se os modelos para a unidade de cálculo original e para um operador sem aquele registo de deslocamento. Na figura 5.6 comparam-se os sequenciamentos nas duas unidades, de duas operações com dependência de dados entre si. Para além da redução da complexidade de *hardware*, considerar este modelo de operador permitiu, no exemplo apresentado, reduzir o número de ciclos necessários à execução das duas instruções consideradas.

5.5 Representação do problema

Como foi já referido antes, considera-se que o problema a traduzir para o processador reconfigurável é especificado por um bloco básico de instruções de baixo nível do tipo *assembly*. Essa representação é usada para construir um grafo dirigido, que representa a sequência de instruções que compõem o programa a executar (GDD - Grafo de Dependências de Dados [ASU86]). Os nós do grafo representam operações e os ramos identificam as relações de precedência existentes entre os resultados e operandos envolvidos nas operações. Associado aos nós e aos ramos é mantida informação complementar que é utilizada por várias funções envolvidas no processo de optimização.

O grafo tem como origem um nó de entrada que representa o início do problema e que é considerado a fonte dos operandos consumidos pelo bloco de instruções. O nó de saída representa o fim do problema e para ele convergem ramos de todos os nós que produzem os resultados finais do programa. Na figura 5.7 mostra-se um bloco básico contendo 9 operações e o GDD que o representa.

As relações de precedência determinam a ordem pela qual as operações devem ser realizadas para obter o resultado pretendido pelo programa e correspondem a dois tipos de dependências de dados entre operações. Dependência verdadeira entre duas operações A e B significa que o resultado produzido por A é usado como um dos operandos de B, o que significa que B só pode consumir esse operando após ser conhecido o resultado da operação A (figura 5.8-(a)).

Outro tipo de dependência acontece quando a operação A ocorre antes da operação B, e em que A tem como operando a mesma variável que é usada como resultado na operação B. Neste caso, embora a operação B não use o resultado produzido por A, existe uma relação de ordem entre as duas que obriga a que B seja realizada após A, de forma a não modificar o valor da variável que A usa como operando. Esta é designada por falsa dependência (ou escrita-após-leitura) e é exemplificada na figura 5.8-(b).

Uma terceiro caso acontece quando as duas situações referidas anteriormente ocorrem

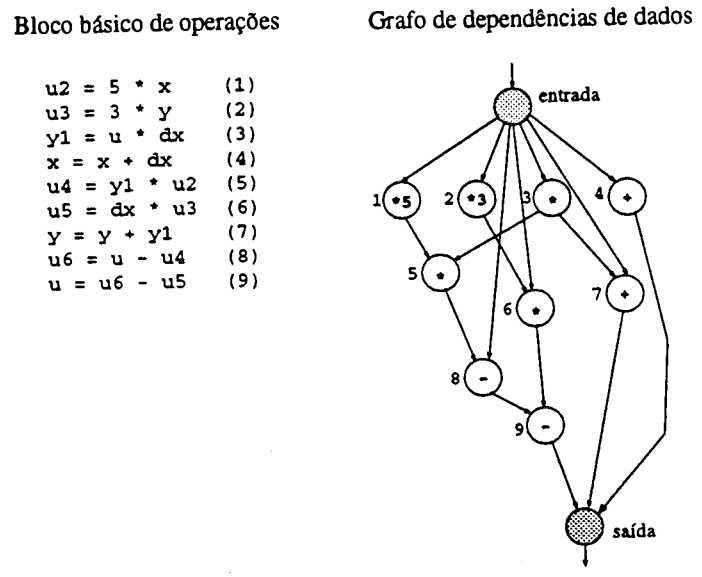


Figura 5.7: Exemplo de um bloco básico de operações e o grafo de dependências de dados (GDD) que o representa.

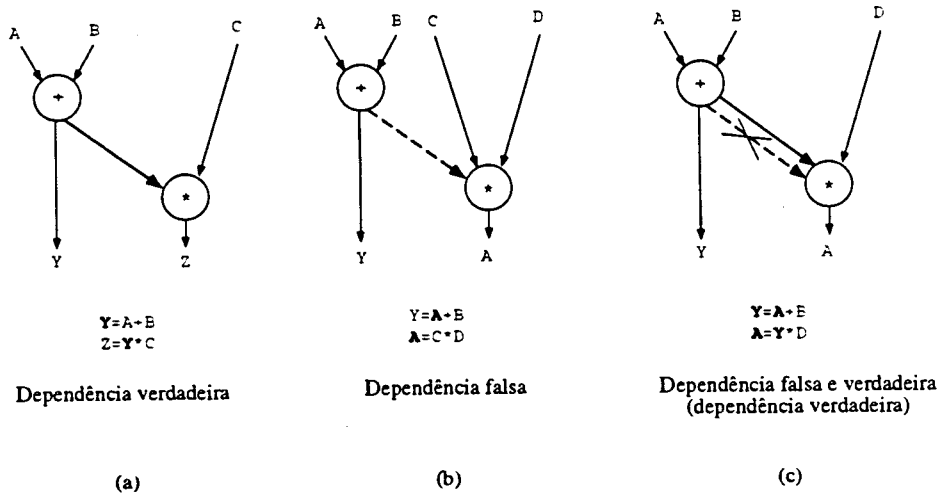


Figura 5.8: Dependência verdadeira, dependência falsa e dependência falsa e verdadeira, ilustradas com uma sequência de duas operações + e *.

ao mesmo tempo. Considerando que numa operação os operandos são sempre consumidos antes do resultado ser produzido, a restrição imposta pela dependência verdadeira é mais apertada, o que obriga a que a segunda operação tenha de ser realizada mais tarde do que se fosse apenas considerada a dependência falsa. Por esse motivo, neste caso apenas é construído um ramo no grafo, representando a dependência verdadeira (figura 5.8-(c)).

5.6 Abordagens com Programação Linear Inteira

Uma técnica para optimização já explorada por vários autores para resolver de forma conjunta o problema da selecção de unidades funcionais, sequenciamento de operações e atribuição de operações às unidades funcionais em síntese de alto nível, recorre a modelos de Programação Linear Inteira (PLI) para representar o problema, empregando técnicas de optimização bem conhecidas e implementadas em vários pacotes de *software*.

O primeiro trabalho conhecido que usa modelos de PLI para o problema de optimização do sequenciamento em síntese de alto nível foi proposto em [HP83]. Em [LHL89] é sugerida uma formulação de PLI para a optimização simultânea do sequenciamento de operações e selecção de unidades funcionais. Esta abordagem foi generalizada em [HLH91] e em [Ach93] para contemplar unidades funcionais com características de funcionamento mais realistas. O trabalho apresentado em [RMJL94] propõe uma formulação de PLI para resolver o problema de selecção de unidades funcionais, atribuição das operações às unidades funcionais e das variáveis a registos, que pressupõe conhecido o sequenciamento no tempo das operações.

Problemas de Programação Linear Inteira são computacionalmente exigentes pela necessidade de resolver vários problemas de Programação Linear (PL), gerados por partições sucessivas da região de soluções admissíveis do problema original. O número de ramificações necessário para atingir a solução óptima inteira é muito variável e dependente da estrutura das restrições do problema. É sabido que, para qualquer região de soluções admissíveis de um problema de PLI, existe uma outra região mais restrita que permite atingir a solução óptima inteira resolvendo apenas um problema de Programação Linear. Essa região é obtida à custa da inclusão de restrições adicionais que reduzem a região de soluções admissíveis de forma adequada. No entanto, só para um número restrito de problemas é conhecida a estrutura dessas restrições. Alguns trabalhos estudam a estrutura dessa região em formulações de PLI para o problema de optimização considerado em síntese de alto nível, visando a obtenção da solução óptima inteira à custa de resolução de um número reduzido de problemas de Programação Linear [GE93, CWM94].

Nas secções seguintes apresentam-se formulações de PLI que foram propostas para a optimização conjunta do problema geral de optimização, no contexto de síntese de alto nível. É apresentada uma formulação inicial e extensões que visam contemplar diferentes objectivos a optimizar, bem como modelos de unidades funcionais com características de funcionamento que se aproximam das consideradas neste trabalho. No entanto, não satisfazem a totalidade das assunções que foram descritas antes e que permitem modelar o funcionamento das unidades funcionais consideradas.

5.6.1 Formulação básica como PLI - minimização de recursos

Uma primeira formulação de PLI para a optimização conjunta do problema de sequenciamento de operações e selecção de unidades funcionais foi proposta em [LHL89]. Este modelo foi o ponto de partida para várias extensões propostas por outros autores que acrescentam outros objectivos e consideram características diferentes de funcionamento das unidades funcionais.

O modelo básico assume que as unidades funcionais podem realizar mais do que um tipo de operação, embora cada tipo de operação seja suportado por apenas uma unidade funcional. Todas as operações têm uma duração fixa e igual a um ciclo, independentemente da unidade funcional que a realiza.

O modelo de PLI parte do conhecimento do sequenciamento ASAP (*As Soon As Possible* e ALAP (*As Late As Possible*), para definir a mobilidade de cada operação, i.e., o intervalo de ciclos em que pode ser atribuída. A determinação dos sequenciamentos ASAP e ALAP pressupõe a existência de um número arbitrário de unidades funcionais, já que não são impostas restrições no número máximo de operações que podem ser realizadas no mesmo ciclo. Além disso, para calcular o sequenciamento ALAP é necessário especificar um limite superior para o número de ciclos, que no mínimo poderá ser igual ao obtido pelo sequenciamento ASAP.

O modelo de PLI minimiza uma função objectivo que representa um custo atribuído ao conjunto de unidades funcionais seleccionadas. A solução óptima determina o ciclo em que cada operação é executada, e o número de unidades funcionais de cada tipo necessárias para satisfazer esse sequenciamento.

Definindo:

o_i	operação i, com $i = 1 \ldots n$
FU_{t_k}	Unidade funcional do tipo t_k, com $k = 1 \ldots m$
S_i	ciclo ASAP da operação o_i
L_i	ciclo ALAP da operação o_i
c_{t_k}	custo de uma unidade funcional do tipo k
$o_i \longrightarrow o_j$	operação o_i antecede o_j

As variáveis de decisão inteiras são:

$$X_{i,j} = \begin{cases} 1 & \text{se operação } o_i \text{ é executada no ciclo } j, \\ 0 & \text{caso contrário} \end{cases}$$

M_{t_k} número (inteiro) de unidades funcionais do tipo k

A função objectivo minimiza o custo das unidades funcionais seleccionadas:

$$\sum_{k=1}^{m} c_{t_k} \cdot M_{t_k} \tag{5.1}$$

Sujeito às restrições seguintes:

$$\sum_{i=1, o_i \in FU_{t_k}}^{n} X_{i,j} \leq M_{t_k}, \text{ para todos } j, k \tag{5.2}$$

$$\sum_{j=S_i}^{L_i} X_{i,j} = 1, \text{ para todas as operações } o_i \tag{5.3}$$

$$\sum_{j=S_k}^{L_k} j \cdot X_{k,j} - \sum_{j=S_i}^{L_i} j \cdot X_{i,j} \geq 1, \text{ para todos } i, k \text{ tais que } o_i \longrightarrow o_k \tag{5.4}$$

O conjunto de restrições (5.2) impõe que em cada ciclo j, o número de unidades funcionais do tipo t_k seja maior ou igual que o número de operações suportadas por esse tipo de unidade funcional.

As restrições (5.3) impõem que cada operação o_i é atribuída a um e um só ciclo entre S_i e L_j, dentro da mobilidade que foi calculada para essa operação.

Finalmente, as relações de precedência entre as operações são descritas pelas inequações (5.4). Se o_k tem que ser realizada após o_i, então o ciclo em que a operação o_k é atribuída (correspondente ao primeiro somatório) tem que ser maior ou igual a 1 mais o ciclo da operação o_i.

5.6.2 Extensões à formulação original

Em [HLH91] e [Ach93] apresentam-se extensões à formulação anterior, para optimizar o problema segundo outros objectivos e considerar outros modelos de funcionamento para as unidades funcionais.

Minimização do número de ciclos do sequenciamento

No modelo original é fixado o número máximo de ciclos em que se pretende sequenciar as operações do problema, e minimiza-se o custo ou número das unidades funcionais seleccionadas. Fixando o número de unidades funcionais de cada tipo, ou estabelecendo um limite para o seu custo máximo, pode-se construir uma função objectivo que minimize o número de ciclos necessários ao sequenciamento das operações [HLH91]. Fazendo constante o número de unidades funcionais de cada tipo, M_{t_k}, define-se uma nova variável inteira C que representa a duração do sequenciamento, sendo superior ou igual a qualquer ciclo em que seja atribuída qualquer operação que não tenha sucessores:

$$\sum_{j=S_i}^{L_i} j \cdot X_{i,j} \leq C, \text{ para todas as operações } o_i \text{ sem sucessores} \tag{5.5}$$

Neste modelo, a função objectivo a minimizar passa a ser apenas a variável inteira C, que representa a duração do sequenciamento.

Unidades funcionais *pipelined*

No modelo proposto em [HLH91] são também incluídas restrições que contemplam a existência de unidades funcionais operando de forma *pipelined*. Para isso considera-se que uma unidade funcional *pipelined* do tipo t_k tem duração d_k e latência l_k, com $d_k > l_k$ (se $d_k = l_k$ então a unidade é não *pipelined*). Para cada variável $X_{i,j}$ associada a uma operação o_i do tipo t_k, define-se "grupo do ciclo j" como o conjunto de ciclos em que podem ser sequenciadas outras operações desse tipo, de forma a partilhar a mesma unidade funcional. Tomando como origem j, esses ciclos são $j, j-l_k, j-2\cdot l_k, \ldots, j-p\cdot l_k$, com $p \leq \lfloor d_k/l_k \rfloor$.

Para cada unidade funcional *pipelined* k e ciclo j, a restrição (5.2) é substituída por duas novas restrições:

$$\sum_{o_i \in FU_{t_k}} X_{i,j-p\cdot l_k} \leq Z_{k,j},\ 0 \leq p \leq \lfloor \tfrac{d_k}{l_k} \rfloor \tag{5.6}$$

e

$$\sum_{p=0}^{l_k-1} Z_{k,j-p} \leq M_{t_k} \tag{5.7}$$

A restrição (5.6) estabelece $Z_{k,j}$ como o número máximo de unidades funcionais do tipo k que são necessárias para satisfazer as operações sequenciadas no ciclo j. A restrição seguinte traduz que o número de unidades funcionais seleccionadas do tipo t_k, M_{t_k}, tem de satisfazer o número máximo de unidades necessárias para as operações sequenciadas nos ciclos $j, j-1, \ldots, j-l_k$, já que estas não podem ser executadas de forma *pipelined* na mesma unidade funcional.

Se nesse modelo for considerado que M_{t_k} são variáveis de decisão, pode-se estabelecer um limite para o número máximo de unidades funcionais seleccionadas, MAX_{FU}, incluindo a restrição:

$$\sum_{k=1}^{m} M_{t_k} \leq MAX_{FU} \tag{5.8}$$

ou incluir também um limite máximo $MAX_{C_{FU}}$ para o custo das unidades funcionais seleccionadas:

$$\sum_{k=1}^{m} c_{t_k} \cdot M_{t_k} \leq MAX_{C_{FU}} \tag{5.9}$$

Note-se que as duas restrições (5.8) e (5.9) podem coexistir no mesmo modelo, já que estabelecem dois tipos de limites diferentes: número e custo das unidades funcionais.

Para a construção deste modelo, é necessário obter os sequenciamento ASAP e ALAP modificados de forma a não exceder os limites impostos para o número ou custo das unidades funcionais. Para obter um primeiro sequenciamento que satisfaça as restrições impostas para o número máximo de recursos e permita conhecer um limite máximo para o número de ciclos necessários, é calculado inicialmente um sequenciamento com lista de prioridades (*List Scheduling* [GWDL92b]).

Verificação de admissibilidade de soluções

Uma terceira variante do modelo PLI apresentado em [HLH91] consiste em fixar um número de ciclos e unidades funcionais, sem especificar qualquer função custo a optimizar. Em vez de encontrar a solução óptima, este modelo permite avaliar a admissibilidade de diferentes compromissos estabelecidos para o custo ou número das unidades funcionais e o número de ciclos para o sequenciamento. Neste problema são apenas usadas variáveis inteiras binárias (que só podem tomar os valores 0 ou 1), o que resulta num problema de Programação Linear Inteira Binária (PLIB), permitindo recorrer a técnicas de optimização mais eficientes do que as requeridas para problemas de programação inteira [HL90].

Unidades funcionais multi-operação

O trabalho apresentado em [Ach93] alarga o âmbito de aplicação dos modelos propostos por [LHL89] e [HLH91], apresentando uma formulação de PLIB, considerando que cada unidade funcional pode realizar mais do que uma operação, e que o mesmo tipo de operação é disponibilizado por unidades funcionais diversas. Numa primeira abordagem, é considerado que um tipo de operação tem a mesma duração independentemente da unidade funcional em que é realizada. O mesmo autor propõe a generalização do modelo para satisfazer operações com tempos de execução dependentes da unidade funcional que a executa.

Esta formulação traduz de forma mais realista as características de unidades funcionais apresentando diferentes compromissos entre custo, diversidade de operadores e desempenho. É mais próximo daquele que se considera para as unidades funcionais do processador reconfigurável, embora não satisfaça inteiramente as características de execução pretendidas. Além disso, o modelo de PLIB proposto só resolve o problema de sequenciamento.

uma vez fixado um conjunto de unidades funcionais que ofereçam todos os operadores necessários às operações do problema.

No modelo apresentado na secção anterior, com excepção da variável C que é inteira e representa a duração do sequenciamento, todas as outras são variáveis inteiras binárias. Para obter um modelo só com variáveis binárias é acrescentada uma operação artificial o_p que representa o fim da sequência de operações a sequenciar. A variável C do modelo anterior, que representa a duração do sequenciamento, corresponde ao ciclo em que o_p é sequenciada, e pode desta forma ser expressa apenas com variáveis inteiras binárias. Introduzindo um conjunto de variáveis $X_{p,j}$ (iguais a 1 se a operação o_p for sequenciada no ciclo j e zero caso contrário), pode-se exprimir a função objectivo que minimiza a duração do sequenciamento como:

$$\sum_{j=S_p}^{L_p} j \cdot X_{p,j} \tag{5.10}$$

As restrições (5.2) do modelo original impõem que o número de unidades funcionais do tipo t_k, M_{t_k}, deve ser suficiente para executar todas as operações desse tipo que são atribuídas em cada ciclo. Para operações o_i que possam ser executadas em duas unidades funcionais de tipos diferentes, t_{k_1} e t_{k_2}, é demasiado forte incluir a variável $X_{i,j}$ naquelas restrições que determinam o número mínimo de unidades funcionais necessárias no ciclo j. Na verdade, isso significa que a mesma operação ocupa as duas (ou mais) unidades funcionais em que pode ser realizada.

A solução que foi proposta para contornar este problema consiste em formar um conjunto com todos os grupos de operações não disjuntas que a biblioteca de unidades funcionais suporta. Para cada grupo Π_k define-se $M(\Pi_k)$ como o número de unidades funcionais que podem realizar pelo menos uma das operações de Π_k. As restrições (5.2) do modelo original são modificadas para:

$$\sum_{i=1, o_i \in \Pi_k}^{n} X_{i,j} \leq M(\Pi_k), \text{ para todos } j, \Pi_k \tag{5.11}$$

Esta transformação pressupõe que o número $M(\Pi_k)$ é constante e função do número de unidades funcionais de cada tipo que forem consideradas. Como as variáveis $X_{i,j}$ são inteiras binárias, todas as restrições do tipo (5.11), que apresentarem no primeiro membro um número de variáveis $X_{i,j}$ igual ou inferior ao valor exibido no segundo membro, $M(\Pi_k)$, são sempre satisfeitas e podem ser eliminadas do conjunto de restrições. Este modelo permite considerar que o mesmo tipo de operação é suportado por unidades funcionais diferentes, mas o número de unidades funcionais de cada tipo, M_{t_k}, é considerado como constante.

operações com tempos de execução diferentes em diferentes unidades funcionais

Outro problema para o qual é proposta solução pelo mesmo autor [Ach93] resulta de considerar que há operações com tempos de execução diferentes, se realizadas por diferentes unidades funcionais. Para distinguir estas operações das que têm durações iguais em todas as unidades funcionais, atribui-se a designação O_i às primeiras. Isto obriga a acrescentar variáveis que determinem o tipo de unidade funcional a que é atribuída cada operação para ser conhecida a sua duração, o que é necessário para construir as restrições que modelam as relações de precedência entre as operações. Para isso definem-se novas variáveis de decisão inteiras binárias, $s_{i,p,j}$, que representam as operações O_i para as quais existam operadores com diferentes durações:

$$s_{i,p,j} = \begin{cases} 1 & \text{se operação } O_i \text{ é executada numa unidade do tipo } p \text{ e no ciclo } j, \\ 0 & \text{caso contrário} \end{cases}$$

Para as operações O_i, as variáveis $X_{i,j}$ são definidas como:

$$X_{i,j} = \sum_{p=1}^{N_{O_i}} s_{i,p,j} \tag{5.12}$$

onde N_{O_i} representa o número de diferentes tipos de unidades funcionais onde a operação O_i pode ser realizada. Para garantir que cada operação só é atribuída a um ciclo e a uma unidade funcional é acrescentada a restrição seguinte, para todas as operações O_i:

$$\sum_{j=S_p}^{L_p} \sum_{p=1}^{N_{O_i}} s_{i,p,j} = 1 \tag{5.13}$$

As novas restrições que representam as relações de precedência entre as operações O_i e o_k são expandidas num conjunto de N_{O_i} restrições, uma para cada tipo de unidade funcional que suporta a operação O_i. Assim, se a operação o_k usa o resultado produzido pela operação O_i, o ciclo em que pode ser sequenciada a operação o_k depende do tipo de unidade funcional p a que é atribuída a operação O_i, já que isso define a sua duração naquele tipo de unidade funcional, $d(t_p(O_i))$. As restrições (5.4) do modelo original, para todos os pares de operações O_i e o_k tais que $O_i \longrightarrow o_k$, são substituídas por:

$$\sum_{j=S_k}^{L_k} (j+D) \cdot X_{k,j} - \sum_{j=S_i}^{L_i} (j+D) \cdot s_{i,p,j} \geq d(t_p(O_i)), \text{ para } p = 1 \ldots N_{O_i} \tag{5.14}$$

onde D é a duração máxima da operação O_i, no conjunto de unidades funcionais que a suportam. O modelo inclui restrições construídas de forma idêntica para traduzir as situações em que $o_i \longrightarrow O_k$ e $O_i \longrightarrow O_k$

Este modelo de PLIB permite optimizar simultâneamente o sequenciamento de um conjunto de operações e a sua atribuição a unidades funcionais que suportam conjuntos de operações não disjuntos, com funcionamento *pipelined* e diferentes durações de realização de operações. No entanto, com esta formulação é apenas resolvido o problema de sequenciamento, já que é fixado o número e tipo de unidades funcionais seleccionadas.

5.6.3 Limitações dos modelos de PLI

Os modelos de PLI e PLIB referidos não contemplam as características de funcionamento das unidades funcionais, da forma que foram referidas na secção 5.4. Nestes modelos é considerada uma latência constante para a unidade funcional, e não é imposta qualquer restrição à execução de forma *pipelined* quando uma unidade funcional realiza sequências de operações de tipos diferentes. Como foi referido na secção 5.4, a forma como são implementadas unidades funcionais que suportem diferentes operadores partilhando recursos entre eles pode condicionar a execução de modo *pipelined* a sequências de operações do mesmo tipo. Outra limitação desses modelos resulta da não consideração de tempos de inactividade numa unidade funcional, dedicados à reconfiguração da operação que ela realiza. Como foi já referido, isso é necessário para modelar unidades funcionais reconfiguráveis dinamicamente. Outros trabalhos conhecidos [WMGB93, GE93, CWM94] propõem diferentes abordagens recorrendo a diferentes modelos de PLI e PLIB para a optimização conjunta deste problema, mas de forma semelhante às formulações que se apresentaram não permitem contemplar as características de funcionamento associadas ao modelo de unidade funcional considerada.

Para optimizar de forma conjunta o problema da selecção das configurações para as unidades funcionais, sequenciamento das operações e atribuição de operações aos operadores realizados nas unidades funcionais foi desenvolvida uma solução baseada na técnica de optimização combinatória conhecida por *Simulated Annealing*, que se apresenta na secção seguinte.

5.7 Uma aplicação baseada em *Simulated Annealing*

Uma técnica de optimização combinatória já aplicada com sucesso a uma gama muito variada de problemas de optimização discreta é o algoritmo *Simulated Annealing* [LA89]. No contexto de síntese de alto nível, esta técnica é usada por vários autores, sendo as primeiras aplicações conhecidas ao problema de optimização conjunta do selecção de operadores e sequenciamento de operações apresentadas em [DN89, SZ90]. Nestas abordagens, o pro-

blema é formulado como distribuir operações num espaço bidimensional, onde uma das dimensões representa tempo (número de ciclos) e a outra representa espaço (*hardware* ocupado pelas unidades funcionais necessárias para realizar as operações). No entanto, nenhuma das aplicações conhecidas assume unidades funcionais que satisfaçam o modelo de execução considerado, nomeadamente a existência de tempos de reconfiguração associados à troca de operadores, que é essencial para modelar o funcionamento de unidades funcionais reconfiguráveis dinamicamente.

Nesta secção apresenta-se uma solução para o problema geral de optimização de síntese de alto nível baseada no algoritmo *Simulated Annealing*, dentro das restrições consideradas. A aplicação desenvolvida teve por base a solução apresentada por [DN89], representando uma solução do problema como uma distribuição das suas operações num espaço bidimensional. Numa primeira parte apresenta-se a implementação do algoritmo considerando tempos nulos de reconfiguração, mas condicionando a execução *pipelined* a sequências de operações do mesmo tipo. Em seguida, são apresentadas as modificações que foram introduzidas e que permitiram generalizar a aplicação desenvolvida de forma a contemplar unidades funcionais reconfiguráveis dinamicamente, impondo um número de ciclos de inactividade requeridos para a sua reconfiguração, sempre que ocorre uma troca do tipo de operação realizada numa unidade funcional.

5.7.1 O algoritmo *Simulated Annealing*

O algoritmo *Simulated Annealing* para optimização discreta de problemas data do início dos anos 80 [KGV83], embora as ideias que lhe deram origem remontem a 1953, com o trabalho de Metropolis *et al.* [MRR+53]. Neste trabalho é proposto um algoritmo para simular em computador o processo termodinâmico de recozimento de materiais, aquecidos para além do seu ponto de fusão. O estado de energia interna do material solidificado depende da forma como é conduzido o arrefecimento. Com arrefecimento lento são formadas estruturas cristalinas de grande dimensão correspondendo a um elevado grau de organização das partículas do material, o que representa estados de baixa energia interna. Com um arrefecimento rápido o material atinge um estado de energia interna elevada, o que corresponde a uma solidificação com uma reduzida organização das partículas. Durante o processo termodinâmico de arrefecimento, a probabilidade de acontecer um aumento de energia δE é dada por:

$$p(\delta E) = e^{\frac{-\delta E}{kt}} \tag{5.15}$$

onde k é a constante de Boltzmann e t a temperatura do material. O algoritmo de Me-

tropolis é conduzido para um estado de baixa energia gerando perturbações no sistema e calculando a variação de energia daí resultante. Se a energia desce, então essa perturbação conduz a um novo estado do sistema. Se a energia aumenta, o novo estado é atingido com uma probabilidade dada pela equação (5.15).

Com o algoritmo *Simulated Annealing*, Kirkpatrik [KGV83] propôs a aplicação do algoritmo de simulação de Metropolis a problemas de optimização discreta. Uma solução do problema é associada a um estado de energia do material e a energia desse estado corresponde ao valor da função objectivo para essa solução do problema. Uma perturbação do estado termodinâmico do sistema corresponde a uma modificação da solução do problema e o estado de baixa energia atingido com a solidificação do material equivale à solução final obtida para o problema. A temperatura do material, que governa a probabilidade de aceitação de transições para estados de energia mais elevada, funciona no processo de optimização como um parâmetro de controlo para o qual é geralmente mantida a designação de "temperatura", sendo, pelo mesmo motivo, referida a forma como esse parâmetro decresce ao longo do processo iterativo por "arrefecimento".

O algoritmo *Simulated Annealing* é uma variante das técnicas de pesquisa local por descida monótona da função objectivo [Ree95]. Estes métodos partem de uma solução inicial do problema e procuram soluções obtidas por perturbações elementares que conduzam a melhorias locais da função objectivo do problema. No entanto, têm o inconveniente de poderem convergir para óptimos locais, embora partindo de diferentes soluções iniciais, ou alargando a gama de perturbações provocadas numa solução, possa ser evitada a convergência para óptimos locais. O algoritmo *Simulated Annealing* introduz uma heurística na técnica de pesquisa local, permitindo piorar a função objectivo com uma probabilidade que depende do valor do parâmetro de controlo (temperatura), decrescente ao longo do processo iterativo. Estudos teóricos [LA89] provam que, com um número de iterações suficientemente grande, o algoritmo converge para o óptimo global do problema. No entanto, o número de iterações necessário para garantir essa convergência é, pelo menos teóricamente, superior ao que é necessário para pesquisar de forma exaustiva todo o espaço de soluções do problema. Aplicações práticas conseguem atingir boas soluções que são sub-óptimas na generalidade dos casos, mas requerendo números de iterações praticáveis, naturalmente muito inferiores ao necessário para pesquisar exaustivamente a região de soluções admissíveis.

O algoritmo parte de uma solução inicial, e gera soluções adjacentes à custa de modificações elementares nessa solução. Soluções que conduzam a melhoria da função objectivo são imediatamente aceites, enquanto que aquelas que piorem a função objectivo são aceites com uma probabilidade $F(\Delta C, T)$, função da variação da função objectivo ΔC e da

temperatura T. Traduzindo da termodinâmica a probabilidade de aceitação de estados de maior energia, uma função que geralmente conduz a bons resultados é:

$$F(\Delta C, T) = e^{-\frac{\Delta C}{T}} \tag{5.16}$$

Uma alternativa àquela função que evita o cálculo da exponencial consiste em aproximá-la por:

$$F(\Delta C, T) = 1 - \frac{\Delta C}{T} \tag{5.17}$$

Assim, para uma solução que melhore a função objectivo, ΔC é negativo e essa solução é aceite. A probabilidade de aceitação de soluções quando a função objectivo é piorada (ΔC positivo), depende do valor de ΔC e do valor de T. Quando T é elevado, é alta a probabilidade de aceitar soluções com ΔC grande. À medida que T decresce, a probabilidade de aceitar soluções piores diminui também, segundo a função de probabilidade considerada.

O decrescimento de T (arrefecimento) pode ser feito seguindo várias estratégias. Uma forma de arrefecimento usual consiste num decrescimento exponencial de T, caracterizado por um parâmetro α que controla a rapidez como T decresce (equação 5.18). Valores típicos deste parâmetro que mostram conduzir a bons resultados em diversas aplicações deste algoritmo situam-se geralmente entre 0.8 e 0.99.

$$T_{n+1} = \alpha \cdot T_n \tag{5.18}$$

A forma como é feito o decrescimento de T pode seguir estratégias diferentes recorrendo a outras funções, embora os resultados obtidos em várias aplicações mostrem que é geralmente mais importante a rapidez de decrescimento da temperatura do que a forma como essa redução é efectuada.

O algoritmo começa com uma temperatura T elevada para permitir uma probabilidade elevada de aceitar soluções que piorem a função objectivo, de forma a evitar o congelamento antecipado da solução em óptimos locais. No entanto, a temperatura inicial não pode ser demasiado elevada para evitar que seja aceite qualquer solução gerada, o que não contribui para melhorar a função objectivo e conduz ao aumento do número de iterações necessário à convergência do algoritmo. Se o valor de T inicial for demasiado baixo que conduza a probabilidades reduzidas de aceitação de soluções piores, então o algoritmo pode degenerar numa pesquisa local pura, conduzindo o processo iterativo só por soluções que melhorem a função objectivo.

Para cada valor de T é gerado um número de soluções que depende naturalmente das características do problema, mas que deve permitir explorar convenientemente o espaço de soluções adjacentes nesse valor de temperatura. Como à medida que T decresce o número de soluções aceites vai sendo cada vez menor, o número de iterações pode ser aumentado com o decrescimento da temperatura. Uma estratégia consiste em estabelecer um limite máximo para o número de soluções que se aceitam em cada valor de T, ao fim do qual este é reduzido. Como para valores baixos de temperatura esse limite pode não ser atingido com um número de iterações razoável, deve ser também imposto um limite superior no número de iterações a efectuar em cada T. Outro processo consiste em estabelecer um número de iterações que cresça, de forma geométrica ou aritmética, com a diminuição de T. Uma técnica proposta em [LM86] consiste em realizar uma única iteração para cada valor de T, reduzindo a temperatura de forma muito lenta.

Em teoria, para o processo iterativo terminar é necessário atingir o valor zero para a temperatura. No entanto, em aplicações práticas não é naturalmente necessário que isso seja verificado, já que para valores de T suficientemente baixos, a probabilidade de aceitar soluções que piorem a função objectivo e com isso permitam sair de soluções óptimas locais é praticamente zero. Com o objectivo de acelerar o processo iterativo, podem ser estabelecidos critérios de paragem especificando um valor mínimo limite a atingir para T, um número máximo de iterações, ou uma percentagem mínima de soluções que devam ser aceites para cada valor de T.

O algoritmo *Simulated Annealing* é representado pelo pseudo-código apresentado na figura 5.9, onde é usado o decrescimento exponencial da temperatura representado pela equação (5.18), e uma função de probabilidade de aceitação de soluções piores igual à definida pela equação (5.16).

Uma aplicação prática do algoritmo *Simulated Annealing* a um problema concreto de optimização requer a construção de estruturas de dados eficientes para representar e trabalhar com as entidades que o algoritmo manipula: soluções, soluções adjacentes e funções para a avaliação de admissibilidade de soluções e avaliação da função objectivo. O desempenho, em termos de rapidez de execução, de uma aplicação construída em torno desta técnica de optimização depende em grande medida da eficiência das funções que geram e avaliam soluções, o que também depende da forma como são construídas as estruturas de dados usadas para representar e manipular essas entidades. A dificuldade de implementação desta técnica a novos problemas reside principalmente na necessidade de afinação dos parâmetros que controlam o algoritmo, para os quais não há regras que sejam universalmente válidas para qualquer aplicação. Cada caso apresenta características próprias, e a afinação desta técnica é geralmente complicada, requerendo trabalho de expe-

```
T = Tinicial;
S = Solução_Inicial;
C = Custo( S );
ENQUANTO critério de paragem não satisfeito
    ENQUANTO critério de paragem para T não satisfeito
        Snova = Nova_Solução( S );
        Cnovo = Custo( Snova );
        ΔC = Cnovo - C;
        SE ΔC < 0 ENTÃO
            S = Snova;
            C = Cnovo;
        SENÃO
            SE exp(-ΔC /T) > RANDOM(0, 1) ENTÃO
                S = Snova;
                C = Cnovo;
    T = alpha * T;
```

Figura 5.9: Pseudo-código do algoritmo *Simulated Annealing*.

rimentação que permita avaliar diferentes estratégias, de forma a conseguir uma aplicação de optimização robusta que consiga boas soluções para a classe de problemas para que é construída.

5.7.2 Representação de soluções

Uma solução para o problema de optimização considerado engloba as soluções para os 3 subproblemas que o compõem. O problema de sequenciamento tem por solução uma atribuição das operações a ciclos de controlo e é admissível se forem respeitadas as relações de precedência entre as operações, especificadas no GDD. A solução para o problema de selecção de unidades funcionais consiste num conjunto de configurações para as unidades funcionais, sendo admissível se oferecer todos os tipos de operadores necessários às operações do problema e se não for excedido o número máximo estabelecido para as unidades funcionais. Finalmente, o problema de atribuição das operações às unidades funcionais tem por solução uma associação entre cada operação do problema e os operadores disponíveis no conjunto de unidades funcionais seleccionadas.

Dado o modelo considerado para as unidades funcionais (suportando tipos diferentes de operações, operadores *pipelined* e ciclos diferentes para consumo dos operandos), os três problemas são intimamente interligados e não é possível obter de forma separada soluções para cada um deles: para definir um sequenciamento que satisfaça as relações de precedência entre operações, é necessário conhecer o número de ciclos que cada ope-

ração demora, quando necessita de ter disponíveis os seus operandos e em que condições pode ser executada de forma *pipelined*; para isso é necessário conhecer qual o operador que realiza cada operação, o que, por sua vez, só pode ser determinado se for conhecido o conjunto de unidades funcionais seleccionadas. Esta interdependência é agravada de forma significativa quando se consideram unidades funcionais em que a troca de operadores requer tempos de inactividade, associados à reconfiguração dos FPGAs que constituem as unidades funcionais.

De forma semelhante à aplicação proposta em [DN89], uma solução é aqui representada por uma distribuição das operações num plano bidimensional, onde o eixo vertical representa tempo discretizado em ciclos de controlo, e o eixo horizontal representa as unidades funcionais seleccionadas nessa solução (figura 5.10). Em cada solução pode naturalmente ser incluída mais do que uma unidade funcional do mesmo tipo, sendo a cada uma atribuídas operações diferentes do problema, tal como é mostrado no exemplo apresentado na figura 5.10.

Uma solução é admissível se forem verificadas todas as restrições consideradas, associadas a cada um dos 3 subproblemas que o compõem. Naturalmente que as soluções finais a atingir com o processo de optimização devem ser admissíveis, satisfazendo simultâneamente aquele conjunto de restrições. No entanto, no decorrer do processo de optimização pode ser desejável a iteração para soluções que violem restrições do problema, embora as operações tenham de ser sempre atribuídas a operadores que as suportem, o que é fundamental para conhecer as características de execução de cada uma. Considera-se por isso que é possível ter soluções que violem as relações de precedência entre as operações e que ultrapassem o limite máximo estabelecido para o número de unidades funcionais seleccionadas. Soluções finais que violem as restrições de precedência podem ser transformadas em soluções admissíveis, atrasando no tempo as operações responsáveis pelas violações dessas restrições de um número de ciclos apropriado, bem como todas aquelas que dela dependem. No entanto, não é possível, pelo menos de forma simples, tornar admissíveis soluções finais que ultrapassem o número máximo de unidades funcionais estabelecido.

Uma solução é caracterizada por variadas medidas de qualidade, que podem ser usadas para construir a função objectivo que orienta o processo de optimização. Tomando como alvo a arquitectura proposta, em que é fixo o número de unidades funcionais, o objectivo global a optimizar consiste em minimizar o número de ciclos necessários à execução do programa. No modelo de solução representado na figura 5.10, isto corresponde a minimizar a altura do espaço ocupado pelas operações, sem exceder segundo a largura, o número máximo de unidades funcionais permitido.

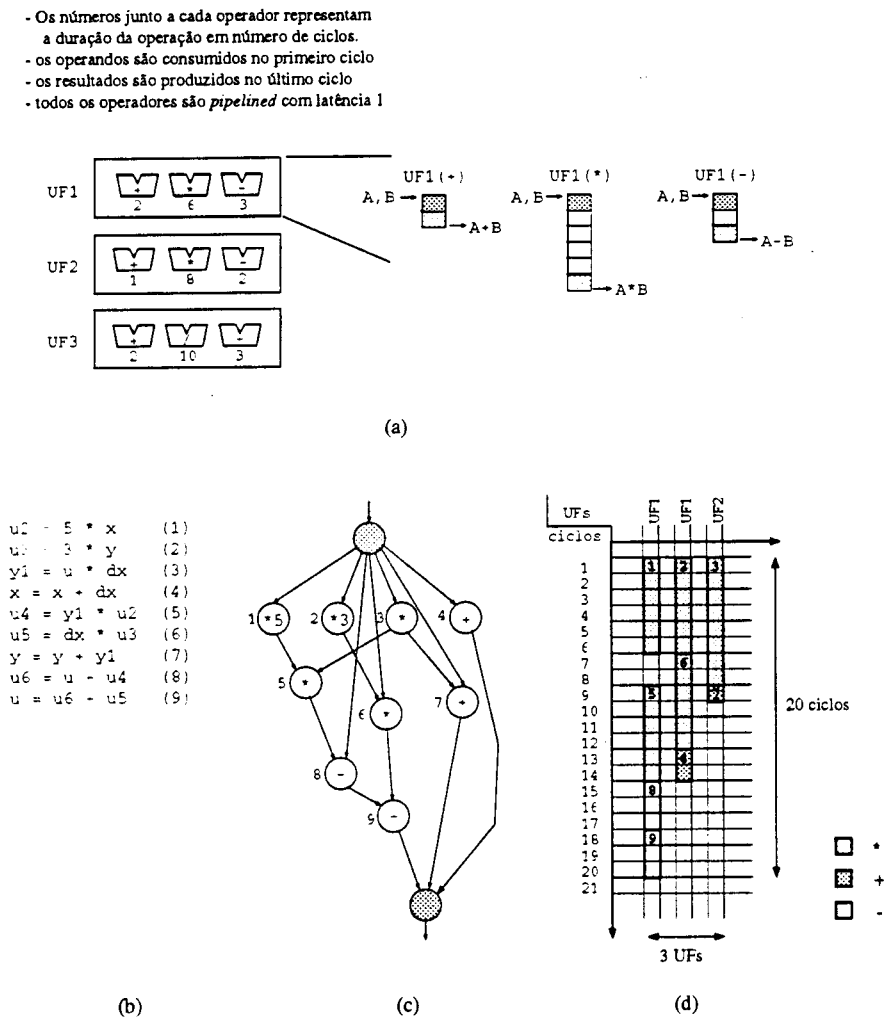


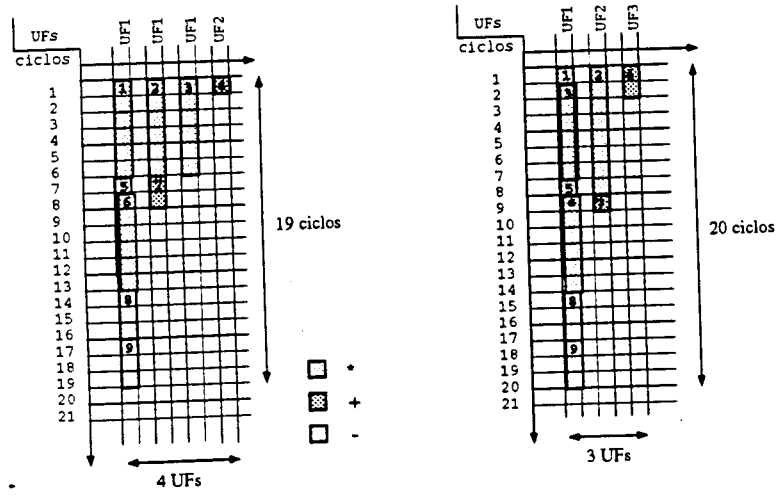
Figura 5.10: Representação de uma solução na aplicação de optimização desenvolvida baseada no algoritmo *Simulated Annealing*. Conjunto de unidades funcionais (a), problema exemplo (b), GDD que o representa (c) e representação de uma solução (d).

5.7.3 Solução inicial

Foram implementadas duas estratégias para obter uma solução inicial. A primeira é obtida realizando um sequenciamento ASAP (*As Soon As Possible*), atribuindo as operações ao primeiro ciclo em que podem ser executadas. A segunda abordagem consiste em distribuir de forma aleatória as operações pelos ciclos e pelas unidades funcionais, após determinar com a estratégia ASAP, um intervalo de ciclos para distribuir as operações. Para ambas as técnicas pode ser especificado um conjunto inicial de unidades funcionais ou podem ser seleccionadas durante a construção do sequenciamento.

Se não for definido um conjunto inicial de unidades funcionais, são escolhidas para a

solução inicial aquelas que forem necessárias para satisfazer o sequenciamento das operações ditado pela estratégia ASAP ou pela distribuição aleatória de operações no espaço de soluções. Se, por outro lado, for especificado um conjunto inicial de unidades funcionais, as operações para as quais não exista um operador disponível no ciclo escolhido, são atrasadas até ser encontrada uma unidade que a possa executar. Na figura 5.11 mostram-se as soluções iniciais obtidas com o sequenciamento ASAP para as duas situações referidas.



(a) (b)

Figura 5.11: Solução inicial obtida com sequenciamento ASAP sem fixar o conjunto inicial de unidades funcionais (a) e seleccionando para esse conjunto as unidades `UF1`, `UF2` e `UF3` (b). Note-se que na solução (a) as operações 5 e 6 são realizadas de forma *pipelined*, o mesmo acontecendo na solução (b) também com as operações 1 e 3

A escolha do conjunto de unidades funcionais inicial pode ser feita pelo utilizador, e deve incluir operadores capazes de executar todas as operações do problema. Uma selecção automática pode ser obtida resolvendo um modelo simples de PLI que obtém um conjunto de unidades funcionais com base num custo calculado para cada uma, que é função das características dos seus operadores e do número de operações de diferentes tipos que o problema a optimizar contém.

Pré-selecção automática de unidades funcionais

Para especificar um conjunto de unidades funcionais para construir a solução de partida para o processo iterativo é necessário garantir que são incluídos operadores para todos os tipos de operações contidos no problema a optimizar. Isso pode ser feito manualmente,

com base no conhecimento dos tipos de operações que o problema contém e dos conjuntos de operadores disponibilizados por cada unidade funcional. No sentido de automatizar essa tarefa, foi desenvolvido um modelo de PLI que permite seleccionar um conjunto inicial de unidades funcionais.

O modelo construído permite encontrar um conjunto de N unidades funcionais que oferecem um conjunto de operadores necessários para executar todos os tipos de operações que o problema contém, sendo minimizada uma função custo construída com base nas características do conjunto de unidades funcionais disponíveis e do problema a resolver. Essa função é construída como uma soma das variáveis de decisão que representam o número de unidades funcionais de cada tipo seleccionadas, pesada por um factor que pretende traduzir uma medida da eficácia dessa unidade funcional para o conjunto de operações do problema.

As M unidades funcionais (UF) são caracterizadas por c_{ij} que representa o número de ciclos que a operação do tipo i demora na UF j e O_{ij} como o número de operadores existentes na UF j capazes de realizar a operação i (considera-se que este número é 1 ou 0, consoante a UF j suporta ou não a operação do tipo i). O número de tipos diferentes de operações existentes no problema é representado por L. As variáveis de decisão X_j são inteiras e não negativas e representam o número de unidades funcionais do tipo j seleccionadas.

Dado um problema constituído por N_{op} operações, seja N_{op_i} o número de operações do tipo i. Estes dois parâmetros permitem calcular a percentagem de cada tipo de operação no problema, $P_{op_i} = N_{op_i}/N_{op}$. Somando, para cada unidade funcional, o produto deste parâmetro pelo número de ciclos que a operação i demora na unidade j, c_{ij}, obtém-se um factor custo para a unidade funcional j, C_{UF_j}, que é função do número de operações de cada tipo que o problema contém:

$$C_{UF_j} = \sum_{i=1}^{L} P_{op_i} \times c_{ij} \tag{5.19}$$

sendo assim a função objectivo a minimizar:

$$\sum_{j=1}^{M} X_j \times C_{UF_j} \tag{5.20}$$

As restrições dividem-se em dois tipos. Por um lado é imposto que o número de unidades seleccionadas tem de ser igual a N (equação (5.21)). Além disso, o conjunto de unidades funcionais tem de oferecer pelo menos um operador que consiga realizar cada tipo de operação que o problema tem. Definindo E_i como 1 se existir no problema a operação do tipo i e 0 caso contrário, então essa restrição pode ser expressa, para cada tipo de operação i,

pela restrição (5.22). Se não for encontrada uma solução admissível, então isso quer dizer que não é possível reunir num conjunto de N unidades funcionais operadores para todos os tipos de operações que o problema contém.

$$\sum_{j=1}^{M} X_j = N \tag{5.21}$$

$$\sum_{j=1}^{M} X_j \times O_{ij} \geq E_i, \text{ pata todos } i = 1 \ldots L \tag{5.22}$$

O modelo de PLI assim construído tem M variáveis inteiras e $L + 1$ restrições. A solução obtida serve para encontrar de forma automática um conjunto inicial de unidades funcionais para construir uma solução de partida para o processo iterativo realizado pela aplicação implementada.

5.7.4 Soluções adjacentes

Uma aplicação do algoritmo *Simulated Annealing* requer a geração e avaliação eficiente de soluções que sejam obtidas por modificações elementares da solução corrente. Estas são chamadas soluções adjacentes e definem a estrutura de vizinhança do problema. Da rapidez da sua geração e avaliação depende em grande parte o desempenho global deste algoritmo. Uma solução adjacente deve ser obtida à custa de transformações simples de uma solução, de forma a conduzirem a iterações para soluções próximas na região de soluções admissíveis.

Uma iteração é realizada escolhendo uma operação e deslocando-a no plano de soluções segundo uma direcção horizontal (deslocamentos H) ou vertical (deslocamentos V), como se mostra na figura 5.12. As iterações podem ser feitas de modo a garantir que são sempre satisfeitas as relações de precedência entre as operações, ou permitindo violar essas restrições, sendo isso escolhido pelo utilizador com a selecção de opções apropriadas. No entanto, iterando apenas por soluções admissíveis não é possível partir de soluções iniciais geradas aleatóriamente que são normalmente não admissíveis. Experiências realizadas mostram que se não for permitido violar as relações de precedência, na generalidade dos casos o processo iterativo converge para soluções piores e de forma mais lenta do que se forem permitidas soluções que violem essas restrições. A violação das relações de precedência é quantificada em ciclos, sendo obtida para cada operação como a soma do número de ciclos que todas as operações que dela dependem directamente devem ser atrasadas, de forma a eliminar essas violações.

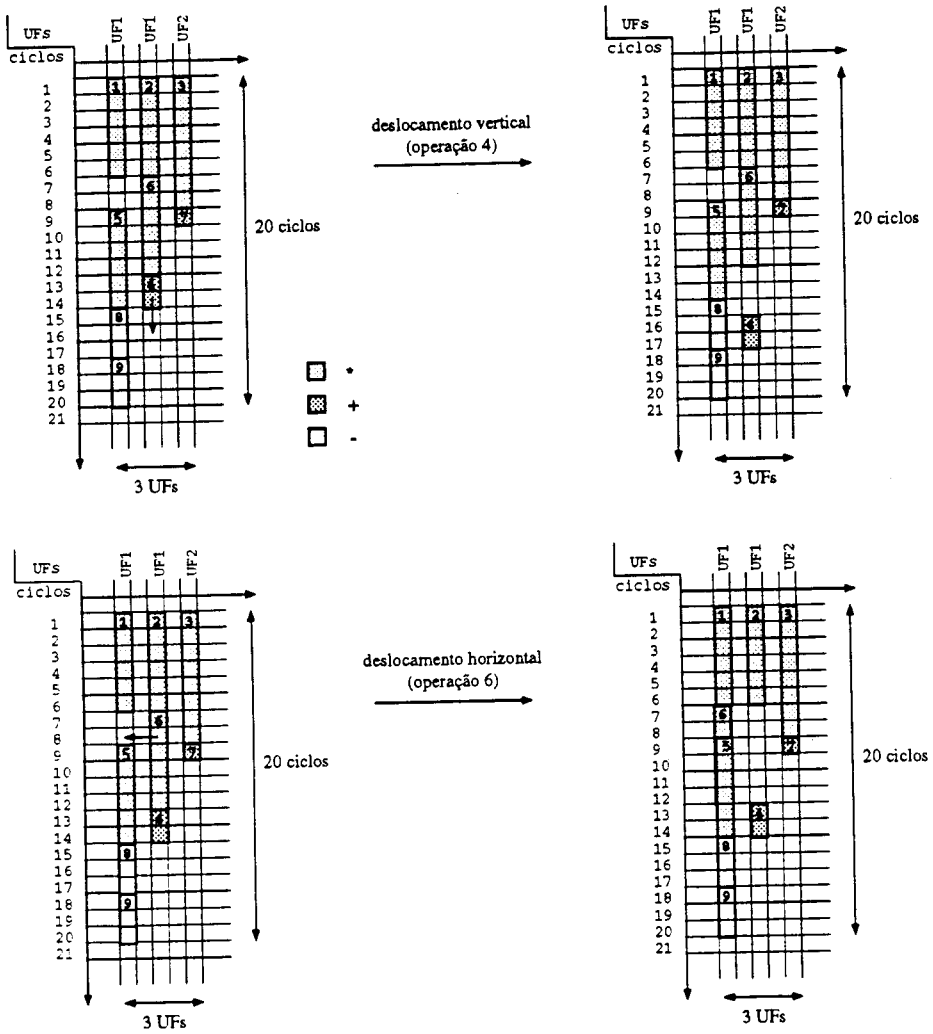


Figura 5.12: Iteração movendo verticalmente uma operação para outro ciclo (deslocamento V), ou movendo-a horizontalmente para outra unidade funcional (deslocamento H).

A selecção da operação a mover, do tipo de deslocamento e dos parâmetros desse deslocamento é feita aleatóriamente, usando uma função densidade de probabilidade uniforme. Isto significa que qualquer operação tem igual probabilidade de ser escolhida para realizar uma iteração, bem com pode ser seleccionado, com igual probabilidade, um deslocamento do tipo H ou V.

Deslocamentos verticais

Um deslocamento segundo o eixo vertical move uma operação no tempo, mantendo-a na mesma unidade funcional. Se não for permitido violar as relações de precedência entre

as operações, só são gerados deslocamentos verticais dentro do intervalo de ciclos em que a operação seleccionada pode ser movida. Para isso, é calculado o deslocamento máximo possível para cima e para baixo, em função dos ciclos a que estão atribuídas as operações imediatamente anteriores e posteriores no GDD (figura 5.13). O deslocamento vertical a realizar é obtido escolhendo, com igual probabilidade, qualquer ciclo dentro desse intervalo, embora limitando a amplitude máxima desse deslocamento a valores que são estabelecidos como parâmetros da aplicação.

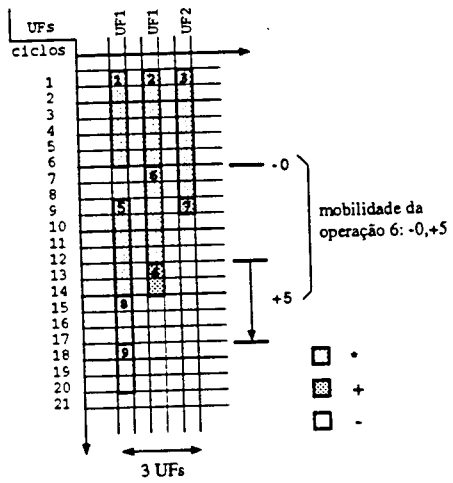


Figura 5.13: Mobilidade de uma operação seleccionada para deslocamento vertical (operação 6).

Se for permitida a iteração para soluções que violem aquelas restrições, então a determinação do deslocamento vertical a realizar não obriga a calcular a mobilidade da operação, sendo obtido como um número aleatório no intervalo definido pela amplitude máxima admitida para os deslocamentos verticais.

Na geração de soluções adjacentes por deslocamentos verticais, a operação deslocada é, em princípio, mantida na mesma unidade funcional. No caso dessa unidade funcional não estar disponível para o tipo de operação pretendida no período de tempo necessário, é seleccionada uma nova unidade funcional do mesmo tipo para acomodar a operação deslocada, como se mostra na figura 5.14.

Deslocamentos horizontais

Gerar uma solução adjacente movendo uma operação segundo o eixo horizontal consiste em manter essa operação no mesmo ciclo e atribuí-la a um tipo de unidade funcional, entre aquelas presentes na biblioteca que suportam a operação. A operação pode ser

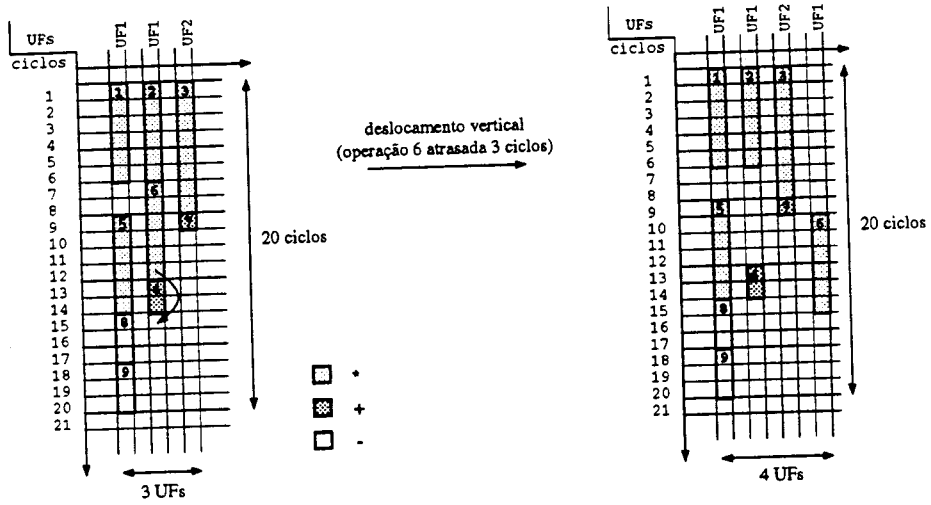


Figura 5.14: Selecção de uma nova unidade funcional (UF1), resultante do deslocamento vertical da operação 6.

deslocada para uma unidade de um tipo diferente ou para outra unidade do mesmo tipo a que estava atribuída. Se nenhuma unidade funcional desse tipo já seleccionada na solução puder realizar essa operação no período considerado é seleccionada uma nova unidade funcional desse tipo (figura 5.15).

Como cada operação pode ter diferentes durações em diferentes unidades funcionais, pode acontecer que ao mover uma operação para uma unidade funcional mais lenta, passem a ser violadas as relações de precedência entre as operações e por isso a nova solução resulte não admissível (figura 5.16). Se for permitida a iteração para soluções que violem essas restrições, é calculado o custo associado a essas violações. No exemplo apresentado na figura 5.16, a violação introduzida é quantificada em 2 ciclos, porque a operação 8, que deveria ser iniciada só após a operação 5 ser concluída, deve ser atrasada 2 ciclos para eliminar essa violação. Se apenas for permitido iterar por soluções admissíveis então as soluções geradas nessas condições são imediatamente rejeitadas.

5.7.5 Função objectivo

A função objectivo traduz o custo de uma solução e orienta o processo iterativo para optimizar a solução segundo os critérios por ela estabelecidos. Esse custo pode incluir várias medidas de qualidade da solução, dependendo do sentido de optimização desejado.

Considerando como alvo a arquitectura proposta, onde é fixo o número de unidades funcionais, o objectivo global é minimizar o tempo necessário à execução das operações que representam o programa a executar. Isto traduz-se em minimizar o número de ciclos,

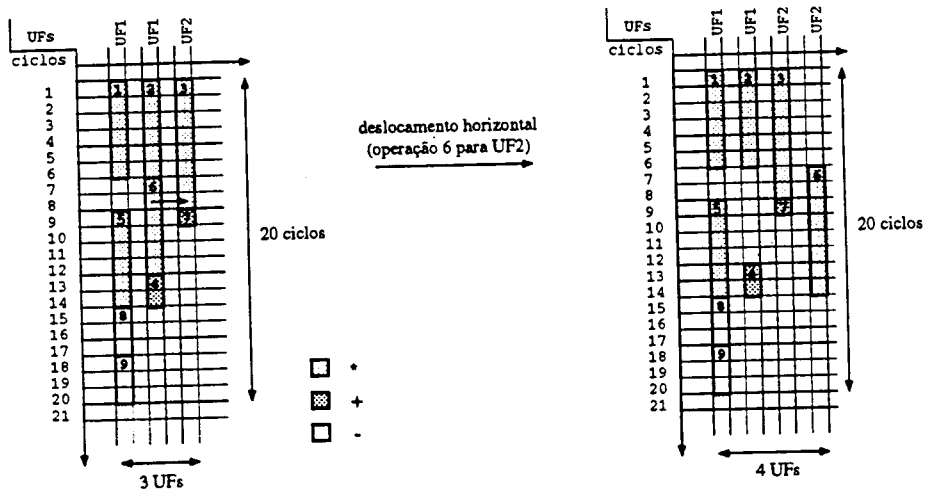


Figura 5.15: Selecção de nova unidade funcional do tipo UF2, provocado pelo deslocamento horizontal da operação 6.

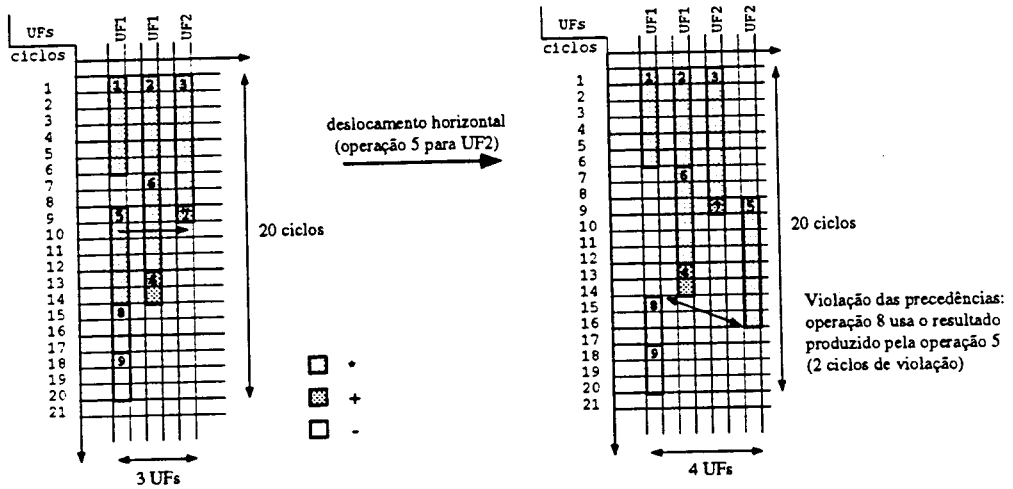


Figura 5.16: Geração de uma solução não admissível com um deslocamento horizontal de uma operação. Movendo a operação 5 para uma nova unidade funcional do tipo UF2, é violada em 2 ciclos a relação de precedência entre as operações 8 e 5.

o que corresponde a encolher o espaço de representação da solução segundo o eixo vertical, limitando-o, segundo o eixo horizontal, ao número máximo de unidades funcionais que a arquitectura suporta. É de notar que, como consequência da relação inversa geralmente verificada entre tempo de execução e complexidade do *hardware*, optimizar o problema segundo aquele critério resulta na tendência para o aumento do número de unidades funcionais (complexidade do *hardware*), o que é traduzido no alargamento do espaço

ocupado pela solução segundo o eixo horizontal.

Sendo assumidas outras plataformas alvo em que não seja fixo o número de unidades funcionais (por exemplo, na perspectiva da realização física de um ASIC), então outro objectivo que pode ser desejado optimizar consiste em minimizar a complexidade do *hardware* associado ao conjunto de unidades funcionais seleccionadas, traduzido pelo seu número ou área ocupada. A optimização segundo este objectivo corresponde a reduzir a largura do espaço que representa a solução, o que pela razão referida antes, conduz geralmente a um aumento do número de ciclos necessários para executar as operações.

A pesquisa de compromissos entre complexidade do *hardware* e tempo de execução requer o uso de funções objectivo que englobem simultâneamente os dois critérios referidos. A optimização simultânea segundo o espaço e o tempo corresponde a encolher o espaço das soluções simultâneamente segundo os dois eixos, e pode ser visto como minimizar a área ocupada pela distribuição das operações naquele espaço. Aplicações do problema geral de optimização aqui tratado no contexto de síntese de alto nível de circuitos digitais consideram geralmente critérios de optimização que estabelecem compromissos entre aqueles dois objectivos.

Na figura 5.17 mostram-se, de forma simplificada, os critérios globais de optimização que podem ser adoptados para o problema, tal como referidos acima.

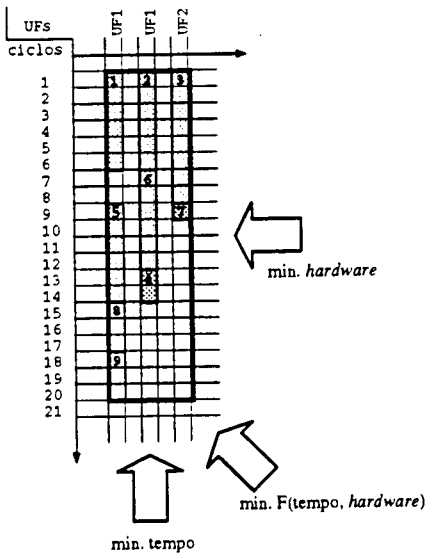


Figura 5.17: Representação dos critérios globais de optimização.

Na perspectiva de uma aplicação do algoritmo *Simulated Annealing*, deve ser possível re-avaliar eficientemente a função objectivo para cada nova solução que é gerada, já que disso depende em grande parte a rapidez de funcionamento do processo iterativo. A

função objectivo pode assumir qualquer forma, podendo mesmo ser modificada ao longo do processo iterativo. Uma função objectivo possível consiste numa soma dos vários parâmetros que caracterizam a solução, pesada por factores constantes que permitem definir o sentido pretendido para a optimização. No entanto, usando simultâneamente parâmetros que medem características que não podem ser comparadas directamente, torna-se difícil estabelecer critérios para definir de forma objectiva esses pesos. Não existe geralmente uma relação mensurável entre a unidade de tempo (ciclo) e a unidade de custo associada às unidades funcionais seleccionadas.

Durante o processo de geração e avaliação de soluções, a função objectivo deve conseguir traduzir variações na qualidade da solução resultantes da transformação efectuada com a iteração. Pode ver-se com os exemplos usados para ilustrar a forma com são realizadas as iterações (figuras incluídas na secção 5.7.4), que se for usado apenas o número de ciclos do sequenciamento e o número de unidades funcionais seleccionadas, podem ser geradas muitas soluções diferentes mas que não provocam qualquer alteração no valor da função objectivo. Embora o objectivo global seja efectivamente minimizar o número de ciclos necessários à execução das operações, são incluídas outras medidas indirectas deste critério, que são mais sensíveis aos deslocamentos de operações considerados do que o número total de ciclos do sequenciamento. São também mantidos outros parâmetros que medem características variadas da solução (número de violações, número de unidades funcionais seleccionadas, por exemplo) Sempre que é gerada uma nova solução, é calculada a variação resultante em cada um desses parâmetros e é recalculado o valor da função objectivo.

5.7.6 Implementação

A aplicação do algoritmo *Simulated Annealing* ao problema geral de optimização foi realizada em C++ (aproximadamente 4500 linhas de código fonte), recorrendo a uma biblioteca que implementa estruturas de dados eficientes e algoritmos para as manipular (LEDA—*Library of Efficient Data types and Algorithms* [Näh95]). O programa é controlado interactivamente com um interpretador de comandos e é apresentada numa janela gráfica a evolução do valor da função objectivo no decorrer do processo de optimização realizado pelo algoritmo *Simulated Annealing*. A presente implementação foi compilada com o compilador de C da GNU, e corre numa estação de trabalho HP715/50, com o sistema operativo HP-UX.

A aplicação desenvolvida suporta um conjunto alargado de parâmetros de configuração que são especificados pelo utilizador num ficheiro de texto. Entre eles incluem-se pesos a

usar na função objectivo, critérios para construção da solução inicial e valores por omissão para alguns dos parâmetros que intervêm no controlo do algoritmo *Simulated Annealing*.

O sistema desenvolvido permite explorar e avaliar variadas estratégias de aplicação do algoritmo *Simulated Annealing* ao problema de optimização considerado. Não pretendendo ser uma ferramenta que ofereça um processo de optimização completamente automático, a sua utilização requer grande interactividade por parte do utilizador, especialmente na especificação dos parâmetros que controlam a técnica de optimização implementada e na definição da função objectivo.

Especificação do problema

Como foi já dito, o problema a optimizar é descrito como um bloco básico de instruções do tipo *assembly*, usando um conjunto de códigos de instrução que são especificados num dos ficheiros de configuração da aplicação. As instruções consideradas são do tipo:

```
OPCODE  res, op1, op2, op3
```

onde `OPCODE` é o código que representa o tipo de operação, `res` é a variável a que é atribuído o resultado da operação e `op1`, `op2` e `op3` são os operandos, cujo número máximo foi fixado em 3. O conjunto de instruções suportado é descrito num ficheiro de texto, e a cada uma é atribuído um carácter que serve para a associar aos operadores presentes na biblioteca de unidades funcionais. Para além das instruções esperadas na descrição do problema, devem também ser definidas duas operações que representam os nós de entrada e saída do GDD, com códigos respectivamente `ENTRY` e `EXIT`. No apêndice D são listados os exemplos que foram experimentados com a aplicação desenvolvida, na linguagem *assembly* suportada.

Especificação das unidades funcionais

As características do conjunto de configurações disponíveis para as unidades funcionais são especificadas num ficheiro de texto (biblioteca de unidades funcionais). Neste ficheiro é definido, para cada unidade funcional, o número de operadores suportados e as características de cada um: número de operandos, duração, latência e ciclos de consumo de cada operando. A cada operador é também atribuído um carácter que representa o tipo de operação que realiza, e que serve para o associar às instruções suportadas no código *assembly* em que é especificado o problema a optimizar. A descrição de uma unidade funcional inclui também uma matriz de custos de reconfiguração, mas que só é considerada quando são assumidas unidades funcionais reconfiguráveis dinamicamente, o que é abordado mais à frente na secção 5.7.8.

Na figura 5.18 mostra-se a descrição de uma unidade funcional ALU, com um custo igual a 300 e suportando 4 operações diferentes: ADD, MUL SUB e ADDSUB. O operador ADD executa instruções que sejam identificadas, na especificação da linguagem *assembly* suportada, pelo carácter +. Tem uma duração de 2 ciclos, uma latência de um ciclo e consome os seus dois operandos no ciclo 0 (primeiro ciclo); o operador ADDSUB realiza as operações identificadas pelo carácter a, tem uma duração de 4 ciclos, consome o primeiro e segundo operandos no ciclo 0, mas só necessita do terceiro operando no ciclo 2.

```
************************************
# Para cada unidade funcional:
#   nome_UF  custo_area  numero_de_operacoes
#     para cada operacao:
#     nome codigo duracao latencia numero_operandos
#       para cada operando:
#         ciclo_consumo
#     ciclo_producao_do_resultado
************************************
# UF ALU (custo 300, 4 operacoes)
ALU 300 4
# operacao ADD:
ADD + 2 2 2
0 0
1
# operacao MUL:
MUL * 6 1 2
0 0
5
# operacao SUB:
SUB - 3 1 2
0 0
2
# operacao ADDSUB:
ADDSUB a 4 1 3
0 0 2
3
```

Figura 5.18: Especificação das características de uma unidade funcional.

Na representação interna do problema são usadas duas operações artificiais que correspondem aos nós de entrada e saída do GDD (designadas por ENTRY e EXIT, respectivamente). A operação que representa o início do problema é colocada inicialmente no ciclo 0 e nunca é seleccionada durante o processo de geração de soluções adjacentes. A operação EXIT representa o fim do problema e o ciclo a que é atribuída é usado como medida do número de ciclos de sequenciamento de uma solução. Por este motivo, esta operação pode ser escolhida para gerar soluções adjacentes, embora só sejam permitidos nesse caso deslocamentos verticais. Para ser tratada como qualquer outra operação durante o processo de optimização, deve existir sempre uma unidade funcional que contenha um operador que a suporte e que não deve ter qualquer custo associado. Para isso, optou-se por impor que a primeira entrada numa biblioteca de unidades funcionais deve representar uma unidade com um custo associado nulo, e cujo primeiro operador suporte a operação EXIT.

Para contemplar operações que não são realizadas nas unidades funcionais (por exem-

plo deslocamento de dados), devem ser definidos operadores em unidades funcionais com custo nulo que suportem esse tipo de operação, embora podendo ser associados tempos de execução diferentes de zero.

Função objectivo

A função objectivo é construída como uma função em C++ compilada separadamente e ligada ao programa que implementa a aplicação desenvolvida. Devem ser criadas duas funções, com nomes `FO()` (a função objectivo propriamente dita) e `FO_setup()` que é invocada com um comando da aplicação para definir parâmetros que sejam requeridos pela função objectivo (por exemplo, pesos para afectar as várias variáveis disponíveis para a construir). A função objectivo recebe como argumentos as variáveis que caracterizam a solução, e que são actualizadas automaticamente pelas funções que geram as soluções adjacentes. Essas variáveis são:

- `seqciclos` - número de ciclos do sequenciamento: é numericamente igual ao ciclo a que é atribuída a operação `EXIT`. Esta variável só é afectada pelos deslocamentos verticais da operação `EXIT` e por isso não reflecte variações na qualidade global da solução resultantes de deslocamentos das outras operações do problema.

- `execiclos` - número de ciclos em que existe pelo menos uma operação em execução: este número é sempre menor ou igual a `seqciclos`, sendo a diferença entre as duas variáveis igual ao número de ciclos que não são utilizados para executar operações. No espaço bidimensional usado para representar as soluções, esses ciclos correspondem a linhas que não são abrangidas por nenhuma operação. Na solução final, esse número deverá ser zero, embora qualquer solução em que isso não aconteça possa ser facilmente transformada noutra com nenhum ciclo vazio, removendo essas linhas do espaço de soluções e decrementando os ciclos de sequenciamento das operações colocadas para além do que é removido. Esta variável representa uma medida indirecta do número de ciclos necessários para o sequenciamento. Ao contrário de `seqciclos`, reflecte alterações resultantes de iterações realizadas sobre operações diferentes de `EXIT`.

- `totciclos` - número total de ciclos usados para executar operações: representa o somatório dos ciclos que no conjunto de todas as unidades funcionais seleccionadas são efectivamente usados para executar operações. Este custo pode ser usado para obter uma medida do número de ciclos não usados no conjunto das unidades funcionais seleccionadas, sendo calculado como a diferença entre a área ocupada

no espaço de soluções (produto de `execiclos` pelo número de unidades funcionais seleccionadas) e `totciclos`.

- `somciclos` - somatório dos ciclos finais de todas as operações: esta variável pretende reflectir variações na qualidade do sequenciamento entre duas soluções adjacentes. O valor desta variável é sempre afectado por deslocamentos verticais de operações e em deslocamentos horizontais, sempre que uma operação é movida entre unidades funcionais cujos operadores a realizem com durações diferentes.

- `totviol` - somatório das violações de precedências de todas as operações: a violação das precedências de uma operação é quantificada como o somatório das violações de cada um dos seus sucessores directos. Para as operações sucessoras que não satisfazem as relações de precedência, são calculadas as diferenças entre o ciclo mais cedo permitido e o ciclo em que a operação está efectivamente atribuída. Numa solução admissível este número deverá ser 0, embora em soluções finais que não satisfaçam essa condição, as violações destas restrições podem ser eliminadas atrasando as operações que as provocam de um número de ciclos adequado.

- `numinst` - número de unidades funcionais seleccionadas: esta variável não contabiliza as unidades funcionais que tenham sido definidas com custo nulo.

- `custoarea` - custo das unidades funcionais: é a soma dos custos definidos para as unidades funcionais seleccionadas. Tem apenas interesse em aplicações onde se pretende incluir na função objectivo uma medida da complexidade do *hardware* associado ao conjunto das unidades seleccionadas.

A flexibilidade permitida pela combinação destes parâmetros possibilita naturalmente a construção de uma variedade muito grande de funções objectivo, sendo impraticável compará-las de forma exaustiva. Após experiências preliminares, optou-se por uma função objectivo que é uma soma das variáveis `seqciclos`, `execiclos`, `totviol` e `numinst`, pesadas por factores constantes que são especificados pelo utilizador no ficheiro de configuração da aplicação. Para conduzir o processo iterativo para soluções finais que não ultrapassem o número máximo de unidades funcionais, é atribuída uma penalização a cada unidade funcional seleccionada para além do número máximo especificado, usando a variável `numinst`. A soma dos ciclos finais de todas as operações, `somciclos`, permitiria em princípio, favorecer soluções onde as operações fossem terminadas mais cedo, por ser mais sensível do que `seqciclos` e `execiclos` a variações resultantes das transformações realizadas nas soluções. No entanto, verificou-se que se esta variável for incluída na função

objectivo considerada, as soluções finais atingidas são na generalidade dos casos piores, convergindo muitas vezes para soluções com número elevado de violações das relações de precedência ou ultrapassando o número máximo permitido para as unidades funcionais.

Como foi dito antes, a medida real do número de ciclos necessários ao sequenciamento é dada por `execiclos`. Incluir `seqciclos` na função objectivo permite que o espaço de soluções não se estenda ao longo de um número demasiado grande de ciclos. A variável `totviol` permite controlar o grau de não admissibilidade que se permite ter nas soluções. Se lhe for atribuído um peso suficientemente elevado, então nunca serão aceites soluções que violem as restrições de precedência. Por outro lado, se esse peso for nulo, não há nada que oriente o processo iterativo no sentido de convergir para soluções finais admissíveis. O resultado de experiências feitas com diferentes combinações de pesos para estas variáveis, conduziu à selecção dos valores que se listam na tabela 5.7.6 e que foram usados com os exemplos experimentados referidos mais à frente.

Tabela 5.1: Pesos usados na função objectivo.

	`seqciclos`	`execiclos`	`totviol`	`numinst`
Pesos	2	5	20	50

Na figura 5.19 mostra-se uma solução do problema exemplo e os valores dos vários parâmetros referidos antes, que são usados para medir a qualidade da solução.

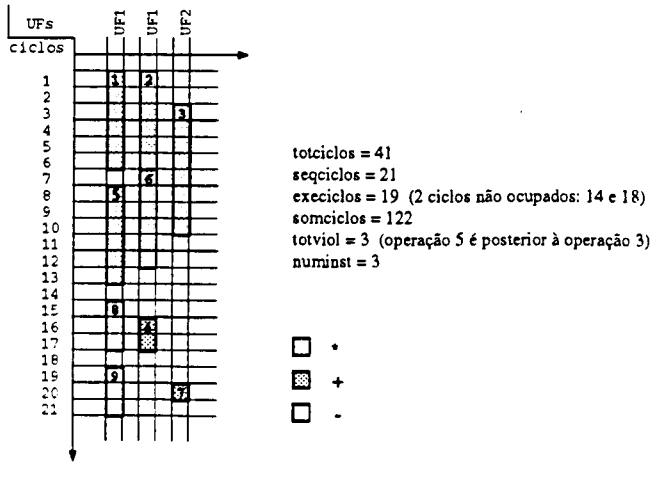


Figura 5.19: Medidas de uma solução do problema exemplo.

Temperatura inicial e estratégia de arrefecimento

Na aplicação desenvolvida é executado um decrescimento exponencial da temperatura, controlado por um parâmetro α segundo a função representada por (5.18), e uma função de probabilidade para aceitação de soluções igual à da equação (5.16), apresentadas na secção 5.7.1.

Os parâmetros T_0 (temperatura inicial) e α (coeficiente de arrefecimento) que controlam o algoritmo *Simulated Annealing*, são especificados pelo utilizador. Para obter um valor para T_0, é possível simular um número especificado de iterações, partindo da solução inicial, sendo calculada a percentagem de soluções geradas que pioram a função objectivo e que são aceites para aquele valor de T_0. Das experiências realizadas conclui-se que valores de T_0 que conduzam a percentagens de aceitações de soluções piores entre 15% e 30% representam bons pontos de partida para o processo iterativo. O valor final atingido para a função objectivo mostrou não ser muito sensível para valores de T_0 entre limites obtido pelo processo referido, embora valores elevados necessitem naturalmente de um maior número de iterações para atingir a solução final. O melhor valor para o parâmetro α que controla a função de arrefecimento referida é difícil de determinar. Valores empíricos ou obtidos por experimentação que mostraram conduzir a bons resultados para uma grande variedade de aplicações estão geralmente compreendidos entre 0.8 e 0.99 [Ree95].

Nos exemplos experimentados, foram usados valores para α entre 0.95 e 0.99. O valor mais baixo requer um número mais elevado de iterações para cada valor de temperatura do que com o outro extremo, que corresponde a um arrefecimento mais lento podendo ser efectuado um menor número de iterações em cada valor de T. Das experiências realizadas com os problemas ensaiados, verificou-se que na generalidade dos casos foram conseguidos melhores resultados para valores de α próximos de 0.95, embora em algumas situações as melhores soluções tenham sido obtidas com valores iguais a 0.99. Valores muito distantes daquele intervalo conduzem quase sempre a soluções finais que são claramente piores do que as conseguidas com os valores referidos.

Número de iterações e critério de paragem

O critério de paragem e o número de iterações para cada valor de temperatura são igualmente estabelecidos pelo utilizador, com base nas características do problema e da rapidez de arrefecimento ditada pelo valor escolhido para α.

É efectuado um número de iterações constante para cada valor de temperatura que depende da rapidez de arrefecimento definida pelo valor de α. Para valores de α pequenos (0.95) deve ser seleccionado para esse número de iterações um valor entre 8 a 15 vezes

o número de operações do problema. Com valores mais elevados para α (próximos de 0.99) pode ser realizado um número de iterações inferior, tipicamente entre 1 a 5 vezes o número de operações.

A paragem automática do processo iterativo é estabelecida quando não há modificação do valor da função objectivo durante todas as iterações realizadas para um número dado de valores de temperatura consecutivos. Este número pode também ser modificado pelo utilizador e é considerado por omissão é igual a 10.

Durante a execução do algoritmo, é representada numa janela gráfica a evolução do valor da função objectivo e da temperatura com o número de iterações. Para cada iteração é desenhado um ponto de imagem em cor verde, vermelha ou amarela, consoante é melhorado, piorado ou mantido o valor da função objectivo, respectivamente. A evolução da mancha criada com o conjunto de pontos que representam os valores da função objectivo no decorrer do processo iterativo permite ao utilizador avaliar visualmente o comportamento do algoritmo e, se desejado, antecipar os critérios de paragem estabelecidos e interromper o processo iterativo.

5.7.7 Resultados

A aplicação desenvolvida foi experimentada com 4 problemas, com características diferentes e complexidade variável. Dois são exemplos utilizados regularmente para comparação de desempenho de técnicas de sequenciamento e selecção de unidades funcionais em síntese de alto nível. O primeiro (*HAL*) foi extraído de [PK89] e é o núcleo de um ciclo de resolução de uma equação diferencial. Este conjunto de operações foi usado ao longo deste capítulo, para ilustrar a forma como foi implementado o algoritmo *Simulated Annealing*. O segundo (*ellip*) implementa um filtro elíptico de 5ª ordem [KWK85]. O terceiro exemplo considerado (*inter*) é formado por uma sequência de operações que calcula o ponto de intersecção de duas rectas, sendo cada uma definida por um par de pontos em coordenadas cartesianas. Esta é uma das tarefas usadas intensivamente em aplicações de empacotamento automático de figuras irregulares, para calcular intersecções de polígonos com vista à determinação de sobreposições. Finalmente, foi usado um segmento de instruções *assembly* extraído do código obtido por compilação do programa em C listado no apêndice A, usando o compilador que foi construído para o processador THD e apresentado no capítulo 3. Este problema (*hdll*) contém as instruções *assembly* em que foi traduzida a iteração do ciclo `for(...)` da função `iteration()`, para `i=2`. As características principais desses problemas são resumidas na tabela 5.2 e no apêndice D são apresentadas as suas listagens na linguagem *assembly* considerada pela aplicação desenvolvida.

Tabela 5.2: Características principais dos 4 problemas experimentados com a aplicação desenvolvida.

nome	operações	tipos de operações	nº de ramos do GDD
HAL	9	3	21
ellip	34	2	80
inter	19	4	47
hd1l	106	10	294

A biblioteca de unidades funcionais usada é listada no apêndice C segundo o formato explicado anteriormente. Para além da unidade funcional de custo nulo criada para acomodar a operação artificial `EXIT` e as operações de deslocamento de dados (`MOV`), a biblioteca contém 5 unidades funcionais com conjuntos diferentes de operações e níveis de desempenho variados, que contemplam todos os tipos de operações existentes nos problemas experimentados. As características das unidades funcionais usadas são resumidas na tabela 5.3.

A aplicação desenvolvida foi usada para optimizar os 4 problemas de teste, usando várias combinações dos parâmetros de controlo do algoritmo e função objectivo, partindo de diferentes soluções iniciais. Pretendeu-se com esse conjunto de experiências obter uma configuração para as variáveis de controlo, solução inicial e função objectivo que conseguisse atingir boas soluções para o conjunto dos 4 problemas ensaiados.

Foram usadas soluções iniciais obtidas por diferentes processos, já referidos na secção 5.7.3. O primeiro (ASAP) consiste em construir um sequenciamento ASAP, sem impor restrições no número de unidades funcionais. A solução inicial obtida deste modo não viola as restrições de precedência e é seleccionado o número de unidades funcionais que for necessário para satisfazer o sequenciamento. Na construção desta solução, é sempre escolhida a unidade funcional mais rápida que possa realizar a operação, no ciclo determinado pela estratégia ASAP. Uma solução inicial diferente (ASAP3) foi obtida realizando o mesmo tipo de sequenciamento, mas fixando um conjunto inicial de 3 unidades funcionais. O terceiro processo (RANDOM) distribui as operações de forma aleatória, ao longo de um número de ciclos que é obtido, aumentando em 40% o que resulta do sequenciamento ASAP.

Após várias experiências preliminares em que foram usadas diferentes combinações dos parâmetros disponíveis para construir a função objectivo, optou-se por uma soma pesada desses parâmetros, tal como foi referido antes, com os pesos apresentados na tabela 5.7.6.

Como não são conhecidas outras soluções para os problemas experimentados, dentro

Tabela 5.3: Características da biblioteca de unidades funcionais. As colunas OP1, OP2 e OP3 representam, para cada operador, os ciclos de consumo de cada operando. O ciclo em que o resultado é produzido coincide com o último ciclo de execução de cada operação.

Unidade	operador	duração	latência	OP1	OP2	OP3
EXITMOVE						
	EXIT	1	0	0		
	MOV	1	0	0		
ALU						
	ADD	2	2	0	0	
	SUB	3	3	0	1	
	MUL	8	4	0	0	
	DIV	10	5	0	0	
	MULADD	10	4	0	0	8
	MULSUB	11	4	0	0	8
	DIVADD	12	5	0	0	10
	DIVSUB	13	5	0	0	10
	MAX	3	3	0	1	
	MIN	3	3	0	1	
GENERICALU						
	ADD	1	1	0	0	
	SUB	2	2	0	0	
	MUL	16	16	0	0	
	DIV	16	16	0	0	
	MULADD	18	16	0	0	16
	MULSUB	18	16	0	0	16
	DIVADD	18	16	0	0	16
	DIVSUB	18	16	0	0	16
	MAX	3	3	0	1	
	MIN	3	3	0	1	
MULPIPE						
	ADD	2	1	0	0	
	SUB	2	1	0	0	
	MUL	10	1	0	0	
	MULADD	12	1	0	0	10
	MULSUB	12	1	0	0	10
	MAX	2	1	0	0	
	MIN	2	1	0	0	
MULDIVPIPE						
	ADD	1	1	0	0	
	MUL	7	1	0	0	
	DIV	9	1	0	0	
ADDSUBMUL						
	ADD	3	1	0	0	
	SUB	3	1	0	0	
	MUL	16	16	0	0	
	MAX	3	1	0	0	
	MIN	3	1	0	0	

das assunções que foram consideradas, foram comparadas as soluções finais atingidas com o algoritmo *Simulated Annealing* com as conseguidas usando pesquisa local pura, i.e., gerando soluções adjacentes e aceitando só aquelas que melhoram a função objectivo. Isto é fácil de realizar com a aplicação desenvolvida, bastando usar para T_0 um valor muito próximo de zero de forma a tornar praticamente nula a probabilidade de aceitar soluções que piorem a função objectivo.

Para cada problema são apresentados os resultados obtidos partindo das 3 soluções iniciais referidas acima. São apresentadas as melhores soluções que foram obtidas para cada tipo de solução inicial, de entre as que foram ensaiadas com diferentes combinações dos parâmetros de controlo do algoritmo. Nas tabelas que resumem os resultados, a coluna "ciclos" corresponde ao valor do parâmetro `execiclos`, que conta o número de ciclos da solução em que é executada pelos menos uma operação.

HAL

Tabela 5.4: Resultados obtidos para o exemplo *HAL*.

Solução inicial				parâmetros S.A.			Solução final			
Método	ciclos	UFs	viol	T_0	α	Iter/T	ciclos	UFs	viol	tempo exec.(s)
ASAP	19	4	0	100	0.95	100	19	3	0	49
ASAP3	20	3	0	100	0.95	100	19	3	0	38
RANDOM	27	2	51	100	0.99	50	19	3	0	35
ASAP	19	4	0	$T_0 \rightarrow 0$ (pesquisa local)			19	3	0	20
RANDOM	27	2	51	$T_0 \rightarrow 0$ (pesquisa local)			18	3	1	23

Este problema é o mais simples de entre os que foram experimentados, e contém 9 operações: 2 adições, 2 subtracções e 5 multiplicações. Dada a sua simplicidade, as soluções finais conseguidas partindo de soluções iniciais obtidas pelas diferentes estratégias são semelhantes, e muito próximas da solução inicial conseguida com o sequenciamento ASAP3 (tabela 5.4). Neste exemplo, pesquisa local pura mostrou conseguir resultados equivalentes aos atingidos por *Simulated Annealing* e tempos de execução significativamente mais curtos.

ellip

Para este problema, as melhores soluções finais foram conseguidas com *Simulated Annealing* partindo das soluções iniciais ASAP3 e RANDOM e foram sempre melhores do que as conseguidas com pesquisa local pura (tabela 5.5). Note-se que, apesar da solução inicial RANDOM apresentar um número elevado de violações das restrições de

Tabela 5.5: Resultados obtidos para o exemplo *ellip*.

Solução inicial				parâmetros S.A.			Solução final			
Método	ciclos	UFs	viol	T_0	α	Iter/T	ciclos	UFs	viol	tempo exec.(s)
ASAP	33	6	0	50	0.95	200	33	4	2	120
ASAP3	39	3	0	20	0.95	200	34	3	0	149
RANDOM	37	5	376	100	0.95	300	34	3	0	132
ASAP	33	6	0	$T_0 \rightarrow 0$ (pesquisa local)			33	4	0	243
RANDOM	37	5	376	$T_0 \rightarrow 0$ (pesquisa local)			30	3	15	240

precedência, é conseguida uma solução final admissível num tempo ligeiramente menor do que é necessário partindo da solução inicial ASAP3, que verifica à partida essas restrições. Além disso, partindo da solução inicial ASAP, nunca foram conseguidas soluções finais admissíveis, por ser sempre ultrapassado o número máximo estabelecido de unidades funcionais e também por serem violadas as restrições de precedência.

inter

Tabela 5.6: Resultados obtidos para o exemplo *inter*.

Solução inicial				parâmetros S.A.			Solução final			
Método	ciclos	UFs	viol	T_0	α	Iter/T	ciclos	UFs	viol	tempo exec.(s)
ASAP	44	4	0	100	0.99	50	41	4	4	98
ASAP3	46	3	0	300	0.95	300	45	3	2	132
RANDOM	46	5	333	300	0.95	300	46	3	0	112
ASAP	44	4	0	$T_0 \rightarrow 0$ (pesquisa local)			44	4	0	248
RANDOM	46	5	333	$T_0 \rightarrow 0$ (pesquisa local)			43	4	1	250

Os resultados obtidos para este problema (tabela 5.6) mostram que a melhor solução final atingida com *Simulated Annealing* foi obtida partindo da solução inicial gerada aleatóriamente. Tal como no exemplo anterior, não foram conseguidas soluções finais admissíveis com pesquisa local pura. É interessante notar que a melhor solução conseguida a partir da solução inicial RANDOM é equivalente, em termos do número de ciclos e admissibilidade, à solução inicial obtida pelo método ASAP3 e que esta não foi melhorada com *Simulated Annealing*. Neste exemplo, o conjunto de unidades funcionais que foi escolhido para construir a solução inicial ASAP3 coincidiu com o que foi obtido pelo algoritmo *Simulated Annealing* a partir da solução inicial RANDOM. É de notar também que a solução inicial obtida com ASAP foi piorada com *Simulated Annealing* e não foi melhorada com pesquisa local pura.

hd1l

Tabela 5.7: Resultados obtidos para o exemplo *hd1l*.

Solução inicial				parâmetros S.A.			Solução final			
Método	ciclos	UFs	viol	T_0	α	Iter/T	ciclos	UFs	viol	tempo exec.(s)
ASAP	266	9	0	300	0.95	800	249	6	20	340
ASAP3	496	3	0	300	0.95	1000	408	3	0	310
RANDOM	301	7	12.7 K	300	0.95	1000	123	7	173	470
ASAP3	496	3	0	$T_0 \rightarrow 0$ (pesquisa local)			455	3	0	520
RANDOM	301	7	12.7 K	$T_0 \rightarrow 0$ (pesquisa local)			225	7	168	570

Este exemplo contém um número de operações significativamente superior ao dos anteriores e 10 tipos de operações diferentes, o que o torna substancialmente mais complexo. A função objectivo usada com este exemplo foi alterada em relação à que foi usada nos exemplos anteriores. Com o mesmo valor (50) para a penalização do número de unidades funcionais seleccionadas para além do máximo permitido, nunca foram atingidas soluções finais em que o número de unidades funcionais não excedesse esse limite, excepto partindo de ASAP3 e usando pesquisa local pura. Isso aconteceu mesmo com soluções iniciais ASAP3, onde essa restrição era já verificada à partida. Aumentando essa penalização para 5000, só foram conseguidas soluções finais admissíveis partindo de soluções iniciais já admissíveis nesse critério (ASAP3), tendo sido conseguidos resultados melhores com *Simulated Annealing* do que com pesquisa local pura.

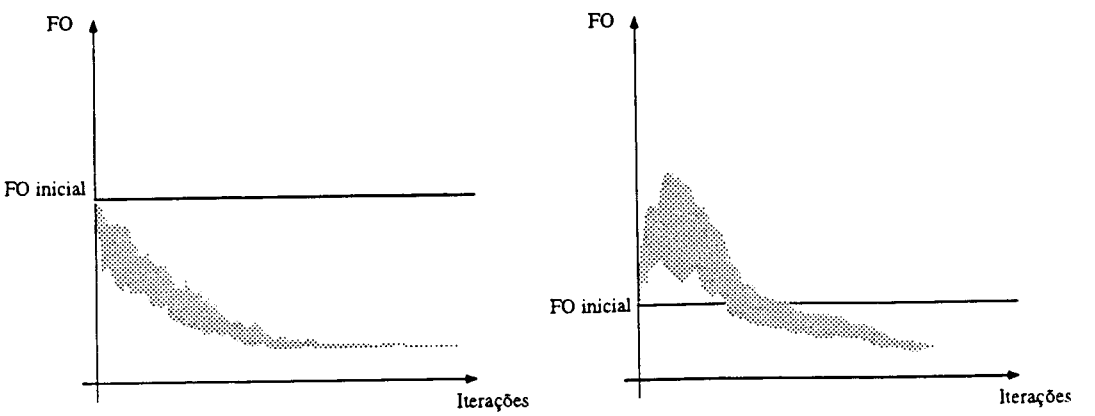
Os resultados obtidos são resumidos na tabela 5.7. É de referir que o conjunto de unidades funcionais na solução final obtida com *Simulated Annealing* é diferente do que é especificado para construir a solução inicial ASAP3, o que não acontece se for usada pesquisa local pura. Verificou-se que valores para aquela penalização em torno do valor referido representam um bom compromisso entre penalizações demasiado elevadas que não permitem a modificação da selecção inicial do conjunto de unidades funcionais e valores demasiado baixos que conduzem a soluções finais em que o número de unidades funcionais é superior ao máximo admissível.

Considerações finais sobre os resultados obtidos

Os resultados obtidos com os 4 exemplos experimentados permitem tecer algumas considerações finais. Pela diferença verificada no desempenho da aplicação desenvolvida entre o conjunto dos 3 primeiros problemas experimentados e o último que é significativamente mais complexo, são divididas em duas partes as conclusões aqui apresentadas.

Para os 3 primeiros problemas, pode concluir-se que os resultados obtidos partindo de soluções iniciais construídas de forma aleatória são geralmente melhores do que se forem usadas soluções iniciais obtidas com o sequenciamento ASAP. Neste caso, dos 3 exemplos ensaiados, só no primeiro (o mais simples) foi conseguida uma solução final admissível. Tomando como ponto de partida soluções RANDOM foram sempre obtidas soluções finais admissíveis que mostraram ser melhores ou pelo menos iguais às conseguidas partindo de ASAP ou ASAP3. Na generalidade dos casos, o algoritmo *Simulated Annealing* mostrou um desempenho melhor do que pesquisa local pura.

A forma como a função objectivo evolui durante o processo iterativo mostra comportamentos muito diferentes, dependendo do tipo de solução inicial de que se parte. Partindo de ASAP ou ASAP3, a solução é inicialmente "estragada", resultando num aumento do valor da função objectivo durante a parte inicial do processo iterativo. Após essa fase, cuja duração, em número de iterações, depende essencialmente do valor da temperatura inicial e da rapidez de arrefecimento, o valor da função objectivo começa a decrescer até ser atingido o valor correspondente à solução final. Partindo de uma solução inicial obtida aleatóriamente, que é por natureza má, são conseguidas geralmente melhores soluções finais com tempos de execução que não são muito diferentes dos necessários partindo de ASAP ou ASAP3. Na figura 5.20 mostram-se os dois comportamentos típicos da variação do valor da função objectivo durante o processo iterativo, partindo de soluções iniciais ASAP ou ASAP3 e RANDOM. Com estes gráficos apenas se pretende ilustrar a forma como evolui a função objectivo ao longo do processo iterativo, não sendo comparáveis as escalas dos seus eixos.



Solução inicial RANDOM Solução inicial ASAP

Figura 5.20: Comparação do comportamento da função objectivo partindo de soluções iniciais obtidas por ASAP e RANDOM.

Para os 3 primeiros exemplos, a aplicação do algoritmo *Simulated Annealing* permitiu conduzir a soluções que não excedem o limite superior estabelecido para o número de unidades funcionais, partindo de soluções iniciais construídas por ASAP ou RANDOM, onde aquele limite é geralmente ultrapassado. No entanto, no problema mais complexo (*hdll*) isso não foi conseguido, mesmo usando valores muito elevados para a penalização incluída na função objectivo, de forma a tornar praticamente impossível a aceitação de soluções que aumentem o número de unidades funcionais seleccionadas. Como foi referido, é necessário partir de uma solução inicial em que o número de unidades funcionais não exceda o máximo admitido, para garantir que a solução final satisfaça essa restrição.

Esta dependência do desempenho da aplicação implementada com a complexidade do problema a optimizar deve-se ao modo como foi construída a estrutura de vizinhança do problema, no que diz respeito à forma como pode ser alterado o conjunto de unidades funcionais incluído numa solução. Se na sequência de uma iteração for seleccionada uma unidade funcional para além do número máximo estabelecido, é atribuída a penalização considerada na função objectivo. Se essa solução for aceite, o deslocamento de outras operações para essa nova unidade funcional não acrescentam qualquer penalização adicional na função objectivo. No sentido inverso, uma unidade funcional só pode ser eliminada se forem deslocadas para outras unidades funcionais todas as operações que ela contém. Em problemas que tenham um número reduzido de operações isso é mais provável de acontecer do que quando o problema é constituído por um número elevado de operações, tal como foi verificado com o último exemplo experimentado.

O recurso a outras estruturas de vizinhança resultantes de diferentes estratégias para gerar soluções adjacentes poderá permitir contornar essa limitação. Uma estratégia possível consistiria em admitir um tipo de iteração diferente dos que foram considerados, que eliminasse uma unidade funcional do espaço de soluções, movendo para as outras unidades seleccionadas todas as operações que lhe estavam atribuídas. No entanto, isso só é possível se as restantes unidades funcionais suportarem operadores que possam acomodar todos os tipos de operações que estavam atribuídos à unidade funcional a eliminar. Dada a generalidade admitida para as unidades funcionais, esse tipo de transformação pode não ser sempre realizável. Outra alternativa consistiria em trocar o tipo de uma unidade funcional. Do mesmo modo, isso só pode ser feito se o conjunto de operações atribuído à unidade funcional original puder ser atribuído à nova unidade seleccionada ou distribuído pelo conjunto de unidades funcionais que resultam após a troca.

5.7.8 Unidades funcionais reconfiguráveis dinamicamente

As unidades funcionais consideradas na implementação apresentada anteriormente suportam vários operadores que se consideram permanentemente disponíveis, embora o funcionamento de forma *pipelined* numa unidade funcional só seja possível para sequências de operações do mesmo tipo. No entanto, considera-se que a troca de um operador por outro não consome tempo: se numa unidade funcional é terminada a execução de uma operação no ciclo i, então pode ser iniciada na mesma unidade funcional, uma operação de um tipo diferente no ciclo $i + 1$.

A aplicação do algoritmo *Simulated Annealing* apresentada foi generalizada para contemplar unidades funcionais reconfiguráveis dinamicamente. Uma unidade deste tipo suporta também vários operadores distintos, mas considera-se que cada um resulta de uma reconfiguração parcial do circuito FPGA que a implementa, o que requer um certo período de tempo para se completar e durante o qual não pode ser utilizada para realizar operações.

Unidades funcionais deste tipo podem ser realizadas por circuitos lógicos programáveis FPGA das gerações recentes, que suportam configuração parcial dos seus recursos, tal como a nova família de FPGAs da XILINX XC6200. O requisito de reconfiguração parcial é importante, já que o tempo necessário à reconfiguração integral de um circuito com uma capacidade minimamente interessante, é demasiado longo. Com este modelo, o número de operadores diferentes que uma unidade funcional física pode implementar ao longo da execução de um programa, não é condicionado pela quantidade de recursos configuráveis que dispõe, o que acontecia com o tipo de unidade funcional considerado anteriormente.

Este modelo de unidade funcional contribui de forma apreciável para o aumento de complexidade do problema de optimização considerado. Com esta característica, é agora importante a ordem pela qual uma unidade funcional realiza um conjunto de operações de diferentes tipos. Isso é determinante no tempo gasto com as reconfigurações necessárias para implementar os diversos operadores, e reflecte-se também no tempo total necessário à realização dessas operações. A ordenação óptima do conjunto de operações atribuído a uma unidade funcional é ditada pelos tempos necessários às reconfigurações entre os diferentes tipos de operações suportados. Além disso, é naturalmente necessário satisfazer também a ordem pela qual todas as operações do problema são efectuadas, de maneira a respeitar as relações de precedência definidas no GDD.

Para contemplar este modelo de unidade funcional, é acrescentada a matriz de custos de reconfiguração referida antes. Nessa matriz representa-se o número de ciclos necessários para transformar um operador noutro. Note-se que, considerando nulos todos os seus elementos é representada uma unidade funcional do tipo assumido pela implementação

anterior, em que a troca de operadores não consome qualquer tempo.

Na figura 5.21 mostra-se a especificação das características de uma unidade funcional incluindo a matriz de custos de reconfiguração. Essa matriz tem tantas linhas e colunas quantos os tipos de operações diferentes suportados. O elemento na posição (i, j) representa o número de ciclos necessários para transformar o operador i no operador j, sendo i e j a ordem pela qual cada operador é definido nessa unidade funcional.

```
####################################
# Para cada unidade funcional:
#   nome_UF  custo_area  numero_de_operacoes
#     para cada operacao:
#     nome codigo duracao latencia numero_operandos
#       para cada operando:
#         ciclo_consumo
#     ciclo_producao_do_resultado
#   matriz de custos de reconfiguracao
####################################
# UF ALU (custo 300, 4 operacoes)
ALU 300 4
# operacao ADD:
ADD + 2 2 2
0 0
1
# operacao MUL:
MUL * 6 1 2
0 0
5
# operacao SUB:
SUB - 3 1 2
0 0
2
# operacao ADDSUB:
ADDSUB a 4 1 3
0 0 2
3
# Custos de reconfiguracao:
0 8 3 4
8 0 9 4
2 9 0 6
4 3 5 0
```

Figura 5.21: Especificação das características de uma unidade funcional incluindo a matriz de custos de reconfiguração dos operadores.

Extensões na função objectivo

Para contemplar este modelo de unidade funcional, foi necessário incluir novos parâmetros para a construção da função objectivo, destinados a caracterizar o tempo dispendido com a reconfiguração das unidades funcionais. Para isso, foi acrescentado um conjunto de novas variáveis que são também actualizadas sempre que é gerada uma nova solução.

Para cada unidade funcional é calculado o número total de ciclos em que essa unidade é reconfigurada. A soma desses valores mede o número total de ciclos que nessa solução são usados para reconfiguração do conjunto de unidades funcionais seleccionadas. Em cada ciclo, é mantida uma variável que conta o número de unidades que estão com recon-

figuração a decorrer. O valor máximo do conjunto dessas variáveis representa o número máximo de reconfigurações que decorrem em paralelo, e pode ser usado para quantificar a complexidade do sistema encarregue das reconfigurações das unidades funcionais. Finalmente, é também calculado o número de ciclos que são usados exclusivamente para reconfiguração. Estes três parâmetros são disponibilizados para a construção da função objectivo nas três variáveis seguintes:

- `totconfig`: número total de ciclos que no conjunto das unidades funcionais seleccionadas são usados para reconfiguração.

- `maxconfig`: representa o número máximo de unidades funcionais que tenham simultâneamente reconfiguração a decorrer.

- `soconfig`: conta o número de ciclos que são usados exclusivamente para reconfiguração. Este número não é contabilizado pela variável `execiclos`, pelo que a duração total do sequenciamento deve ser considerada como a soma de `soconfig` e `execiclos`.

Na figura 5.22 apresenta-se uma solução do problema exemplo considerando os períodos de reconfiguração e os valores correspondentes para estas novas medidas de qualidade da solução.

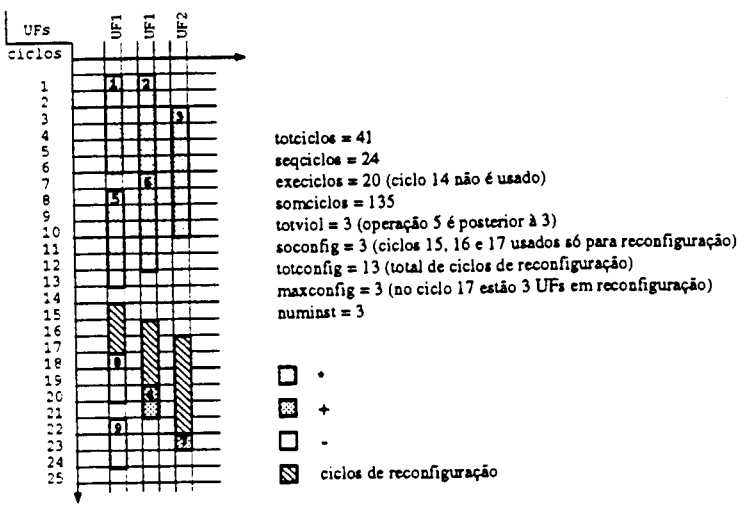


Figura 5.22: Medidas de uma solução do problema exemplo, considerando unidades funcionais reconfiguráveis dinamicamente.

Geração de soluções adjacentes

A geração de soluções adjacentes é realizada segundo um procedimento igual ao efectuado na implementação anterior. É seleccionada uma operação, definido o tipo de deslocamento H ou V e são calculados os parâmetros desse deslocamento. Com este novo modelo de unidade funcional é agora necessário considerar as alterações que resultam nos ciclos de reconfiguração das unidades funcionais envolvidas no deslocamento da operação e actualizar as variáveis que caracterizam os tempos de reconfiguração. Considera-se que as reconfigurações das unidades funcionais são efectuadas imediatamente antes de ser realizada a operação que requer essa reconfiguração.

O efeito de retirar ou colocar uma operação na lista das operações realizadas por uma unidade funcional traduz-se, na generalidade das situações, numa modificação dos períodos de reconfiguração realizados nessa unidade funcional. Isso depende do tipo de operação que é movida (inserida ou retirada) e dos tipos das operações anterior e posterior àquela que é deslocada.

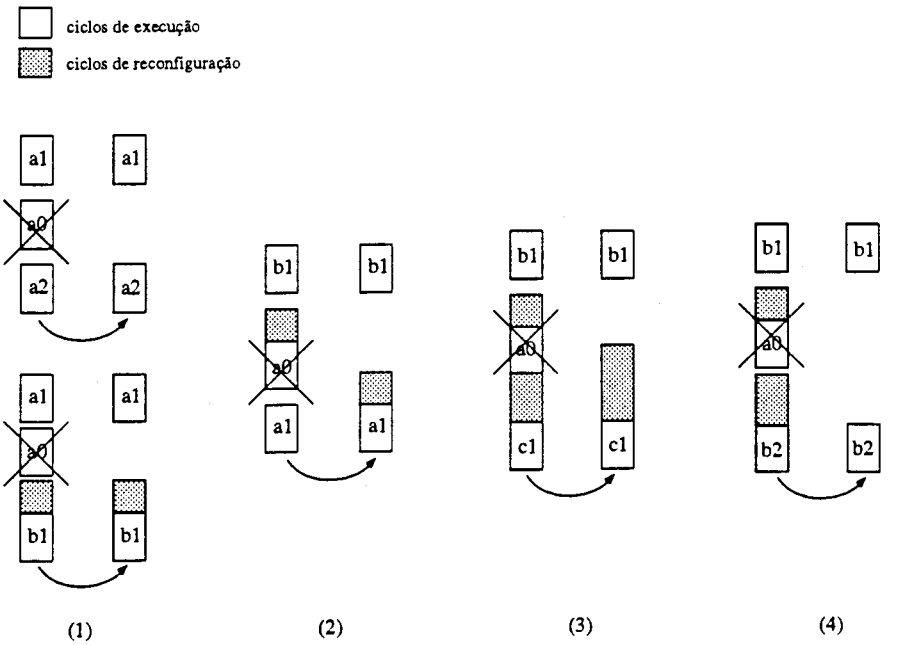


Figura 5.23: Actualização de ciclos de reconfiguração com a eliminação da operação a0. O procedimento a efectuar para inserir uma operação é o inverso do representado.

Para ilustrar os tipos de modificações que podem ocorrer nas sequências de reconfiguração, considere-se que `A`, `B` e `C` são 3 tipos de operações suportadas por diferentes operadores de uma unidade funcional, e que `ai`, `bi` e `ci` representam operações desses tipos, respectivamente. O período de reconfiguração necessário para transformar o ope-

rador que realiza a operação do tipo A no operador que suporta a operação do tipo B é representado por A->B. Admita-se que numa sequência de 3 operações daqueles tipos é retirada a operação a0 do meio. Na figura 5.23 mostram-se as 4 situações que podem ocorrer quando são consideradas sequências de operações de diferentes tipos, que se explicam em seguida.

1. a1-a0-a2 e a1-a0-b1: Neste caso não é necessária qualquer alteração dos ciclos de reconfiguração da unidade funcional. Enquanto na primeira situação não é envolvida qualquer reconfiguração, no segundo caso é mantido o período de reconfiguração A->B, associado à operação b1.

2. b1-a0-a1: Retirar a operação a0 obriga a mover o período de reconfiguração B->A da operação a0 retirada para a segunda operação a1.

3. b1-a0-c1: Retirar a operação a0 elimina o período de reconfiguração B->A que estava associado à operação a0 e substitui o período de reconfiguração A->C por B->C. Se o tempo necessário à reconfiguração B->C for superior ao tempo que fica disponível entre b1 e c1, então a solução é rejeitada.

4. b1-a0-b2: Neste caso, são apenas eliminados os períodos de reconfiguração B->A e A->B.

O processo a realizar para actualizar os ciclos de reconfiguração quando uma operação é inserida na lista de operações de uma unidade funcional é o inverso do que foi referido acima. As soluções geradas só são admissíveis se a operação deslocada couber na unidade funcional destino, juntamente com os ciclos de reconfiguração que lhe são associados. Se nenhuma das unidades funcionais já existentes puder acomodar essa operação, é seleccionada uma nova unidade funcional do mesmo tipo.

Geração da solução inicial

A solução inicial é obtida por processos semelhantes ao usado na implementação anterior. Os procedimentos realizados para construir o sequenciamento ASAP e a distribuição aleatória de operações foram modificados de forma a incluir os ciclos de reconfiguração das unidades funcionais.

Resultados

A implementação do algoritmo de *Simulated Annealing* considerando o modelo de unidade funcional com reconfiguração dinâmica de operadores foi aplicado aos dois exemplos mais

complexos experimentados com a implementação anterior. Foi usada a mesma biblioteca de unidades funcionais, mas são agora considerados os custos de reconfiguração definidos para cada unidade funcional.

Com a introdução das 3 novas variáveis que quantificam o custo associado às reconfigurações em cada solução, cresce de forma significativa o número de alternativas que é possível considerar para a função objectivo. Optou-se por manter o mesmo tipo de função objectivo considerada na implementação anterior, acrescentando a variável `soconfig` que conta o número de ciclos usados exclusivamente para reconfiguração. É de referir que, com a mesma função objectivo usada na implementação anterior, nunca foram conseguidas soluções finais admissíveis por ser sempre excedido o número máximo de unidades funcionais e atingido um número elevado de violações das restrições de precedência. Foram por isso experimentadas outras combinações de pesos daquelas variáveis, que conduziram aos valores apresentados na tabela 5.7.8. Como foi já referido, o número de ciclos necessários à execução de uma sequência de operações é a soma do valor desta variável com o de `execiclos` razão pela qual foi atribuído o mesmo peso a estas duas variáveis.

Tabela 5.8: Pesos usados na função objectivo para a aplicação com unidades funcionais reconfiguráveis dinamicamente.

	`seqciclos`	`execiclos`	`soconfig`	`totviol`	`numinst`
Pesos	2	5	5	40	80

Com esta aplicação foram experimentados apenas os problemas *inter* e *hdll*, por apresentarem um número de diferentes operações superior ao número máximo de unidades funcionais considerado. Isto obriga a que em qualquer solução ocorram necessáriamente reconfigurações das unidades funcionais. Também neste caso, por não serem conhecidas outras soluções para estes problemas dentro das características consideradas, é comparada a solução final obtida com *Simulated Annealing* com a que se atinge realizando pesquisa local pura partindo da mesma solução inicial. Nas tabelas 5.9 e 5.10 resumem-se os resultados obtidos para os dois exemplos experimentados, incluindo na coluna `config` o número total de ciclos envolvidos nas reconfigurações das unidades funcionais. Os valores apresentados na coluna `ciclos` correspondem à soma das variáveis `execiclos` e `soconfig` e representam o número de ciclos necessários ao sequenciamento das operações, incluindo os períodos de reconfiguração.

De forma semelhante ao que se verificou na implementação anterior, com as melhores soluções para o problema *inter* foram obtidas com *Simulated Annealing*, partindo das

Tabela 5.9: Resultados obtidos para o exemplo *inter* com unidades funcionais reconfiguráveis dinamicamente.

Solução inicial					parâmetros S.A.			Solução final				
Método	ciclos	config	UFs	viol	T_0	α	Iter/T	ciclos	config	UFs	viol	tempo (s)
ASAP	44	15	6	0	300	0.95	400	38	7	3	8	214
ASAP3	51	15	3	0	300	0.95	400	46	7	3	0	230
RANDOM	46	6	5	333	300	0.95	400	47	3	3	0	232
ASAP	44	15	6	0	$T_0 \rightarrow 0$ (pesquisa local)			46	9	4	1	480
RANDOM	46	6	5	333	$T_0 \rightarrow 0$ (pesquisa local)			45	3	4	0	490

soluções iniciais ASAP3 e RANDOM e com pesquisa local pura não foram conseguidas soluções finais admissíveis.

Tabela 5.10: Resultados obtidos para o exemplo *hd1l* com unidades funcionais reconfiguráveis dinamicamente.

Solução inicial					parâmetros S.A.			Solução final				
Método	ciclos	config	UFs	viol	T_0	α	Iter/T	ciclos	config	UFs	viol	tempo (s)
ASAP	394	221	7	0	300	0.95	1000	325	213	7	5	520
ASAP3	773	319	3	0	300	0.95	1000	745	298	3	0	590
RANDOM	426	234	6	32K	300	0.95	1000	520	336	5	7K	534
ASAP3	773	319	3	0	$T_0 \rightarrow 0$ (pesquisa local)			769	300	3	0	820
RANDOM	426	234	6	32K	$T_0 \rightarrow 0$ (pesquisa local)			438	292	6	13K	815

Para o problema *hd1l* e pelas razões que já foram apontadas antes, usou-se na função objectivo o valor de 5000 para penalizar cada unidade funcional que seja seleccionada para além do número máximo permitido. Tal como aconteceu com a implementação anterior, para este problema só foram atingidas soluções finais admissíveis partindo da solução inicial ASAP3. No entanto, também neste caso o conjunto de unidades funcionais na solução final é diferente do que é especificado para a solução inicial, o que significa que ocorrem trocas das unidades funcionais seleccionadas no decorrer do processo iterativo. As soluções atingidas partindo de soluções iniciais ASAP ou RANDOM ficam sempre muito distantes da admissibilidade, por nunca ser atingido o número desejado de 3 unidades funcionais. É de notar que partindo da solução inicial obtida por ASAP3 o resultado obtido com pesquisa local é pior do que o conseguido com *Simulated Annealing*.

Como considerações finais sobre os resultados obtidos com esta implementação, pode concluir-se que a aplicação desenvolvida mostrou conduzir a resultados razoáveis para o conjunto de problemas que foram ensaiados. Com o maior exemplo experimentado só foi possível encontrar uma solução final admissível definindo um conjunto inicial de unidades

funcionais. No entanto, pode verificar-se que a selecção de unidades funcionais na solução final era diferente da especificada inicialmente. Tal como foi referido antes, o facto de no problema mais complexo não ter sido possível reduzir o número de unidades funcionais até ao limite máximo estabelecido, resulta da estrutura de vizinhança considerada não ser a mais adequada. No entanto, pelas razões que foram já referidas, as características genéricas que se consideram para as unidades funcionais não facilita a adopção de outras estratégias mais flexíveis para gerar soluções adjacentes, que favoreçam a eliminação de unidades funcionais no decorrer do processo iterativo.

5.8 Conclusões

Neste capítulo apresentou-se o desenvolvimento de uma aplicação baseada no algoritmo *Simulated Annealing*, para um dos problemas centrais de optimização no processo de configuração da arquitectura reconfigurável proposta no capítulo anterior. A consideração de unidades funcionais que são configuradas para constituir um conjunto de operadores adequado às necessidades de cada aplicação, aproxima o processo de configuração deste processador para uma aplicação específica, das tarefas centrais de optimização empregues em síntese de alto nível de circuitos digitais. O problema global de optimização que consiste em seleccionar um conjunto de unidades funcionais, sequenciar as operações do problema e atribuí-las aos operadores realizados nas unidades funcionais foi aqui abordado numa perspectiva de optimização conjunta, o que é fundamental dada a elevada interdependência existente entre os 3 subproblemas que o compõem, resultante da consideração de modelos muito genéricos para as unidades funcionais.

Apesar desta técnica de optimização ter já sido explorada por outros autores no contexto de síntese de alto nível, a aplicação desenvolvida apresenta novas contribuições resultantes das características consideradas para as unidades funcionais reconfiguráveis. O âmbito de aplicação do sistema desenvolvido não se restringe apenas à arquitectura que se considerou como plataforma alvo. Com efeito, situações onde seja requerida a optimização conjunta daqueles 3 problemas, dentro de assunções que possam ser contempladas pelo modelo considerado para as unidades funcionais, podem beneficiar da aplicação desenvolvida. Exemplos são o processador reconfigurável Spyder [Ise96] construído igualmente em torno de unidades funcionais reconfiguráveis implementadas como circuitos FPGA e o processador DISC [WH96a] que adapta dinamicamente o conjunto de operadores necessários a uma aplicação, explorando a reconfiguração parcial de FPGAs. Tanto quanto é conhecido, para nenhum destes sistemas ou de outros com características similares foi abordado o problema que aqui se tratou, como parte integrante do processo de

optimização de configurações para essas arquitecturas. Embora os exemplos experimentados contenham apenas operações aritméticas convencionais, a flexibilidade oferecida pela aplicação possibilita igualmente o seu uso com problemas que incluam operações representando tarefas com maior complexidade, que beneficiem da sua realização como operadores construídos em *hardware* dedicado. A aplicação desenvolvida pode também ser facilmente generalizada para contemplar operadores com características diferentes, por exemplo envolvendo um número de operandos ou de resultados diferente daquele que é considerado pela implementação actual

A implementação realizada mostrou atingir bons resultados nos 4 exemplos seleccionados, e permitiu estabelecer certos critérios para a determinação dos parâmetros de controlo do algoritmo de optimização. No entanto, a forma como são geradas soluções adjacentes, definindo a estrutura de vizinhança do problema, não demonstrou ser a ideal para o tipo de problema a optimizar, no que se refere à forma como podem ser eliminadas unidades funcionais do conjunto seleccionado para a solução. No entanto, pela sua generalidade e abertura constitui uma boa base de trabalho para explorar em trabalhos futuros outras estratégias para construir diferentes estruturas de vizinhança, permitindo também incluir no processo de optimização restrições que modelem características variadas das unidades funcionais, para além daquelas que foram consideradas neste trabalho.

O problema aqui tratado representa uma tarefa fundamental no processo de optimização de uma configuração para o processador reconfigurável proposto, do qual depende em grande parte a rapidez conseguida na execução de um programa no processador reconfigurável proposto. Um problema que não foi estudado e que poderá ser alvo de futuros desenvolvimentos consiste na identificação de grupos de operações realizadas pelas unidades funcionais que possam ser realizadas como combinações dos operadores permitidas pela rede de interligações flexível entre as unidades funcionais.

Capítulo 6

Concepção e implementação de um processador dedicado

6.1 Introdução

Neste capítulo apresenta-se o desenvolvimento e a realização de um processador dedicado para uma aplicação de processamento digital de sinal, que conduziu à construção de um novo sistema reconfigurável, concebido tendo em vista a implementação de unidades de processamento auxiliar orientadas para processamento vectorial.

O processador dedicado que aqui se apresenta, ProHos, foi desenvolvido para realizar uma tarefa computacionalmente pesada numa aplicação específica de processamento digital de sinal. Foi desenhado como um circuito lógico inteiramente dedicado para essa aplicação, não sendo por isso programável como acontece com os processadores THD e PUFR3 apresentados nos capítulos anteriores. A arquitectura criada resultou da análise e transformação para *hardware* do problema estudado, conduzindo a um processador vectorial que explora de forma eficiente as características exibidas por essa aplicação. O ProHos destina-se a trabalhar como processador auxiliar de um computador de uso geral, para acelerar aplicações que usem Estatísticas de Ordem Superior (EOS), executando a parte computacionalmente crítica que é a estimação de momentos de ordem superior. Com vista à implementação da arquitectura desenvolvida, foi desenhado e construído com base nos seus requisitos, um sistema digital reconfigurável em torno de circuitos programáveis FPGA, destinado a computadores pessoais do tipo PC (CVR—Coprocessador Vectorial Reconfigurável). O trabalho que aqui se apresenta está em curso no âmbito de um projecto de investigação por contrato (HAREOS, *Hardware* Reconfigurável para Estatísticas de Ordem Superior), financiado pela Junta Nacional de Investigação Científica

e Tecnológica (JNICT), contrato PBIC/TIT/2489/95.

Neste capítulo apresenta-se o trabalho realizado para a concepção e desenvolvimento de raiz do ProHos e do CVR, bem como a implementação da arquitectura vectorial desenvolvida naquela plataforma reconfigurável. No sentido de enquadrar o processador dedicado na aplicação para a qual foi desenvolvido, é também apresentado o problema da estimação de estatísticas de ordem superior e uma revisão sucinta de trabalhos de outros autores visando a construção de arquitecturas de processamento dedicadas para esta aplicação.

6.2 Aplicação - Estimação de Estatísticas de Ordem Superior

Estatísticas de Ordem Superior—EOS—é a designação geralmente empregue para identificar os momentos, cumulantes e espectros de ordem superior a 2. Estas estatísticas são ferramentas poderosas na análise e classificação de sinais e a sua utilização é necessária quando estes exibem não linearidades ou se afastam da gaussianidade. Nessas situações, falham as técnicas de análise baseadas nas estatísticas de ordem 2 (função de auto-correlação e espectro de potência) [NM93].

Um exemplo clássico que justifica a utilização desta ferramenta é na identificação de sistemas que difiram entre si apenas na fase da sua resposta. Como as estatísticas de segunda ordem são "cegas na fase", é impossível distinguir sistemas com esta característica. No entanto, métodos baseados em estatísticas de ordem superior permitem contornar essa limitação.

Como exemplo, considerem-se dois sistemas com resposta impulsional finita de segunda ordem [NP93], descritos pelas seguintes equações às diferenças:

$$y_I(k) = x(k) - a \cdot x(k-1) + b \cdot x(k-2) \tag{6.1}$$

$$y_{II}(k) = x(k) - a \cdot x(k+1) + b \cdot x(k+2) \tag{6.2}$$

Métodos de identificação de sistemas baseados na função de autocorrelação (cumulante de ordem 2) da resposta a ruído branco não gaussiano com média nula, não permitem distingui-los porque essa função é idêntica para ambos e igual a:

$$C_2^{y_I}(\tau_1) = C_2^{y_{II}}(\tau_1) = \begin{bmatrix} 1 + a^2 + b^2 & -a(1+b) & b \end{bmatrix} \tag{6.3}$$

No entanto os cumulantes de ordem 3, representados por (6.4) e (6.5), já são diferentes, permitindo a sua identificação.

$$C_3^{yI}(\tau_1,\tau_2) = \begin{bmatrix} 1-a^3+b^3 & -a+ba^2 & b \\ -a+ba^2 & a^2-ab^2 & -ab \\ b & -ab & b^2 \end{bmatrix} \tag{6.4}$$

$$C_3^{yII}(\tau_1,\tau_2) = \begin{bmatrix} 1-a^3+b^3 & a^2-ab^2 & b^2 \\ a-ab^2 & -a+ba^2 & -ab \\ b^2 & -ab & b \end{bmatrix} \tag{6.5}$$

Esta ferramenta de análise tem sido aplicada a vários campos de Engenharia, em aplicações de reconhecimento de padrões [TG92], processamento de som e imagem [AG95] e identificação de sistemas. Devido à elevada complexidade computacional dos cálculos envolvidos na estimação de EOS, a sua utilização em sistemas que requeiram processamento em tempo real obriga a recorrer a arquitecturas de cálculo dedicadas, que explorem o paralelismo e regularidade das operações exibidos pelos algoritmos de estimação.

Para um processo estacionário, as estatísticas de ordem superior de interesse são os poli-espectros, cumulantes e momentos de ordem superior a 2. Por razões práticas relacionadas com a complexidade da sua estimação, não são geralmente utilizadas estatísticas de ordem superior a 4.

Os poli-espectros (bi-espectro e tri-espectro) são as transformadas de Fourier dos cumulantes de ordem 3 e 4, respectivamente. Para sinais gaussianos com média conhecida, a informação contida na função de autocorrelação é suficiente para os caracterizar. No entanto, na presença de não linearidades ou desvios da distribuição gaussiana, é necessário recorrer ao bi-espectro ou tri-espectro para permitir a sua caracterização.

Para um processo estacionário unidimensional de média nula $\{X(k)\}$, os cumulantes de ordem 2, 3 e 4 são definidos como:

$$C_2^y(\tau_1) = m_2(\tau_1) \tag{6.6}$$

$$C_3^y(\tau_1,\tau_2) = m_3(\tau_1,\tau_2) \tag{6.7}$$

$$\begin{aligned} C_4^y(\tau_1,\tau_2,\tau_3) &= m_4(\tau_1,\tau_2,\tau_3) - m_2(\tau_1)\cdot m_2(\tau_2-\tau_3) - \\ & m_2(\tau_2)\cdot m_2(\tau_3-\tau_1) - m_2(\tau_3)\cdot m_2(\tau_1-\tau_2) \end{aligned} \tag{6.8}$$

onde $m_2(\tau_1)$, $m_3(\tau_1, \tau_2)$ e $m_4(\tau_1, \tau_2, \tau_3)$ são os momentos de ordem 2, 3 e 4, respectivamente, definidos por:

$$m_2(\tau_1) = E[x(k) \cdot x(k + \tau_1)] \tag{6.9}$$

$$m_3(\tau_1, \tau_2) = E[x(k) \cdot x(k + \tau_1) \cdot x(k + \tau_2)] \tag{6.10}$$

$$m_4(\tau_1, \tau_2, \tau_3) = E[x(k) \cdot x(k + \tau_1) \cdot x(k + \tau_2) \cdot x(k + \tau_3)] \tag{6.11}$$

Nas condições acima referidas para o processo $\{X(k)\}$, a tarefa computacionalmente mais pesada para o cálculo dos poli-espectros e dos cumulantes consiste na estimação dos momentos de ordem superior. Por este motivo, os esforços desenvolvidos para acelerar os procedimentos de estimação de EOS têm sido concentrados na optimização dos algoritmos para estimar os momentos de ordem superior.

6.2.1 Estimação de EOS

A estimação dos momentos de ordem superior requer um número elevado de operações aritméticas, nomeadamente multiplicações e adições, realizadas sobre vectores de dados. No entanto, certas propriedades que essas funções exibem permitem simplificar de forma significativa o número de elementos a calcular, reduzindo com isso o número de operações aritméticas necessárias.

Propriedades de simetria

A exploração de propriedades de simetria que os momentos de ordem superior apresentam [PA95] permite reduzir de forma importante o esforço de cálculo necessário à sua estimação. Em primeiro lugar, para um processo estacionário de média nula $\{X(k)\}$, apenas interessa calcular os componentes dos momentos para as regiões positivas do seu domínio ($\tau_1, \tau_2, \tau_3 \geq 0$). Para além disso, pelas equações (6.10) e (6.11) verificam-se as igualdades seguintes para os momentos de ordem 3 e 4:

$$m_3(\tau_1, \tau_2) = m_3(\tau_2, \tau_1) \tag{6.12}$$

$$\begin{aligned} m_4(\tau_1, \tau_2, \tau_3) &= m_4(\tau_1, \tau_3, \tau_2) = m_4(\tau_2, \tau_1, \tau_3) \\ &= m_4(\tau_2, \tau_3, \tau_1) = m_4(\tau_3, \tau_1, \tau_2) \\ &= m_4(\tau_3, \tau_2, \tau_1) \end{aligned} \tag{6.13}$$

O domínio não redundante dos momentos de ordem 2, 3 e 4 fica assim restringido às regiões tais que:

$$\tau_3 \leq \tau_2 \leq \tau_1 \wedge \tau_1, \tau_2, \tau_3 \geq 0 \tag{6.14}$$

Todas as outras partes dos seus domínios podem ser obtidas por reflexões daquelas regiões. Estas simplificações conduzem a uma redução do número de componentes a calcular para cada momento que é igual a 2, 8 e 32 vezes, respectivamente para $m_2(\tau_1)$, $m_3(\tau_1, \tau_2)$ e $m_4(\tau_1, \tau_2, \tau_3)$.

Para além das simplificações devidas às propriedades de simetria, aplicações práticas necessitam apenas de um número de pontos de cada função (usualmente designados por *taps*) que é limitado por uma função da dimensão do segmento de dados L. Esses parâmetros são normalmente designados por $P2$, $P3$ e $P4$, respectivamente para os momentos de ordem 2, 3 e 4 [PA95, Kay88]:

$$P2 = \frac{L}{5}, P3 = \sqrt{\frac{L}{5}}, P4 = \sqrt[3]{\frac{L}{5}} \tag{6.15}$$

Assim, as denominadas regiões de interesse para os momentos de ordem superior são delimitadas por:

$$0 \leq \tau_1 \leq P_2 \tag{6.16}$$

$$0 \leq \tau_1 \leq P_3 \wedge 0 \leq \tau_2 \leq \tau_1 \tag{6.17}$$

$$0 \leq \tau_1 \leq P_4 \wedge 0 \leq \tau_3 \leq \tau_2 \leq \tau_1 \tag{6.18}$$

Procedimento de estimação

Os momentos de ordem superior de um processo estocástico $\{X(k)\}$ podem ser estimados, a partir de uma amostra de comprimento N, com o seguinte algoritmo de estimação [NM93]:

1. Segmentar o vector de entrada $x(k)$ em M segmentos $x_{m0}(k)$ com L elementos cada:

$$x_{m0}(l) = x(l + m \times L),\ m = 0 \ldots M-1,\ l = 0 \ldots L-1 \tag{6.19}$$

2. Determinar o valor médio Me_m de cada segmento $x_{m0}(k)$, e subtraí-lo aos elementos de cada segmento, obtendo os segmentos normalizados $x_m(l)$:

$$\begin{aligned} Me_m &= \frac{1}{L}\sum_{l=0}^{L-1} x_{m0}(l),\ m = 0 \ldots M-1 \\ x_m(l) &= x_{m0}(l) - Me_m,\ m = 0 \ldots M-1,\ l = 0 \ldots L-1 \end{aligned} \tag{6.20}$$

3. Estimar os momentos de ordem 2, 3 e 4 para cada segmento $x_m(l)$, nas regiões de interesse definidas pelas equações (6.16), (6.17) e (6.18):

$$m_2^{x_m}(\tau_1) = \frac{1}{L}\sum_{l=0}^{L-1-\tau_1} x_m(l) \cdot x_m(l+\tau_1) \tag{6.21}$$

$$m_3^{x_m}(\tau_1, \tau_2) = \frac{1}{L}\sum_{l=0}^{L-1-\tau_1} x_m(l) \cdot x_m(l+\tau_1) \cdot x_m(l+\tau_2) \tag{6.22}$$

$$m_4^{x_m}(\tau_1, \tau_2, \tau_3) = \frac{1}{L}\sum_{l=0}^{L-1-\tau_1} x_m(l) \cdot x_m(l+\tau_1) \cdot x_m(l+\tau_2) \cdot x_m(l+\tau_3) \tag{6.23}$$

4. Estimar os momentos de todo o vector $x(k)$, calculando cada componente como a média aritmética dos componentes respectivos dos momentos de cada segmento $x_m(k)$:

$$m_2(\tau_1) = \frac{1}{M}\sum_{m=0}^{M-1} m_2^{x_m}(\tau_1) \tag{6.24}$$

$$m_3(\tau_1, \tau_2) = \frac{1}{M}\sum_{m=0}^{M-1} m_3^{x_m}(\tau_1, \tau_2) \tag{6.25}$$

$$m_4(\tau_1, \tau_2, \tau_3) = \frac{1}{M}\sum_{m=0}^{M-1} m_4^{x_m}(\tau_1, \tau_2, \tau_3) \tag{6.26}$$

A tarefa computacionalmente mais pesada deste algoritmo é o passo 3, onde são estimados os momentos de cada segmento normalizado $x_m(k)$. Considerando as regiões de interesse definidas pelas equações (6.15) a (6.18), o número de elementos a calcular para cada momento é de ordem $O(L)$. Como o número de operações necessárias para calcular cada componente de um momento também apresenta ordem de complexidade $O(L)$, a estimação de todos os componentes de cada momento tem complexidade $O(L^2)$.

No entanto, analisando as equações (6.21), (6.22) e (6.23) do algoritmo de estimação, pode ver-se que o termo somado em (6.21) é repetido em (6.22), e por sua vez este último

aparece na equação (6.23) que calcula o momento de ordem 4. Assim, guardando os valores desses produtos de forma a não os repetir no cálculo dos momentos de ordem imediatamente superior, é possível reduzir de forma significativa o número total de multiplicações efectuadas.

Implementação em *software*

A implementação computacional do passo 3 do algoritmo de estimação pode ser feita realizando separadamente as somas que calculam cada componente dos momentos de ordem superior. Nesse caso, para reduzir o número de multiplicações realizadas de acordo com o que foi dito antes, é necessário guardar em vectores todos os produtos que são produzidos durante o cálculo de um momento e que são necessários no cálculo do momento de ordem imediatamente superior.

Deste modo, a estimação dos momentos de um segmento $x_m(k)$ pode ser realizado com os passos seguintes:

$$\begin{aligned} p_2(\tau_1, k) &= x_m(k) \cdot x_m(k+\tau_1),\ k = 0 \ldots L-1-\tau_1 \\ m_2^{x_m}(\tau_1) &= \frac{1}{L} \sum_{l=0}^{L-1-\tau_1} p_2(\tau_1, l) \end{aligned} \tag{6.27}$$

$$\begin{aligned} p_3(\tau_1, \tau_2, k) &= p_2(\tau_1, k) \cdot x_m(k+\tau_2),\ k = 0 \ldots L-1-\tau_1 \\ m_3^{x_m}(\tau_1, \tau_2) &= \frac{1}{L} \sum_{l=0}^{L-1-\tau_1} p_3(\tau_1, \tau_2, l) \end{aligned} \tag{6.28}$$

$$m_4^{x_m}(\tau_1, \tau_2, \tau_3) = \frac{1}{L} \sum_{l=0}^{L-1-\tau_1} p_3(\tau_1, \tau_2, l) \cdot x_m(l+\tau_3) \tag{6.29}$$

Numa implementação computacional, é possível codificar aquela sequência de operações de forma a evitar o uso de vectores para armazenar os produtos $p_2(\tau_1, k)$ e $p_3(\tau_1, \tau_2, k)$. Na figura 6.1 mostra-se em pseudo-código o algoritmo de estimação dos momentos de ordem superior, tirando partido da re-utilização daqueles produtos [SM94, PA95]. Os vectores `m2[]`, `m3[]` e `m4[]` consideram-se iniciados com zero, e não é representada a divisão final pelo comprimento do segmento L.

6.2.2 Trabalhos relacionados

Com o objectivo de acelerar os procedimentos de estimação de EOS, outros autores propuseram soluções baseadas em arquitecturas sistólicas, construídas como agregados regulares

```
Para τ1=0 até P2
  Para i=0 até L-τ1-1
    p2 = x[i]*x[i+τ1];
    m2[τ1]=m2[τ1]+p2;
    Se τ1<=P3
      Para τ2=0 até τ1
        p3=p2*x[i+τ2];
        m3[τ1,τ2]=m3[τ1,τ2]+p3;
        Se τ1<=P4
          Para τ3=0 até τ2
            m4[τ1,τ2,τ3]=m4[τ1,τ2,τ3]+p3*x[i+τ3];
```

Figura 6.1: Algoritmo para estimação dos momentos de ordem superior do vector x[]

de unidades de processamento dedicadas a esta aplicação [SM94, SM95] e implementações de *software* em máquinas paralelas [KM95].

Em [SM94] é aplicada uma metodologia de transformação sistemática ao algoritmo de estimação, de forma a transformá-lo numa arquitectura sistólica constituída por um agregado regular de unidades idênticas de processamento. O algoritmo de estimação é primeiramente transformado numa descrição equivalente contendo apenas dependências de dados locais. A transformação é feita analisando as dependências de dados entre as operações das iterações dos ciclos que compõem o algoritmo. Essa representação é importante como ponto de partida para a construção de arquitecturas sistólicas, já que não contém dependências globais de dados entre as unidades de processamento. A transformação assim obtida é posteriormente traduzida num agregado regular de nós de processamento idênticos que comunicam entre si apenas por ligações locais entre nós adjacentes e permitem calcular em paralelo os componentes dos momentos de ordens 2, 3 e 4 (figura 6.2).

Pelas suas características de regularidade e planaridade, a estrutura assim obtida mostra ser adequada à implementação em tecnologia microelectrónica. No entanto, para estimar os momentos de um segmento com L elementos, o agregado resultante tem $L \times (L+1)/2$ nós de processamento, sendo cada um constituído por 4 somadores e 4 multiplicadores. Para segmentos com centenas ou milhares de amostras (dimensões típicas para se atingirem estimativas estatisticamente válidas), a realização física desse sistema como um circuito integrado torna-se impraticável. Por exemplo, para estimar os momentos de um segmento com 1000 elementos seriam necessários mais de 2 milhões de somadores e de multiplicadores, para além do restante sistema que seria necessário para fornecer os elementos do vector na sequência correcta e recuperar os resultados.

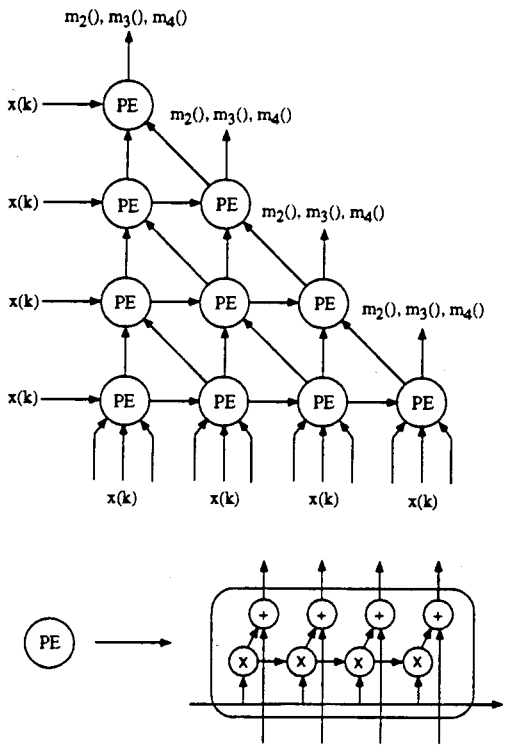


Figura 6.2: Arquitectura do agregado regular de unidades de processamento para estimação em paralelo dos momentos de ordem superior.

No seguimento do trabalho anterior, é proposto em [SM95] um algoritmo adaptativo que permite actualizar as estimativas dos momentos de ordem superior sempre que é fornecido um novo elemento da sequência a processar. O algoritmo trabalha sobre uma janela contendo as últimas M amostras recebidas e foi transformado numa arquitectura regular de unidades de processamento idênticas, seguindo a metodologia proposta em [SM94]. A arquitectura resultante apresenta características de regularidade e localidade de dados idênticas às exibidas pelo trabalho anterior, mas requer igualmente um número demasiado elevado de unidades de processamento, para ser praticável a sua realização física para segmentos de dados de dimensão razoável.

Em [KM95], a estrutura de processamento sistólico obtida com as transformações propostas em [SM94] foi traduzida para uma máquina de processamento paralelo, MasPar-1, permitindo processar vectores de dados com dimensões até 1024 amostras. Esta máquina paralela é constituída por uma rede de 64×64 processadores RISC de 4 *bits*, tendo cada um 32 registos de 32 *bits* e 64 K*bytes* de memória RAM. Cada processador possui ligações directas a cada um dos seus 8 vizinhos mais próximos. Para vectores com dimensão inferior ou igual a 64, cada processador implementa um dos nós de processamento da

estrutura mostrada na figura 6.2; para vectores com dimensão superior, são atribuídas a cada processador as operações correspondentes a dois ou mais nós daquela rede, embora a dimensão máxima suportada pela implementação proposta seja limitada a 1024 amostras. A realização nesta máquina paralela conduziu a ganhos em rapidez que atingem 5 vezes, quando comparada com implementações em C executadas em estações de trabalho UNIX com processadores RISC e 64 M*bytes* de RAM (SuperSparc, MIPS e Alpha). A título comparativo, refira-se que os 4096 processadores da máquina paralela MasPar-1 contêm um total de 256 M*bytes* de memória RAM.

Com o objectivo de explorar a utilização de EOS em sistemas de codificação e compressão de vídeo digital para baixas cadências, baseados em computadores pessoais de baixo custo, foi desenvolvido o processador ProHos dedicado à estimação de momentos de ordem superior [APCRM96, APCRM97] que se apresenta na secção seguinte.

6.3 ProHos - um estimador de momentos de ordem superior

O ProHos é um processador dedicado que foi concebido para acelerar aplicações que façam uso de estatísticas de ordem superior, realizando a parte computacionalmente crítica desta tarefa: estimação dos momentos de ordem superior. O primeiro objectivo deste desenvolvimento é o estudo da viabilidade da utilização de EOS em aplicações de processamento de sinal em tempo real, nomeadamente na compressão de vídeo digital para baixas cadências (vídeo-telefone ou sistemas de vídeo-vigilância, por exemplo). Com o objectivo de explorar convenientemente a estrutura do problema a resolver, optou-se por desenvolver este processador como uma arquitectura inteiramente dedicada à aplicação, construindo para isso circuitos lógicos dedicados que implementam as estruturas de controlo, processamento e memória requeridas especificamente para esta aplicação. Ao contrário das arquitecturas propostas nos trabalhos referidos antes, o ProHos apresenta complexidade reduzida, tornando praticável a sua implementação em plataformas baseados em circuitos FPGA. São também planeadas implementações futuras desta arquitectura como um circuito integrado de aplicação específica, o que permitirá naturalmente atingir níveis superiores de desempenho e de redução de espaço, se comparados com os conseguidos por implementações em torno de circuitos programáveis FPGA.

Para a realização física desta arquitectura, foi desenvolvido e construído o sistema reconfigurável CVR, com base em circuitos programáveis FPGA da XILINX. A arquitectura do CVR foi criada tendo em vista a implementação imediata do ProHos, embora constitua

uma plataforma flexível igualmente apta para a realização de outros processadores dedicados para diferentes aplicações. Apesar da utilização do sistema reconfigurável construído conduzir a implementações com pior desempenho do que é possível obter recorrendo a circuitos integrados de aplicação específica, torna possível explorar a reconfigurabilidade oferecida por essa plataforma. Por um lado, permite experimentar e afinar eficazmente variadas alternativas arquitecturais para os elementos constituintes do processador proposto. Por outro lado, possibilita também a construção de outras arquitecturas derivadas da que é proposta, mais bem adaptadas a características específicas de variantes do problema original (por exemplo, número diferente de *bits* dos elementos do vector ou estimar os momentos em regiões específicas dos seus domínios).

6.3.1 Vectorização do algoritmo de estimação

Da análise do algoritmo de estimação representado na figura 6.1, pode ver-se que, para além do elevado número de multiplicações e adições necessário ao cálculo dos momentos propriamente ditos, é também executado um número igualmente grande de operações para gerar a sequência de índices dos ciclos e para calcular os endereços de memória dos vectores utilizados. Embora essas operações não sejam evidenciadas quando o algoritmo é codificado numa linguagem de programação de alto nível, elas existem e consomem tempo de execução. Para além disso, também o acesso intensivo aos elementos do vector de dados e aos componentes dos momentos, normalmente residentes em memória, condicionam a rapidez que um processador de uso geral pode atingir na execução deste algoritmo. Nesta secção apresenta-se a análise do algoritmo de estimação dos momentos de ordem superior que conduziu à identificação das operações vectoriais e à arquitectura que é proposta para o ProHos.

Os componentes do momento de segunda ordem $m_2(\tau_1)$ são obtidos calculando o produto interno do segmento de dados $x_m(k)$ por ele próprio avançado de τ_1 (função de auto-correlação). Sendo L a dimensão do segmento de dados, o número de elementos usados para calcular o elemento $m_2(\tau_1)$ é igual a $L-\tau_1$. De forma semelhante, o componente $m_3(\tau_1,\tau_2)$ do momento de ordem 3 , é obtido calculando o produto interno entre $x_m(k)$, $x_m(k+\tau_1)$ e $x_m(k+\tau_2)$. Como as regiões de interesse são restringidas por $\tau_1 \geq \tau_2$, também neste caso o número de elementos usados de cada um dos três vectores é igual a $L-\tau_1$. De igual modo, os componentes do momento de ordem 4 são obtidos à custa do produto interno de 4 cópias do segmento de dados, com desfasamentos entre eles iguais a τ_1, τ_2 e τ_3, sendo também neste caso considerados $L-\tau_1$ elementos em cada cópia desse vector.

Na figura 6.3 apresenta-se um exemplo que mostra os conjuntos de dados necessários

para o cálculo dos componentes do momento de ordem 3 de um vector com 8 elementos, na região de interesse definida por $P3 = 3$. Pode ver-se que os $L - \tau_1$ elementos usados para calcular cada componente de $m_3(\tau_1, \tau_2)$ são os que estão no interior da região delimitada pelo traço a carregado. As operações realizadas para cada conjunto de 3 elementos são sempre iguais e consistem em acumular o resultado do produto entre os elementos em posições correspondentes das partes de cada segmento incluídas nessas regiões.

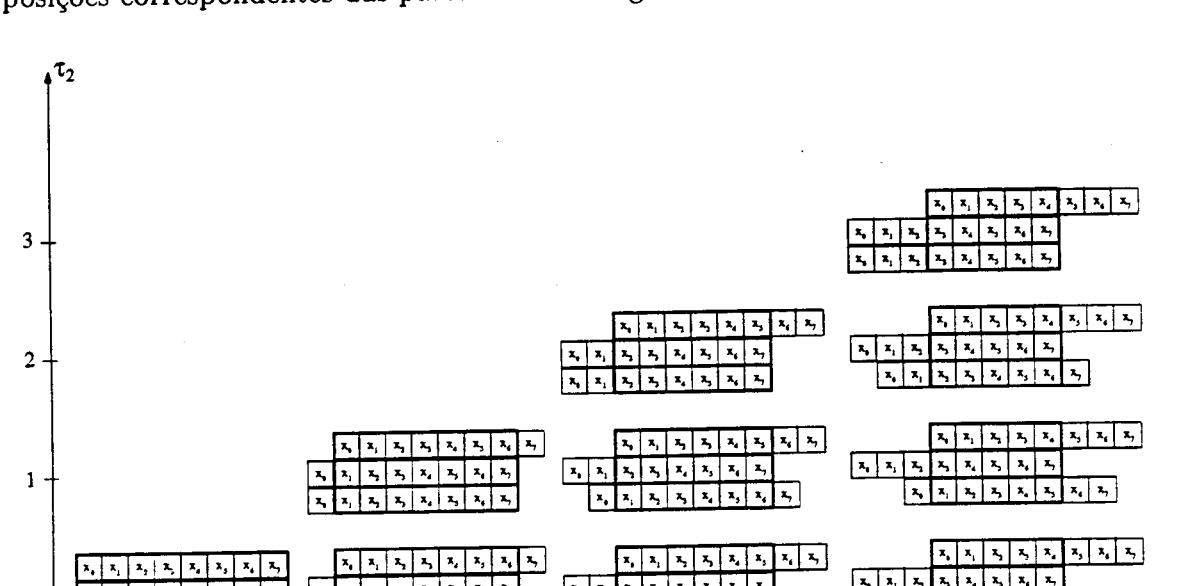


Figura 6.3: Conjunto de dados usados na estimação de cada componente do momento de ordem 3, para um vector de 8 elementos e $P3 = 3$.

Justapondo aquelas 10 regiões, obtêm-se 3 vectores com 60 elementos cada, que permitem calcular todos os componentes do momento de ordem 3 repetindo uma sequência de operações iguais e que consistem em duas multiplicações e uma adição, para cada grupo de 3 elementos desses vectores (figura 6.4). Para obter todos os elementos do momento de ordem 3 é apenas necessário colocar com zero a variável onde são acumulados os produtos, que correspondem aos traços a carregado no desenho da figura 6.4. Nesta sequência de operações, bem como na análise que se segue, não é considerada a realização da divisão final pelo comprimento do segmento de dados.

Uma unidade para calcular os componentes dos momentos de ordem 3 pode ser construída como um circuito síncrono com dois multiplicadores e um acumulador (somador com registo acumulador), da forma que se mostra na figura 6.5. Para este circuito, considera-se que, em cada ciclo de relógio, é fornecido um grupo de 3 elementos e que

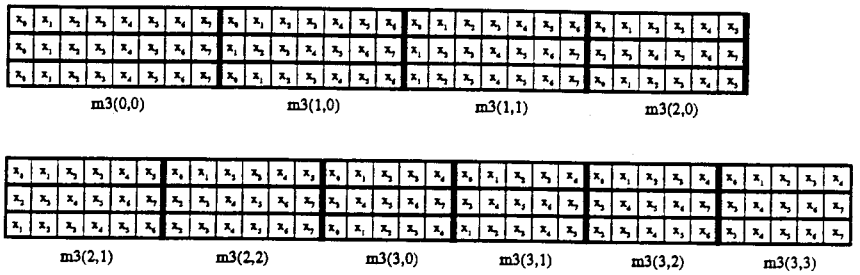


Figura 6.4: Composição dos 3 vectores de dados para estimação do momento de ordem 3.

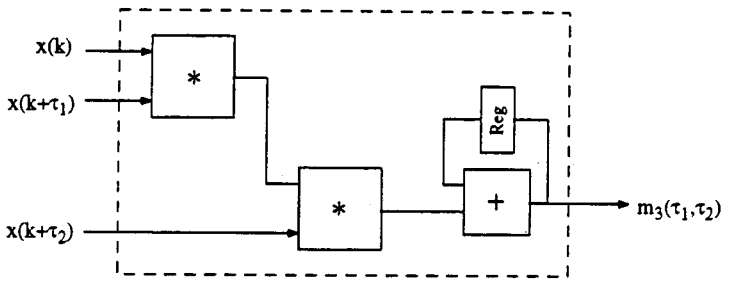


Figura 6.5: Unidade para calcular os componentes do momento de ordem 3.

o registo acumulador é carregado com o resultado presente na saída do somador sincronizado com o mesmo relógio. Fornecendo nas entradas desta unidade os elementos do vector de dados na sequência representada pela figura 6.4, e carregando com zero o registo acumulador em ciclos de relógio determinados, obtêm-se na saída do acumulador os componentes do momento de ordem 3. O registo acumulador deve ser colocado em zero nos mesmos ciclos de relógio em que os valores na saída do somador representam os componentes pretendidos, correspondendo às fronteiras dos blocos de dados identificadas com traço carregado no desenho da figura 6.4.

Analisando a figura 6.3, pode ver-se que na sequência de operações que são realizadas para calcular os componentes do momento de ordem 3, são também efectuados os produtos necessários para obter os componentes do momento de ordem 2. Se na unidade de cálculo mostrada na figura 6.5 for acrescentado um acumulador na saída do primeiro multiplicador, como se mostra na figura 6.6, então os valores calculados por esse acumulador correspondem aos componentes do momentos de segunda ordem. É de notar que durante o cálculo dos 10 componentes do momento de ordem 3 para o exemplo apresentado, aquela unidade produz também 10 elementos de $m_2(\tau_1)$, embora apenas 4 sejam diferentes.

Expandindo o operador mostrado na figura 6.6, acrescentando um multiplicador e um acumulador, obtém-se uma unidade com 3 multiplicadores e 3 acumuladores que calcula em paralelo todos os componentes dos momentos de ordem 4, 3 e 2. Com um processo

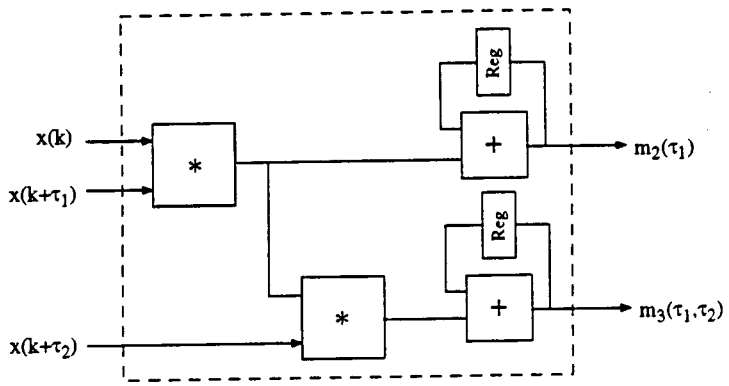


Figura 6.6: Unidade de cálculo para calcular em paralelo os componentes do momento de ordem 3 e 2.

semelhante ao que foi exemplificado para o momento de ordem 3, podem ser construídos os 4 vectores com a sequência correcta de elementos a fornecer às 4 entradas dessa unidade de cálculo desse tipo, produzindo em paralelo todos os componentes dos momentos de ordem superior.

O cálculo dos momentos de ordem superior com o processamento de longos vectores de dados permite tirar partido do aumento de rapidez que é possível conseguir usando uma unidade de cálculo *pipelined* com vários estágios de processamento. Com efeito, o atraso inicial que resulta da propagação pelos estágios de *pipeline* terá pouca influência no tempo total de processamento do vector de dados, uma vez que o número de elementos a processar será certamente muito superior ao número de estágios de *pipeline* que tal unidade possa ter. Após ultrapassado esse atraso inicial, em cada ciclo de relógio essa unidade recebe 4 operandos e produz o resultado de 3 multiplicações e 3 adições.

O processador vectorial desenvolvido para esta aplicação e que se apresenta detalhadamente na secção seguinte recorre a uma unidade de cálculo com as características referidas acima. Para além disso, é incluído um conjunto de memórias rápidas actuando como *cache* e circuitos dedicados para geração dos endereços dessas memórias, que produzem as sequências apropriadas de endereços para colocar nas entradas da unidade de cálculo os elementos do vector de dados na ordem apropriada.

6.3.2 Arquitectura do ProHos

O ProHos foi concebido para ser usado como processador auxiliar de um computador de uso geral e é constituído por 6 blocos funcionais principais: interface com o computador principal, memória principal, geradores de endereços, memórias *cache*, unidade de cálculo e unidade de controlo (figura 6.7).

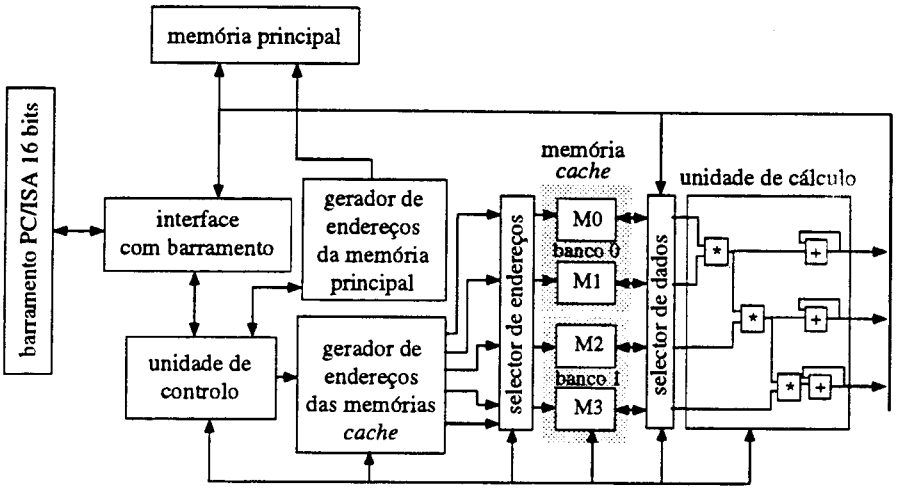


Figura 6.7: Arquitectura do ProHos.

O computador principal coloca o vector de dados a processar na memória principal, inicia a operação do ProHos e recupera os resultados quando é sinalizado o fim de processamento. Na implementação actual, o ProHos calcula os momentos de uma sequência de segmentos com dimensão L igual a potências inteiras de dois. Deste modo, a divisão pelo comprimento do segmento reduz-se a um deslocamento de *bits*, bastando para isso desprezar os $\log_2(L)$ *bits* menos significativos dos resultados obtidos à saída dos acumuladores da unidade de cálculo.

O ProHos processa um segmento de dados de cada vez; o segmento a processar é carregado para as memórias *cache*, de onde são enviados os seus elementos na sequência apropriada para as entradas da unidade de cálculo. Enquanto um dos bancos de memória *cache* fornece os dados à unidade de cálculo, o outro é carregado com o próximo segmento a processar, obtido da memória principal. Desta f ·na é possível manter um fluxo de dados constante de 4 elementos por ciclo de relógio nas entradas da unidade de cálculo, durante o processamento de todo o vector de dados. Para possibilitar a utilização de apenas 2 memórias físicas em cada um desses bancos e fornecer os 4 dados à unidade de cálculo, foram incluídos circuitos selectores (*multiplexers*) nos barramentos de endereços e de dados dessas memórias, o que permite ler duas vezes cada uma em cada ciclo de relógio.

As sequências de endereços necessárias para leitura e escrita da memória principal e das memórias *cache* são obtidas com circuitos que foram construídos em torno de contadores binários e comparadores. Considerando apenas a sequência de endereços para leitura das memórias *cache*, as operações realizadas em cada ciclo de relógio equivalem a 4 adições e uma comparação de inteiros.

A unidade de cálculo foi obtida como a extensão da que foi apresentada na figura 6.6 para os momentos de ordem 2 e 3, de forma a calcular em paralelo os componentes dos momentos de segunda, terceira e quarta ordem. É composta por três multiplicadores ligados em cascata e três acumuladores, efectuando desse modo 6 operações aritméticas em cada ciclo de relógio.

Finalmente, o controlo do processador é assegurado por duas máquinas de estados. O controlador principal comanda o funcionamento global do ProHos, o acesso à memória principal e o controlo da máquina de estados secundária. Esta é a responsável por gerar os sinais de comando para a unidade de cálculo e para o circuito gerador de endereços das memórias *cache*.

6.3.3 Unidade de controlo e interface com o computador principal

O ProHos comunica com o computador principal através do barramento de expansão de uso geral. A implementação no sistema CVR apresentado mais à frente neste capítulo, é destinada a computadores pessoais tipo PC e utiliza o barramento ISA de 16 *bits*, realizando transferências de dados com instruções de leitura e escrita em portas de entrada e saída. Os dados a processar e os resultados produzidos são transferidos de e para a memória principal do ProHos com operações de leitura e escrita de blocos de dados, realizadas por aplicações executadas pelo processador principal. O controlo do funcionamento do processador é também realizado com operações de escrita e leitura em portas de entrada e saída apropriadas.

O controlo do ProHos é realizado por duas máquinas de estados organizadas hierarquicamente: controlador principal e controlador secundário. O controlador secundário produz os sinais necessários para o gerador de endereços de leitura das memórias *cache* e para a unidade de cálculo. A sua operação é iniciada pelo controlador principal e completa o processamento de todos os segmentos de dados, sem requerer qualquer intervenção adicional do controlador principal.

O controlador principal é responsável pelo funcionamento global do ProHos: carregar os segmentos de dados da memória principal para as memórias *cache*, trocar a função atribuída aos bancos de memória *cache* quando é completado o processamento de um segmento, guardar os resultados produzidos pela unidade de cálculo na memória principal e notificar o processador principal quando é concluído o processamento. A funcionalidade implementada por este controlador é representada pelo fluxograma da figura 6.8.

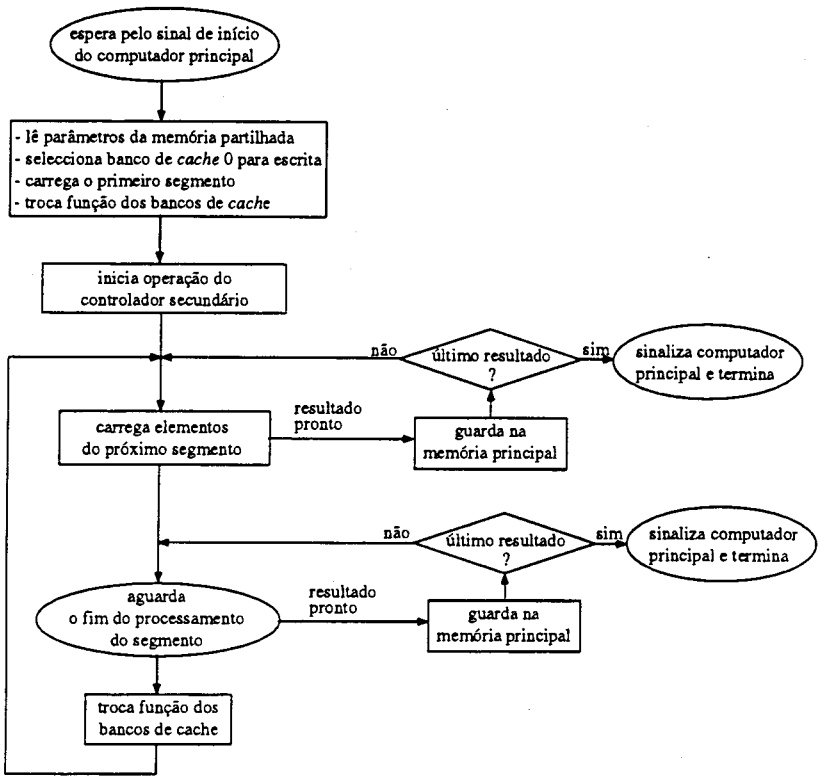


Figura 6.8: Fluxograma representando a funcionalidade implementada pelo controlador principal do ProHos.

6.3.4 Unidade de cálculo

A unidade de cálculo é composta por três multiplicadores ligados em cascata com um acumulador na saída de cada um, como se mostra na figura 6.9. Deste modo, são acumulados em paralelo os produtos $x(i) \times x(i+\tau_1)$, $x(i) \times x(i+\tau_1) \times x(i+\tau_2)$ e $x(i) \times x(i+\tau_1) \times x(i+\tau_2) \times x(i+\tau_3)$, realizando 3 multiplicações e 3 adições em cada ciclo de relógio. Na implementação presente são processados dados inteiros de 8 *bits*, o que é suficiente para aplicações de processamento vídeo digital. As operações realizadas pela unidade de cálculo são efectuadas sobre dados inteiros com sinal, representados em complemento para dois. Considera-se também que os multiplicadores e acumuladores trabalham sobre operandos com número suficiente de *bits* para garantir que não é excedida a gama de representação de inteiros durante os cálculos realizados.

O primeiro multiplicador[1] recebe dois operandos de 8 *bits* e produz um resultado de 16 *bits*. O segundo multiplica o resultado do primeiro (16 *bits*) com um elemento de 8 *bits*, produzindo assim um resultado de 24 *bits*; finalmente, o terceiro multiplica operandos de

[1] A ordem referida é de cima para baixo no desenho da figura 6.9

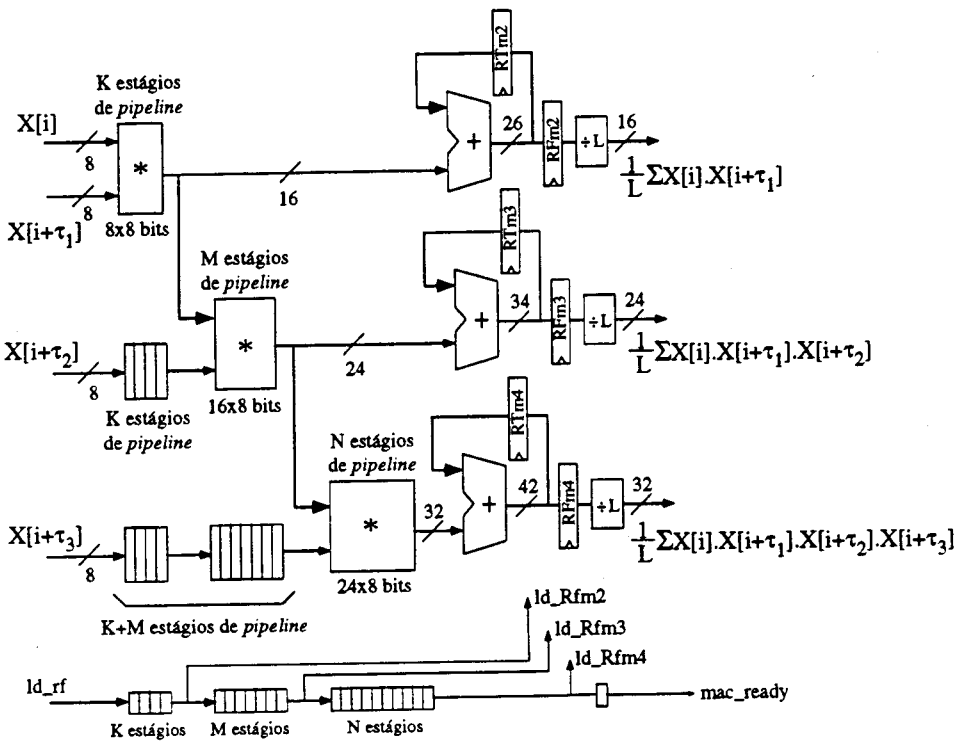


Figura 6.9: Unidade de cálculo do ProHos.

24 e 8 *bits*, produzindo o resultado com 32 *bits*.

Os 3 multiplicadores podem ser *pipelined* com um número diferente de estágios de *pipeline*. Para serem fornecidos simultaneamente os 4 dados nas entradas da unidade de cálculo, foi necessário incluir os registos de deslocamento mostrados na figura, para atrasar os elementos do segmento de dados que são fornecidas como operandos do segundo e do terceiro multiplicador. Esta solução é semelhante à que foi adoptada na unidade de cálculo do processador THD e serve para atrasar aqueles operandos um número adequado de ciclos de relógio, permitindo simplificar a operação das unidades de controlo e de geração de endereços das memórias *cache*.

Como as operações aritméticas são realizadas sobre dados inteiros, é necessário garantir que o número de *bits* utilizado em cada operador seja suficiente para acomodar os resultados que possam ser produzidos. Os multiplicadores com as características referidas acima garantem que isso é sempre verificado com os resultados que produzem. No entanto, como os acumuladores têm de acumular os resultados dos multiplicadores um número de vezes que, no pior dos casos, é igual ao comprimento do segmento L, o número de *bits* mínimo para os acumuladores (somador e registo acumulador) depende do comprimento do segmento a processar. Para garantir que não é excedida a gama de representação nos

resultados produzidos, o número de *bits* desses operadores deverá ser igual a $16+\lceil \log_2 L \rceil$, $24+\lceil \log_2 L \rceil$ e $32+\lceil \log_2 L \rceil$, respectivamente para o primeiro, segundo e terceiro acumuladores. Para os segmentos com 1024 elementos que são considerados na implementação actual, estes números são respectivamente 26, 34 e 42 *bits*.

Para evitar o uso de circuitos divisores, são considerados segmentos de dados cuja dimensão L seja igual a uma potência inteira de 2. Deste modo permite-se realizar a divisão por L do resultado produzido pelos acumuladores como uma operação de deslocamento de *bits*, bastando desprezar os $\log_2 L$ *bits* menos significativos dos valores calculados pelos acumuladores. De modo a não impor esta restrição ao comprimento do segmento de dados, podem ser seguidas duas abordagens. Uma solução consiste em deixar essa operação a cargo do computador principal, embora nesse caso os resultados calculados pelo processador auxiliar ocupem um número maior de *bits*, e impliquem um maior número de informação a transferir entre o ProHos e o computador principal. Outra solução consiste em usar um único circuito divisor iterativo, que opere em sequência sobre os 3 resultados produzidos pela unidade de cálculo. Como esses resultados são gerados com períodos aproximados de L ciclos de relógio, basta que esse tempo seja suficiente para efectuar essas três divisões.

Para além do sinal de relógio, inibição de relógio e do sinal de inicialização, esta unidade é controlada por uma única linha de controlo (`ld_rf`). A activação desta linha provoca o carregamento dos registos na saída de cada acumulador (`RFm2`, `RFm3` e `RFm4`), e coloca com zero cada registo acumulador (`RTm2`, `RTm3` e `RTm4`), em ciclos de relógio apropriados. A cadeia de registos representada na parte inferior da figura permite atrasar este sinal, de forma a poder ser activado no mesmo ciclo de relógio em que é colocado nas entradas da unidade de cálculo o último conjunto de 4 dados necessários para o cálculo de um componente dos momentos. Os sinais atrasados são designados na figura por `ld_Rfm2`, `ld_Rfm3` e `ld_Rfm4`, respectivamente para os registos do primeiro, segundo e terceiro acumuladores. O sinal que activa o carregamento do registo onde é colocado o momento de ordem 4 (`ld_rfm4`) é atrasado um ciclo de relógio adicional e serve para sinalizar para a unidade de controlo que está disponível um novo conjunto de resultados (sinal `mac_ready`).

6.3.5 Arquitectura de memória

A memória do ProHos é constituída pela memória principal e pelas memórias *cache*. A memória principal é usada para transferir dados com o computador principal, e as memórias *cache* funcionam como memórias vectoriais, armazenando vectores recebidos

da memória principal e enviando-os para as entradas da unidade de cálculo.

Memória principal

A memória principal é usada para transferir dados e resultados entre o ProHos e o computador principal. Os endereços e sinais de controlo da memória principal do ProHos são produzidos localmente com circuitos dedicados construídos em torno de contadores. O acesso a esta memória pelo processador principal é realizado sob controlo do ProHos.

Para manter à entrada da unidade de cálculo um fluxo de dados constante de 4 amostras por ciclo de relógio, obtidas de uma única memória, seria necessário utilizar dispositivos de memória com um ciclo de acesso pelo menos 4 vezes mais curto do que o período de relógio do sistema. Por exemplo, para uma frequência de relógio de 12.5 MHz (período igual a 80 ns), seria necessário ler dados dessa memória de 20 em 20 ns. Embora isto não seja impossível de obter com uma implementação como um circuito integrado com memórias internas, torna-se impraticável em implementações com circuitos programáveis e memórias externas, já que ao tempo de acesso da memória é necessário adicionar também os tempos de atraso introduzidos pelos blocos de entrada e saída dos circuitos programáveis e também os atrasos resultantes das ligações externas entre os circuitos integrados. Para além disso, com uma só memória esse problema seria agravado pela necessidade de ser também usada para armazenar os resultados produzidos pela unidade de cálculo.

Memórias *cache*

Foi implementado um sistema de memória *cache* que permite enviar para a unidade de cálculo a sequência de elementos do vector a processar, na ordem necessária para calcular os componentes dos momentos. Esse sistema é constituído por 4 memórias organizadas em dois bancos, sendo cada um usado para armazenar duas cópias do mesmo segmento de dados. Enquanto um dos bancos é lido duas vezes em cada ciclo de relógio para colocar 4 valores nas entradas da unidade de cálculo (assumindo a função de banco de leitura), as duas memórias do outro banco são carregadas da memória principal com os elementos do próximo segmento a processar (funcionando como banco de escrita). Quando termina o processamento do segmento corrente, a função desses bancos de memória é trocada com a mudança do estado de uma linha de controlo, dando-se início ao processamento do novo segmento. Outra implementação possível para este sistema de memória poderia ser obtida usando dois bancos com 4 memórias cada, mantendo em cada banco 4 cópias do segmento de dados. Deste modo, seria realizado apenas um acesso de leitura de cada memória em

cada ciclo de relógio, o que permitiria usar um ciclo de relógio mais curto ou memórias com tempo de acesso mais longo. No entanto, obrigaria a duplicar o número de circuitos de memória bem como o número de linhas de endereços necessárias entre eles e o sistema gerador de endereços.

A opção pela utilização de 2 bancos com 2 memórias cada resultou da consideração de dois aspectos fundamentais, tendo em vista a futura implementação desta arquitectura num sistema reconfigurável baseado em FPGAs XILINX de uma família concreta (XC4000): a frequência máxima de operação estimada para a unidade de cálculo e o número de terminais necessários para os barramentos de endereços e controlo das memórias. Em primeiro lugar, durante o estudo e definição da arquitectura do ProHos eram já conhecidos os desempenhos de implementações de multiplicadores paralelos nos FPGAs referidos. Esse conhecimento permitiu obter estimativas iniciais para o desempenho máximo atingível pela unidade de cálculo com a estrutura e tecnologia de implementação propostas, que apontavam para frequências máximas de trabalho inferiores a 16 MHz (período igual a 62.5 ns), condicionadas pelos multiplicadores incluídos na unidade de cálculo. O recurso a memórias estáticas rápidas (15 ns) permite acomodar uma leitura em metade desse ciclo, incluindo também os tempos de atraso associados às entradas e saídas dos circuitos programáveis. O segundo aspecto considerado diz respeito à quantidade de terminais necessários para alimentar os barramentos de endereços e de controlo dos circuitos de memória. Se fossem usados 8 em vez dos 4 que foram considerados, esse número seria naturalmente duplicado o que poderia conduzir a condicionamentos de implementação relacionados com o número de terminais configuráveis disponíveis nos circuitos FPGA. Estes aspectos mostram como a opção tomada para a arquitectura deste componente fundamental do ProHos foi influenciada de forma decisiva pela tecnologia de implementação considerada. Uma situação diferente que terá de ser alvo de estudo pormenorizado, resulta de considerar a realização física deste sistema como um circuito integrado de aplicação específica que inclua também aquelas memórias.

Selectores de endereços e dados das memórias *cache*

Para agulhar os barramentos de dados e endereços destas 4 memórias, é utilizado um conjunto de circuitos selectores (*multiplexers*) que comutam os endereços e dados, distribuindo-os de acordo com a função atribuída a cada banco de *cache* em cada estágio da operação do processador. Os endereços para as 4 memórias são obtidos de circuitos dedicados que produzem 4 endereços para leitura das duas memórias do banco de leitura e 1 endereço para escrita nas duas memórias do banco de escrita. Em cada ciclo de relógio são produzidos 4 endereços que são usados para ler duas vezes cada memória do banco

de leitura, de forma a obter os 4 dados a enviar para as entradas da unidade de cálculo.

Para carregar um novo segmento de dados da memória principal, o barramento que contém os endereços para as 2 memórias do banco de escrita é ligado aos barramentos de endereços das memórias que formam esse banco. Os barramentos de dados das 4 memórias são também agulhados de acordo com a função que cada banco de *cache* realiza em cada estágio. Os barramentos de dados das memórias do banco de escrita recebem da memória principal os elementos do próximo segmento a processar. Os barramentos de dados das duas memórias do banco de leitura alimentam registos incluídos nas 4 entradas da unidade de cálculo, de modo a receber os 4 valores lidos dessas memórias em cada ciclo de relógio. Na figura 6.10 exemplifica-se o fluxo de dados no sistema de memória do ProHos, na situação em que o banco 0 funciona como banco de leitura, e o banco 1 funciona como banco de escrita.

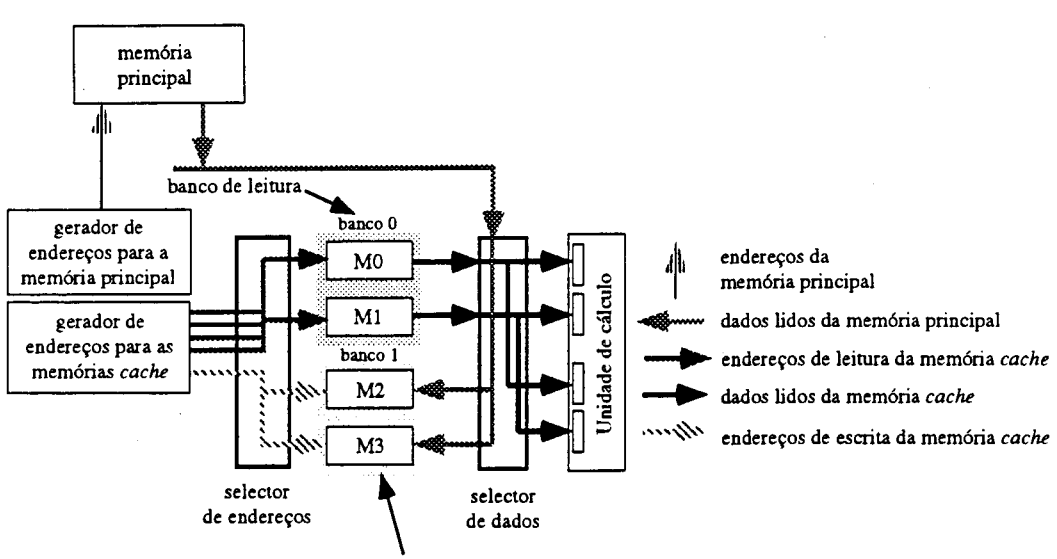


Figura 6.10: Exemplo do fluxo de dados no sistema de memória do ProHos: enquanto o banco 0 é lido para fornecer dados à unidade de cálculo, o banco 1 é carregado com o próximo segmento a processar.

Os endereços para escrita e leitura das memórias *cache* são produzidos por circuitos dedicados, baseados em contadores e apresentados na secção seguinte. Em cada ciclo de relógio é produzido um novo conjunto de 4 endereços de leitura, correspondente aos 4 elementos a enviar para as entradas da unidade de cálculo. Os endereços gerados para o banco de escrita são obtidos de um contador binário, e o próximo segmento a processar é lido da memória principal e carregado em paralelo nas duas memórias do banco de escrita, a partir do endereço zero.

6.3.6 Geradores de endereços

Estes módulos produzem as sequências de endereços necessárias para a escrita e leitura das memórias que processador contém. Existem 4 circuitos dedicados à geração de sequências de endereços para 4 funções diferentes, operando em paralelo sob comando da unidade de controlo: leitura da memória principal, escrita na memória principal, escrita nas memórias *cache* e leitura das memórias *cache*.

Da memória principal são lidos os segmentos do vector de dados a processar, para além de 3 parâmetros adicionais: amplitude do domínio dos momentos a calcular, número de segmentos e comprimento de cada segmento (P, L e M, respectivamente). Como a memória principal é acedida de forma sequencial e cada posição é lida apenas uma vez, esta sequência de endereços é gerada por um contador binário, iniciado em zero e incrementado sempre que é efectuada uma operação de leitura dessa memória.

Os resultados produzidos pela unidade de cálculo são colocados na memória principal também em posições de memória consecutivas. Por esse motivo, os endereços para escrita na memória principal são gerados por outro contador binário, que é carregado inicialmente com um valor constante e é também incrementado sempre que é efectuada uma operação de escrita na memória principal.

Os endereços para escrita nas memórias *cache* que formam o banco de escrita são igualmente produzidos por um contador binário. Antes de ser carregado um novo segmento da memória principal, esse contador é colocado em zero, de forma a armazenar esses elementos nas posições de memória com endereços desde 0 até $L-1$.

A sequência de endereços para obter da memória *cache* os elementos do segmento a processar, na ordem requerida para calcular as componentes dos momentos, é produzida por circuitos dedicados que foram construídos em torno de contadores binários. Esse módulo é constituído por duas secções com funções distintas: gerador de domínio e sequenciador de endereços. O gerador de domínio produz a sequência de valores para τ_1, τ_2 e τ_3 que correspondem ao domínio não redundante do momento de ordem 4. O sequenciador de endereços produz, para cada ponto desse domínio, a sequência de endereços apropriada para as memórias *cache* do banco de leitura.

Para cada componente dos momentos definido por τ_1, τ_2 e τ_3, deve ser fornecida à unidade de cálculo a sequência de elementos $x(k)$, $x(k+\tau_1)$, $x(k+\tau_2)$ e $x(k+\tau_3)$, com k tomando valores desde zero até $L-1-\tau_1$. Na figura 6.11 exemplifica-se o conjunto de elementos necessários desse vector para calcular $m_4(3,2,2)$, $m_3(3,2)$ e $m_2(3)$.

Conhecidos os valores de τ_1, τ_2 e τ_3, a sequência de endereços pretendida pode ser obtida com 4 contadores binários, iniciados com 0, τ_1, τ_2 e τ_3 e incrementados em cada ciclo de relógio, terminando quando o contador que é iniciado com 0 atinge $L-1-\tau_1$. O

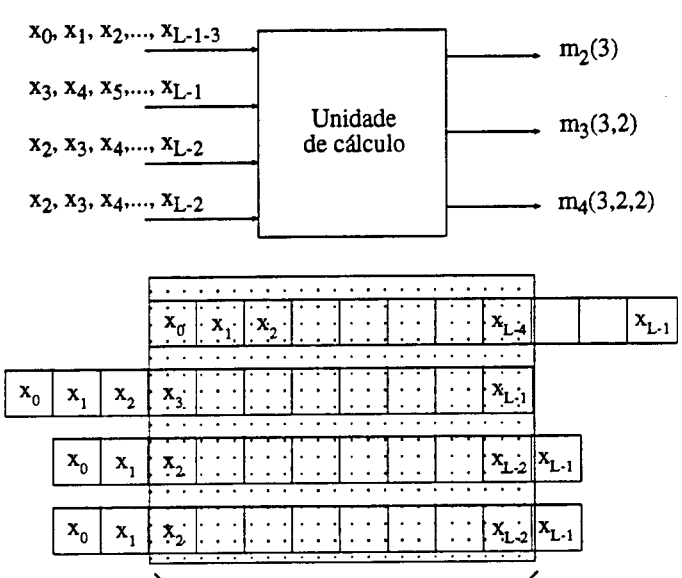


Figura 6.11: Conjunto de elementos a fornecer à unidade de cálculo para produzir $m_4(3,2,2)$, $m_3(3,2)$ e $m_2(3)$.

sequenciador de endereços assim obtido é implementado pelo circuito do lado direito da figura 6.12 e a sequência de valores produzidos equivale à realização de 4 adições e uma comparação de inteiros em cada ciclo de relógio.

O gerador de domínio corresponde à parte do lado esquerdo da figura 6.12 e produz a sequência de valores para τ_1, τ_2 e τ_3 necessária para carregar os contadores do sequenciador de endereços. Esse conjunto de valores corresponde ao domínio não redundante do momento de ordem 4, embora a região de interesse para os momentos de ordem 2 e 3 seja definida pelos valores máximos para τ_1 dados por (6.15). A implementação presente considera a região de interesse para o momento de ordem 4, sendo calculados apenas os componentes dos momentos de ordem 2 e 3 incluídos nesse domínio.

A sequência de valores para τ_1, τ_2 e τ_3 produzida pelo gerador de domínio corresponde à que é obtida pelos três ciclos encadeados seguintes:

```
Para τ1=0 até  P
   Para τ2=0 até  τ1
      Para τ3=0 até  τ2
         ...
```

O circuito que implementa esse gerador foi construído traduzindo os ciclos `Para...` representados acima em contadores e comparadores de igualdade, da forma que se mostra

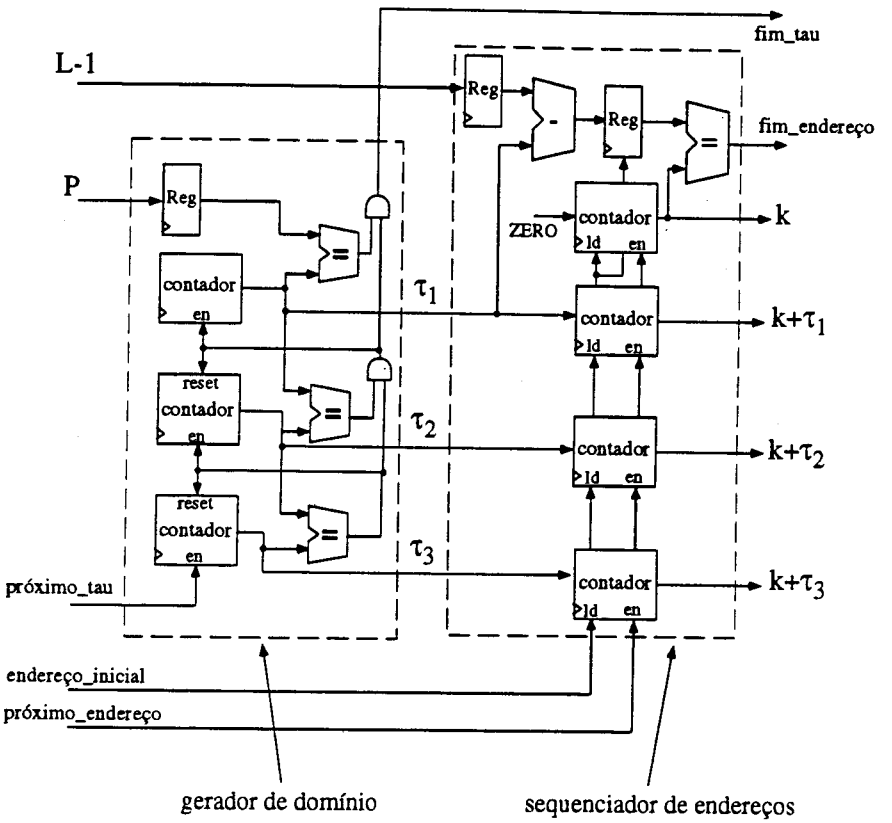


Figura 6.12: Gerador de endereços de leitura das memórias *cache*.

com o circuito do lado esquerdo da figura 6.12. O tempo necessário à geração destes valores não é crítico para o desempenho global do sistema, já que só é necessário um novo conjunto de valores para τ_1, τ_2 e τ_3 entre cada $L-1-\tau_1$ ciclos de relógio, necessários ao cálculo de cada componente dos momentos. No entanto, esta implementação apresenta complexidade reduzida e requer apenas um sinal de controlo para iterar para o próximo ponto do domínio (`proximo_tau`). Quando é produzido o último conjunto de valores, é activado o sinal `fim_tau`, que juntamente com o sinal activado quando se atinge o fim da sequência de endereços para um ponto do domínio (`fim_endereco`), sinalizam à unidade de controlo o fim do processamento desse segmento.

6.4 CVR - um sistema reconfigurável para processamento vectorial

Tendo como primeiro objectivo permitir uma realização física da arquitectura desenvolvida para o ProHos, foi projectado e construído um sistema reconfigurável destinado ao

barramento ISA de computadores pessoais do tipo PC, que foi designado por CVR—Coprocessador Vectorial Reconfigurável. O sistema contém 5 FPGAs XILINX da família 4000, 5 M*bytes* de memória RAM estática e dois circuitos programáveis ALTERA [ALT96] que agregam todos os circuitos lógicos com funcionalidade fixa que o sistema contém. A característica que torna este sistema especialmente apto para aplicações requerendo processamento vectorial consiste na existência de um conjunto de memórias estáticas rápidas, que que se destinam a fornecer de forma eficiente vectores de dados a unidades de cálculo dedicadas realizadas nos FPGAs. São também incluídos conectores que possibilitam a expansão deste sistema com cartas que acrescentem outros FPGAs ou circuitos que realizem outras funcões, que pela sua natureza (funções analógicas, por exemplo) ou complexidade não possam ser incluídas nos circuitos lógicos programáveis.

A estrutura das interligações entre os circuitos programáveis e as memórias é fixa e foi influenciada pela arquitectura desenvolvida para o ProHos. Por esse motivo, cada um dos 5 FPGAs que o sistema contém é orientado para cumprir tarefas de controlo ou processamento, em função da posição que ocupa no sistema e das ligações físicas que oferece aos restantes componentes. Apesar desta opção determinar o número de interligações que podem ser estabelecidas entre os circuitos programáveis, apresenta no entanto a vantagem de permitir que um número elevado de ligações sejam realizadas entre pinos fisicamente próximos, o que contribuiu para melhorar o desempenho global do sistema, reduzindo também a complexidade e o custo da carta de circuito impresso onde o sistema foi construído.

O CVR dispõe ainda de acesso à infra-estrutura normalizada de teste IEEE-1149.1 (BST) para os 5 circuitos FPGA. Esse barramento está também disponível nos conectores de expansão, de forma a permitir incluir na cadeia de teste outros circuitos integrados residentes em cartas de expansão que sejam ligadas a este sistema. O uso desta infra-estrutura contribui para incrementar a testabilidade deste sistema, tornando possível testar os seus componentes fundamentais de forma individual. Para além de permitir o teste dos circuitos que são implementados nos FPGAs para uma dada aplicação, o uso desta infra-estrutura tem particular interesse no teste dos circuitos de memória que o sistema contém, de forma independente dos circuitos que são configurados em cada FPGA. O controlo deste barramento requer o uso de equipamento de controlo adequado que permita deslocar as cadeias de *bits* apropriadas para cada dispositivo, de forma a definir o modo de funcionamento requerido para efectuar diferentes procedimentos de teste. O processador dedicado XBST que foi apresentado na secção 2.3.5 do capítulo 2, é um sistema que suporta o controlo desta infra-estrutura.

A configuração dos FPGAs é realizada por meio de um barramento que permite o

controlo individual de cada circuito, possibilitando assim a exploração de aplicações que tirem partido da reconfiguração dinâmica dos FPGAs. Para além disso, é também oferecido suporte para permitir a leitura do estado interno de cada circuito programável, suportado pela família de FPGA usada (função *readback* [Höf95]). Esta função facilita a tarefa de teste e depuração de erros dos circuitos lógicos implementados em cada dispositivo, permitindo conhecer o estado de nós internos que de outro modo seriam inacessíveis.

Presentemente, a divisão da funcionalidade de um sistema a realizar nesta plataforma pelos 5 circuitos programáveis é definida pelo utilizador. Os circuitos a implementar em cada FPGA são realizados com captura esquemática ou síntese a partir de descrições em ABEL ou VHDL e usando as ferramentas integradas no sistema de desenvolvimento da XILINX. Para apoiar o desenvolvimento dos circuitos a realizar em cada FPGA foi construída uma biblioteca de componentes dedicados ao uso desta plataforma. Esta biblioteca contém módulos que representam a interface de cada FPGA com os restantes componentes a que está ligado, incluindo a especificação das ligações aos pinos físicos apropriados de cada circuito. São também incluídos componentes que implementam duas unidades fundamentais para a utilização deste sistema: interface com o barramento ISA e controlo da configuração dos FPGAs.

Embora a construção deste sistema tenha sido motivada pela necessidade de implementar o ProHos e consequentemente influenciada pela sua arquitectura, outros processadores dedicados destinados a diferentes aplicações podem ser implementados neste sistema reconfigurável. Um trabalho em curso tem por objectivo desenvolver e realizar um processador dedicado para efectuar os cálculos envolvidos na determinação de intersecções entre polígonos, usado por aplicações de optimização de empacotamentos de figuras irregulares. Este trabalho insere-se num projecto de investigação financiado pelo programa PRAXIS XXI: "OPTHAR-Planeamento do corte de matéria prima em peças irregulares: optimização com *hardware* dedicado".

6.4.1 Arquitectura do CVR

O sistema reconfigurável CVR destina-se ao barramento ISA de PCs e é orientado para a implementação de processadores auxiliares dedicados com requisitos de processamento vectorial. A arquitectura do CVR é apresentada na figura 6.13, onde se mostram os tipos de FPGA incluídos na implementação actual.

O sistema é constituído pelos elementos principais seguintes: 5 FPGAs XILINX, identificados na figura como X0, X1, X2, X3 e X4 que constituem a área reconfigurável do sistema; uma memória estática de 1 M×32 *bits* que realiza a função de memória principal;

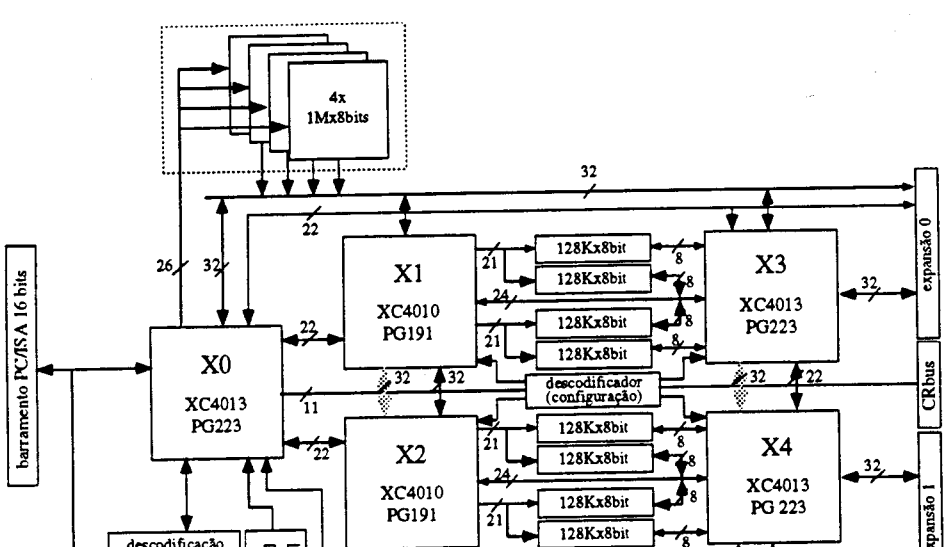


Figura 6.13: Arquitectura do CVR.

8 memórias estáticas rápidas de 128 K*bytes* cada, que se destinam a fornecer dados a unidades de cálculo realizados nos FPGAs X3 e X4; dois circuitos ALTERA com programação permanente que implementam parte dos circuitos para interface com o barramento ISA do PC, configuração do FPGA X0 e descodificação do barramento dedicado à configuração e leitura do estado interno dos FPGAs. Para além disso são também incluídos sinalizadores luminosos (*leds*) que apresentam o estado da configuração das FPGAs, um oscilador de 40 MHz, micro-interruptores para configuração de endereços e do modo de configuração dos FPGAs e vários conectores de expansão.

Interface com o computador principal

O CVR foi construído como uma carta de expansão para o barramento ISA de 16 *bits* de PCs, sendo o seu acesso realizado através de operações de escrita e leitura em portas de entrada e saída de 8 ou 16 *bits*. Para simplificar os circuitos de interface a realizar em X0 e para permitir configurar o FPGA X0 com operações de escrita realizadas pelo PC, foi incluído um circuito integrado com programação permanente (ALTERA EPM5128) que realiza funções de descodificação de endereços do barramento e produz os sinais necessários para a configuração do FPGA X0.

Este circuito descodifica blocos de 16 endereços consecutivos do espaço de endereçamento de entrada e saída do PC, correspondentes a 16 portas de 8 *bits* ou 8 portas de 16 *bits*. O endereço base é configurado com um banco de 4 micro-interruptores entre 0200h

e 03F0h. O resultado da descodificação deste endereço base é enviado para X0 como um só sinal, reduzindo dessa forma a complexidade dos circuitos de interface a realizar em X0.

Duas portas de 8 *bits* são implementadas nesse circuito e destinam-se à configuração do FPGA X0. Efectuando uma operação de escrita no endereço base adicionado de 3, o FPGA X0 é desprogramado; escrevendo um *byte* no endereço base adicionado de 1, os seus *bits* são enviados em série sincronamente com o sinal de relógio do barramento ISA (8 MHz), para os terminais de X0 que recebem a cadeia de *bits* de configuração, sendo colocado o barramento em estado de espera enquanto decorre essa operação. Deste modo é possível programar o FPGA X0 escrevendo nessa porta o bloco de *bytes* que contém a cadeia de *bits* de configuração, usando apenas uma instrução máquina do processador o que minimiza o tempo necessário para esta operação.

Este circuito controla ainda um conjunto de *leds* que assinalam o estado da configuração de X0, um micro-altifalante que pode ser usado por circuitos implementados em X0 para produzir sinalizações acústicas e uma memória série que pode ser incluída no sistema para configurar automaticamente o FPGA X0 sempre que é ligada a alimentação.

Organização de memória

O CVR dispõe de um total de 5 M*bytes* de memória RAM estática, organizados em 5 blocos endereçáveis separadamente: 1 banco de memória principal e 4 bancos de memória rápida.

A memória principal é realizada por um módulo com 4 M*bytes*-25 ns, e é acessível a partir do barramento ISA através de X0. Este módulo (Cypress, CYM1851PM-25) é organizado em 4 bancos de 1 M*byte*, com barramentos de endereços e sinais de escrita e leitura comuns, e com os 4 barramentos de dados e sinais de selecção separados para cada banco (figura 6.14). Os barramentos de endereços e de controlo desta memória são acessíveis apenas a partir do FPGA X0.

Os 4 bancos de memória rápida implementam um total de 1 M*byte*, organizados em 4 blocos com 128 K×16 *bits* endereçáveis independentemente. Dois bancos são colocados entre os FPGAs X1 e X3 e os outros dois bancos são ligados entre X2 e X4. Cada banco é realizado com 2 circuitos integrados Cypress CY7C109-15 (128 K*bytes*-15 ns), com barramentos de endereços e sinais de leitura e escrita comuns, e com sinais de selecção e barramentos de dados separados. Em cada banco, uma das memórias tem o seu barramento de dados ligado apenas ao FPGA da direita, enquanto que o barramento de dados da outra memória é comum aos FPGAs da direita e da esquerda. Na figura 6.15 mostra-se a forma como as duas memórias de cada banco são ligadas entre os dois FPGAs. Estes

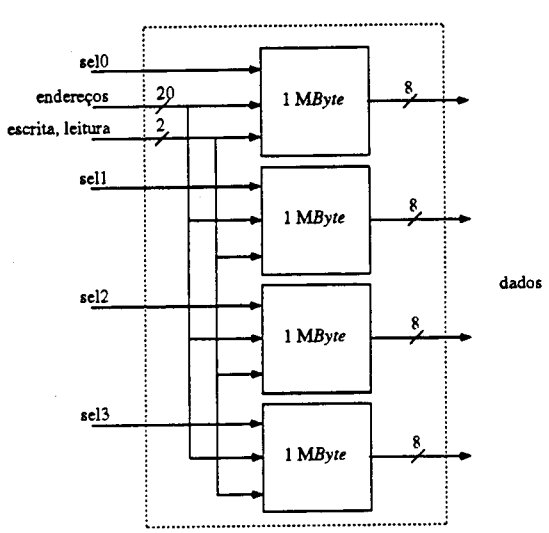


Figura 6.14: Memória principal do CVR.

4 bancos de memória destinam-se a fornecer dados a unidades de cálculo implementadas nos FPGAs X3 e X4 e permitem ler ou escrever até um máximo de 8 *bytes* em cada ciclo de acesso dessa memória.

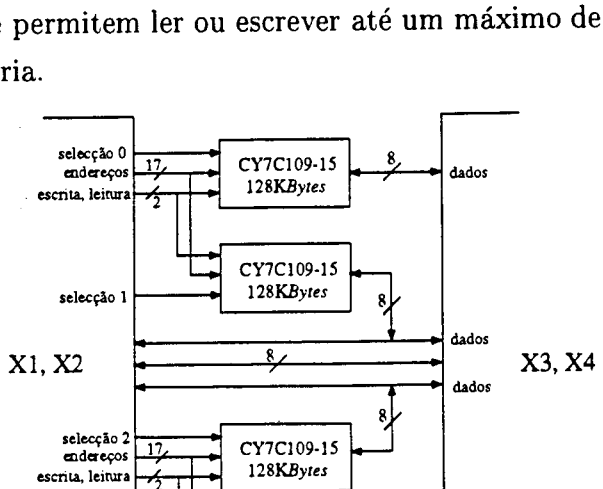


Figura 6.15: Organização dos bancos de memória rápida do CVR.

Circuitos reconfiguráveis

Os 5 FPGAs XILINX são da família 4000E, e podem ser de dois tipos: 4010E PG191 (10K portas lógicas, 191 pinos, *pin grid array*) ou 4013E PG223 (13K portas lógicas, 223

pinos, *pin grid array*) [XIL95]. Os dois circuitos são compatíveis pino-a-pino, embora o segundo ofereça mais 32 entradas ou saídas configuráveis. Isto permite utilizar FPGAs de um ou de outro tipo, variando naturalmente o número de ligações disponíveis entre os circuitos. Os barramentos que se indicam na figura 6.13 a sombreado representam 32 ligações que só existem quando os FPGAs a que ligam são do tipo 4013E PG223. Para permitir o acesso de X0 ao barramento de dados da memória principal, este deverá ser sempre do tipo 4013E PG223. Os tipos de FPGA usados na implementação actual do CVR são os que se mostram na figura 6.13: X0, X3 e X4 são 4013E PG223 e X1 e X2 são 4010E PG191, oferecendo assim uma capacidade total equivalente a 59K portas lógicas.

Embora não constitua uma imposição rígida para a utilização deste sistema, a posição que cada FPGA ocupa e as ligações fixas que apresenta com os restantes componentes é orientada para incluir certos tipos de funções específicas, em processadores dedicados implementados nesta plataforma.

Pelas ligações que são estabelecidas com componentes fundamentais do sistema, o FPGA X0 tem de conter os circuitos que realizam a interface com o barramento ISA e o controlo do barramento de configuração dos restantes 4 FPGAs. Para além disso, deve também ser implementado neste FPGA a interface com a memória principal e unidades de processamento que sejam necessárias para tratamento dos dados transferidos entre o PC e a memória principal. Outra função que tem de ser incluída em X0 são circuitos para dividir o sinal produzido pelo oscilador de cristal e gerar os sinais de relógio a distribuir para os restantes FPGAs do sistema.

Os FPGAs X1 e X2 destinam-se a conter unidades de controlo que produzam endereços para as memórias rápidas. Como também têm acesso ao barramento de dados da memória principal, também podem ser usados para construir unidades funcionais que operem sobre dados residentes na memória principal, ou para canalizar resultados produzidos por X3 ou X4 para a memória principal.

Finalmente, X3 e X4 são ligados directamente aos barramentos de dados da memória principal e dos 4 bancos de memórias rápidas e destinam-se principalmente à implementação das unidades de processamento que trabalhem sobre os dados recebidos dessas memórias. Nos conectores de expansão que o sistema contém são incluídos dois barramentos de 32 ligações cada, ligados a 32 terminais de X3 e a 32 terminais de X4.

Geração e distribuição de sinais de relógio

Os sinais de relógio para os circuitos implementados nos 5 FPGAs são obtidos a partir de um oscilador de cristal de 40 MHz ligado directamente a um terminal do FPGA X0. De X0 partem duas ligações directas para cada FPGA (incluindo o próprio X0), ligando

a terminais que alimentam circuitos dedicados para a distribuição interna de sinais de relógio (*buffers BUFGS*). Na figura 6.16 mostram-se as ligações existentes entre X0 e os restantes FPGAs, destinadas à distribuição de sinais de relógio.

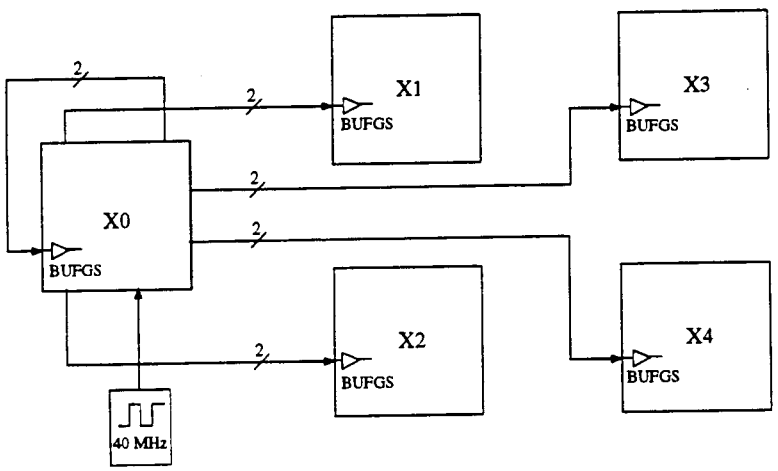


Figura 6.16: Geração e distribuição de sinais de relógio no CVR.

Configuração dos FPGAs

Todos os FPGAs incluídos no CVR são configurados em modo série, recebendo uma cadeia de *bits* com a informação de configuração, sincronamente com um sinal de relógio gerado externamente (referido por *slave serial mode* [XIL95]). Como foi já descrito anteriormente, o CVR inclui um circuito que permite a configuração do FPGA X0 através de operações de escrita realizadas pelo processador principal em endereços apropriados.

Os restantes 4 FPGAs são configurados sob controlo de X0, através de um barramento dedicado a essa função. Esse barramento contém 11 sinais, entre os quais 4 linhas de endereços, sinal de relógio, sinal de dados, iniciação de programação e linhas de estado de configuração. A cada FPGA (excepto X0) é atribuído um de 15 endereços possíveis, correspondendo o endereço 0 a nenhum circuito seleccionado. Associado a cada FPGA existe um circuito que descodifica o endereço que lhe é atribuído, e estabelece a ligação dos sinais do barramento de configuração aos terminais apropriados de cada FPGA que intervêm no processo de configuração. Os 4 descodificadores deste barramento são implementados num único circuito ALTERA EPM5064, representado no centro do desenho da figura 6.13. Este barramento é também disponibilizado para o exterior num dos conectores de expansão (CRbus), o que permite configurar outros FPGAs presentes em cartas de expansão recorrendo a esta infra-estrutura. Foi desenvolvido um circuito destinado ao FPGA X0 que permite controlar os sinais deste barramento através de operações de es-

crita e leitura em portas apropriadas de entrada e saída do PC. Na figura 6.17 mostram-se os sinais que formam este barramento e o circuito lógico que implementa o descodificador associado a cada FPGA.

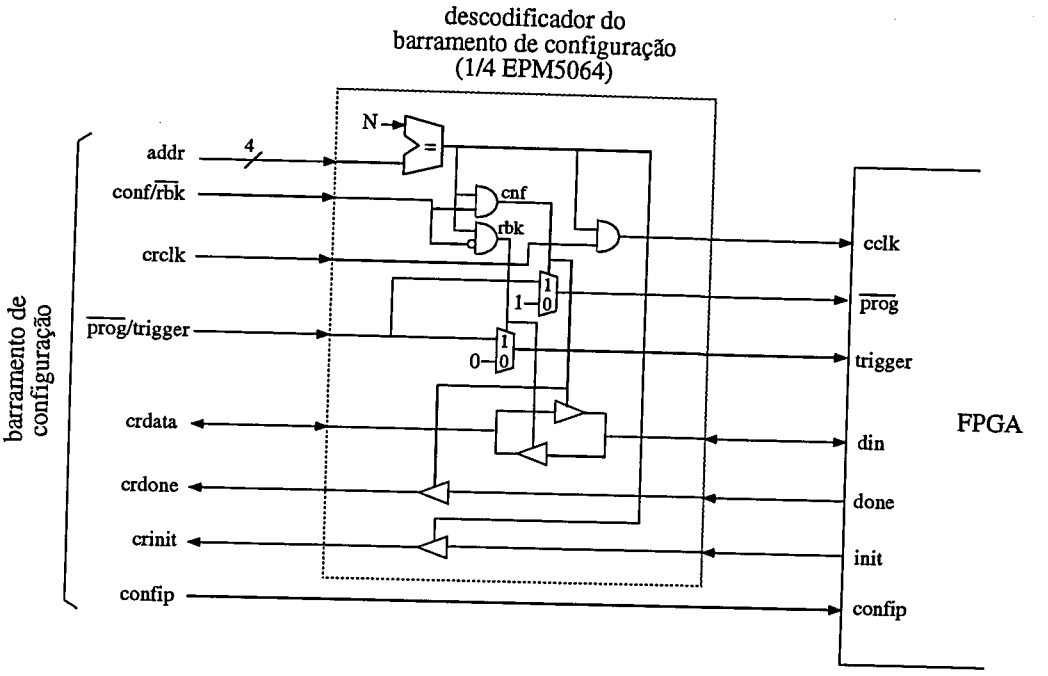


Figura 6.17: Sinais do barramento de configuração e circuito lógico do descodificador associado a cada FPGA.

As 4 linhas addr juntamente com o sinal conf/$\overline{\text{rbk}}$ seleccionam um dos FPGAs presentes no barramento e a operação a realizar: configuração ou leitura do estado interno. Em função da operação escolhida, os restantes sinais do barramento são ligados ao FPGA endereçado, sendo a linha de dados ligada no sentido determinado pela operação seleccionada. O sinal confip não é usado pelo descodificador, sendo distribuído a todos os FPGAs do barramento, com vista a sinalizar que está a decorrer uma configuração ou uma leitura de estado.

Com este barramento é possível reconfigurar um FPGA sem interferir com o funcionamento das restantes, o que permite explorar aplicações que tirem partido da reconfiguração dinâmica dos recursos desta plataforma, embora ao nível da reconfiguração integral de cada circuito FPGA. A reconfiguração dos FPGAs ligados nesse barramento pode também ser efectuada por circuitos implementados em X0 que funcionem de forma autónoma sem intervenção do processador principal, obtendo a informação de configuração da memória principal do CVR.

Para além da configuração dos FPGAs, este barramento permite também usar a função

readback suportada pela família de FPGA usada [Höf95]. Como já foi dito, esta função permite amostrar e ler uma cadeia de *bits* que representam o estado interno de pontos determinados dos blocos lógicos configuráveis de um FPGA. Deste modo, é possível monitorar o funcionamento de um circuito lógico implementado num FPGA, e ficam facilitadas as tarefas de teste, detecção e depuração de erros.

Realização do CVR

O CVR foi construído como uma carta de expansão para o barramento ISA de 16 *bits*. A carta de circuito impresso foi projectada no Centro de CAD para Electrónica do INESC-Porto e foi realizada com 8 camadas, sendo 5 para pistas de sinal, dois planos de massa e um plano de alimentação. Todas as ligações consideradas críticas foram colocadas nas camadas exteriores, possibilitando deste modo o seu acesso para permitir eventuais correcções de erros que viessem a ser detectados. Entre essas ligações estão os sinais do barramento de configuração, os sinais de controlo das memórias e as ligações destinadas à distribuição dos sinais de relógio pelos FPGAs. Na figura 6.18 mostra-se uma fotografia do sistema CVR.

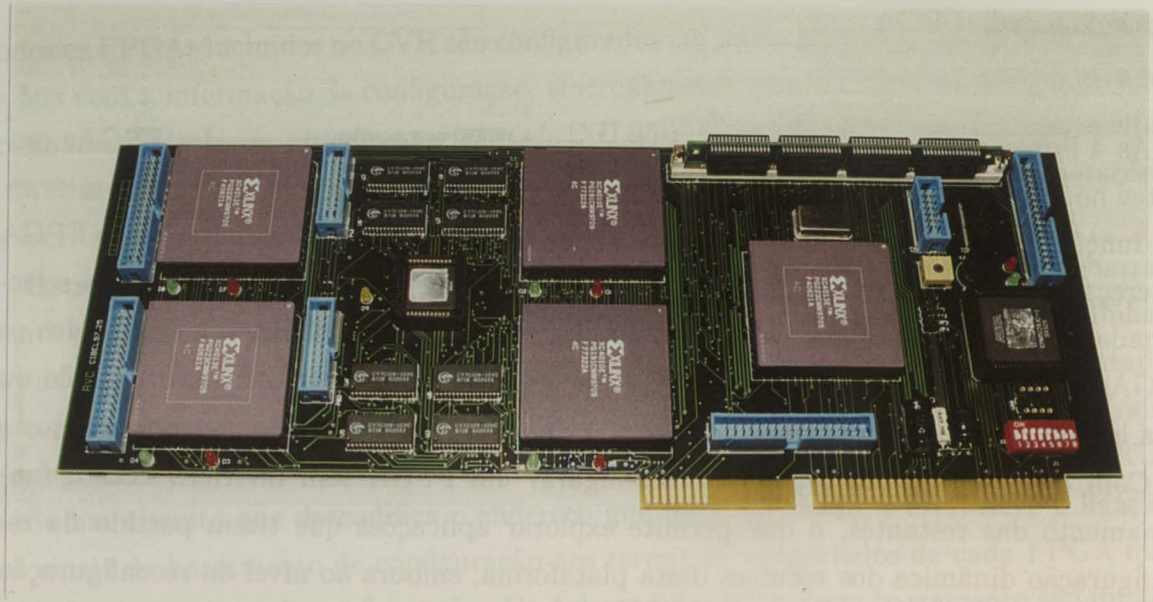


Figura 6.18: Fotografia do CVR.

6.4.2 Implementação do ProHos no CVR

O processador dedicado ProHos foi implementado no sistema reconfigurável CVR, usando o sistema de desenvolvimento da XILINX XACT Step 6.1, e as ferramentas para captura esquemática e simulação da ViewLogic. Como foi já referido antes, esta plataforma reconfigurável foi criada tendo como primeiro objectivo a realização do ProHos, pelo que foi simplificada a partição dos elementos do ProHos pelos componentes disponíveis no CVR.

Partição da funcionalidade

A divisão da arquitectura do ProHos pelos FPGAs que equipam o CVR, obdece à partição representada de forma simplificada na figura 6.19.

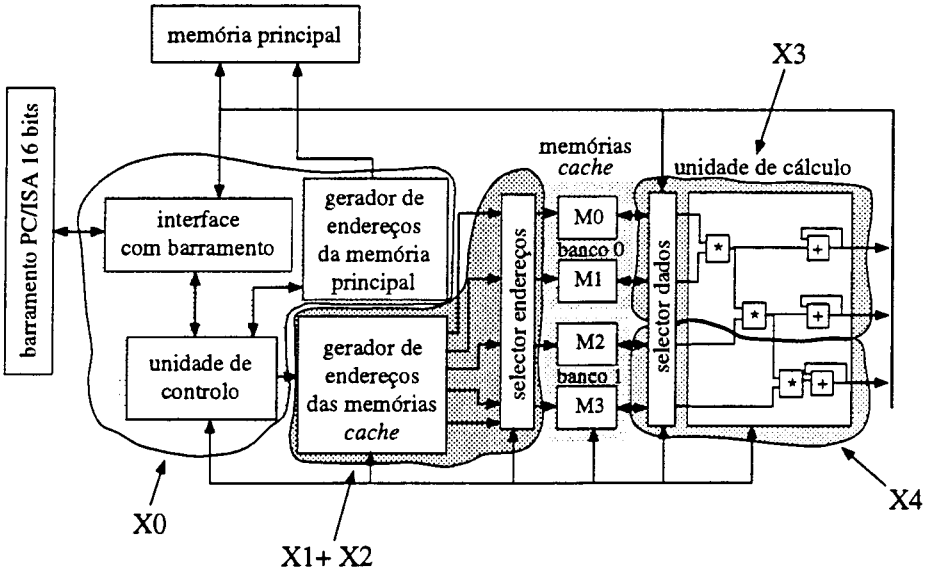


Figura 6.19: Partição do ProHos no CVR.

As 4 memórias *cache* do ProHos, identificadas na figura como M0, M1, M2 e M3, são realizadas pelos 4 bancos de memórias rápidas existentes no CVR. Como a implementação actual apenas considera dados de 8 *bits*, só é usado um dos 2 circuitos de memória que forma cada um desses bancos.

Em X0, para além dos circuitos que são necessários para o funcionamento do sistema (interface com o barramento do PC, geração de relógio e controlo das configurações dos restantes FPGAs), é implementado o controlador principal do ProHos e os contadores que produzem os endereços para leitura e escrita da memória principal. De modo a reduzir o número de transacções no barramento necessárias para transferir de blocos de dados

entre a memória principal do ProHos e o computador principal, esses contadores são incrementados automaticamente sempre que é escrita ou lida uma palavra pelo computador principal. Deste modo, a transferência de blocos de dados é efectuada com uma sequência de operações de escrita ou leitura para o mesmo endereço físico, podendo ser realizada com uma única instrução máquina dos processadores 80x86 depois de especificar o endereço inicial da memória principal do ProHos que se pretende ler ou escrever.

Nos FPGAs X1 e X2 são implementados os circuitos que produzem os endereços para leitura e escrita das memórias *cache*, juntamente com os selectores dos barramentos de endereços dessas memórias. Como estes circuitos são pouco complexos foram duplicados nos FPGAs X1 e X2, reduzindo desta forma o número de ligações necessárias entre os dois FPGAs. A sua implementação num só FPGA obrigaria a que ambos fossem do tipo 4013E PG223 para disporem do número necessário de ligações entre eles. As duas cópias destes circuitos trabalham em paralelo, embora sejam apenas usados os endereços produzidos para leitura ou para escrita, conforme estão atribuídas as funções de leitura ou escrita a cada um dos bancos de memória *cache*, em cada estágio da operação do ProHos.

A unidade de cálculo é realizada nos FPGAs X3 e X4. Uma primeira realização desta unidade foi implementada no FPGA X3, sendo X4 usado apenas para canalizar para X3 os barramentos de dados das duas memórias a que está ligado. Foi implementada também uma versão mais rápida desta unidade, recorrendo a multiplicadores *pipelined*. Nesta versão os dois primeiros multiplicadores e respectivos acumuladores são realizados em X3 e o terceiro multiplicador e acumulador é incluído em X4.

Implementação

Tendo por objectivo a validação preliminar do funcionamento dos elementos que constituem a secção de controlo e a arquitectura de memória do ProHos, foi realizada uma versão reduzida dessa parte num único FPGA XILINX 4010E. Esse circuito foi experimentado e validado com uma implementação no sistema reconfigurável ISAX (secção 2.3.5, capítulo 2). O sistema construído inclui as unidades de controlo, os circuitos de geração de endereços, os selectores de endereços e dados das memórias *cache* e versões reduzidas destas memórias e da memória principal com 64 e 128 *bytes*, respectivamente[2]. Esta realização permitiu validar o correcto funcionamento da secção de controlo do ProHos, responsável por enviar os segmentos de dados para as entradas da unidade de cálculo. Este circuito ocupa 98% dos blocos configuráveis de um FPGA 4010E PC84 e pelos resultados obtidos por simulação pode trabalhar com uma frequência máxima de relógio de

[2]A memória principal foi implementada como uma memória ROM para evitar o seu carregamento com o segmento de dados a processar.

22 MHz. No entanto, na implementação no sistema ISAX foi usada a frequência máxima de relógio de 10 MHz permitida pelo gerador de relógio disponível.

A implementação do ProHos no CVR foi realizada de acordo com a partição referida antes. O controlador principal e os circuitos geradores de endereços para a memória principal foram incluídos no FPGA X0. A funcionalidade do controlador principal descrita pelo fluxograma apresentado na figura 6.8, foi especificada na linguagem de descrição de *hardware* ABEL e compilada com a ferramenta XABEL, integrada no sistema de desenvolvimento da XILINX. O circuito completo realizado no FPGA X0 ocupa 42% dos seus blocos lógicos configuráveis.

Os circuitos geradores de endereços das memórias *cache* foram duplicados nos FPGAs X1 e X2, pelas razões que foram já explicadas antes. Em cada um, é incluído o módulo que produz a sequência de endereços de leitura das memórias *cache*, o contador que produz a sequência de endereços de escrita e os circuitos selectores que distribuem os endereços produzidos por esses módulos para as memórias apropriadas do CVR. Os esquemas dos circuitos realizados em X1 e X2 são apresentados no apêndice E. As suas implementações ocupam respectivamente 42% e 49% dos blocos configuráveis dos FPGAs X1 e X2, e ambos suportam uma frequência máxima de relógio igual a 20 MHz.

A unidade de cálculo é implementada nos FPGAs X3 e X4 e opera sobre dados de 8 *bits* em complemento para dois. Como já foi referido, para evitar a construção de circuitos divisores, a implementação actual considera segmentos de dados com dimensão igual a 1024 elementos. Deste modo, a divisão pelo comprimento do segmento dos resultados produzidos pelos acumuladores da unidade de cálculo é simplificada para uma operação de deslocamento de 10 *bits*. Os resultados produzidos pela unidade de cálculo têm assim 16, 24 e 32 *bits*, correspondendo respectivamente aos componentes dos momentos de ordem 2, 3 e 4. Os 3 multiplicadores foram desenhados manualmente, de forma a optimizar a utilização dos recursos configuráveis dos FPGAs, nomeadamente tirando partido de circuitos dedicados que a família de FPGA usada contém para implementar cadeias eficientes de propagação do *bit* de transporte em circuitos somadores. Os multiplicadores construídos têm por base a estrutura do multiplicador paralelo de Pezaris apresentado em [Hwa90], tendo sido modificada de forma a suportar operandos com números diferentes de *bits*, requeridos nesta aplicação.

Uma primeira versão da unidade de cálculo tem 5 andares de *pipeline* e inclui os 3 multiplicadores e os 3 acumuladores no FPGA X3, ocupando a totalidade dos seus blocos lógicos configuráveis. Nesta implementação, os 3 multiplicadores são realizados como circuitos combinacionais e os 5 andares de *pipeline* resultam dos registos que foram inseridos entre os multiplicadores, e entre os multiplicadores e os acumuladores. A frequência

máxima de trabalho para esta realização é de 7.5 MHz, condicionada pelo tempo de propagação do multiplicador mais lento (24 × 8 *bits*). No apêndice E apresentam-se os esquemas dos componentes fundamentais desta implementação da unidade de cálculo.

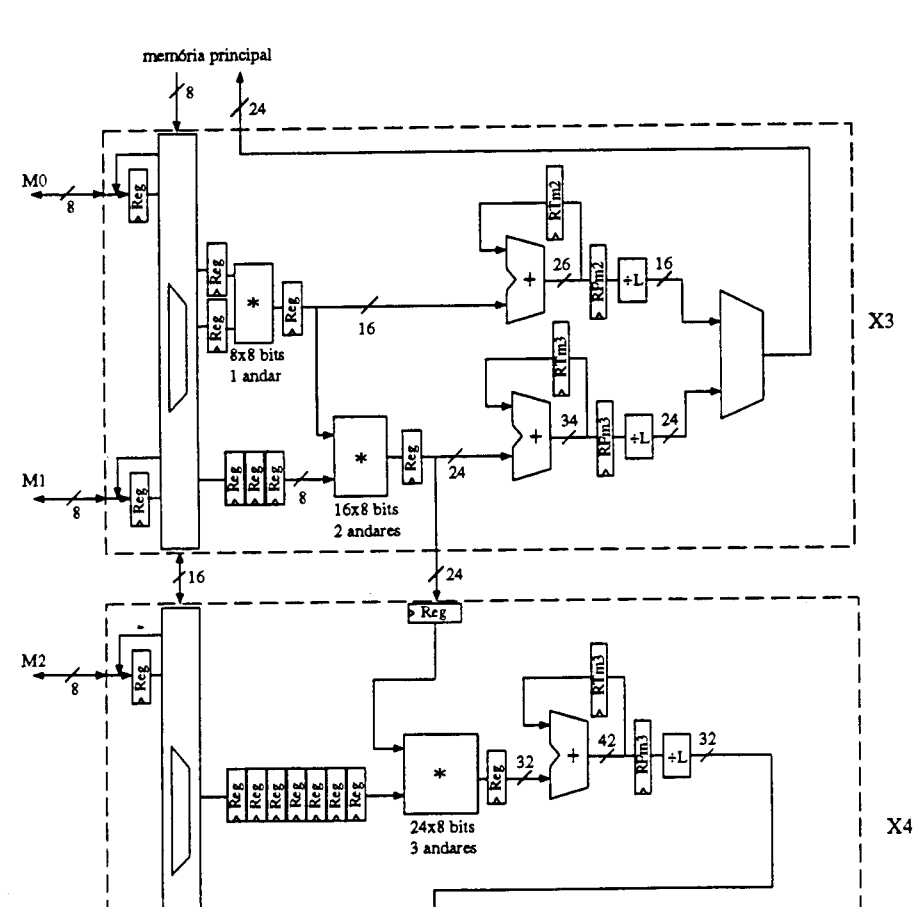


Figura 6.20: Partição da unidade de cálculo pelos FPGAs X3 e X4.

Com vista a aumentar a frequência de operação da unidade de cálculo, foi realizada outra versão recorrendo a multiplicadores *pipelined*. Estes foram obtidos modificando os usados na versão anterior, de forma a incluirem 1, 2 e 3 andares de *pipeline*, respectivamente no primeiro, segundo e terceiro multiplicadores. Esta unidade de cálculo tem 13 andares de *pipeline* e é distribuída pelos FPGAs X3 e X4, como se mostra na figura 6.20. Para simplificar o desenho, não são representados sinais de controlo nem detalhes dos circuitos selectores existentes nas entradas da unidade de cálculo. Resultados obtidos por simulação garantem uma frequência máxima de funcionamento para esta unidade igual a

12.5 MHz, ocupando 76% dos recursos configuráveis do conjunto dos FPGAs X3 e X4. Esta implementação considera igualmente segmentos de dimensão fixa e igual a 1024 elementos. No entanto, podem ser construídas outras versões desta unidade para segmentos com dimensões superiores (mas iguais a potências inteiras de dois), que apenas afectam de forma pouco significativa a complexidade dos 3 acumuladores. Por exemplo, para segmentos com 65536 elementos (2^{16}), só são acrescentados 6 andares a cada somador e aos registos acumuladores que lhes estão associados.

6.5 Resultados

A arquitectura desenvolvida para o processador dedicado apresenta um desempenho máximo que é superior a 11 operações aritméticas sobre números inteiros, em cada ciclo de relógio, uma vez preenchida a *pipeline* da unidade de cálculo. Esta unidade produz em cada ciclo o resultado de 3 multiplicações e 3 adições e na unidade que gera os endereços de leitura das memórias *cache* é realizado o equivalente a 5 operações aritméticas em cada ciclo de relógio (4 adições e uma comparação). Nestes números não são contabilizadas outras operações que não sejam efectuadas em todos os ciclos de relógio, como a geração da sequência de endereços para a escrita nas memórias *cache* e para a leitura e escrita da memória principal. Trabalhando com a frequência de relógio de 12.5 MHz prevista para a implementação mais rápida da unidade de cálculo, a arquitectura proposta atinge um desempenho máximo superior a 137 milhões de operações por segundo.

Para avaliar a rapidez conseguida por uma implementação da unidade de cálculo como um ASIC, foi projectado um circuito integrado com o sistema de projecto CADENCE *Design Framework II*, usando a tecnologia da ES2 ECPD07 (0.7μ) [ES295]. Os multiplicadores paralelos foram obtidos automaticamente com o gerador de módulos incluído nesse sistema de projecto. Resultados preliminares indicam uma área activa próxima de $7mm^2$, e uma frequência máxima de relógio igual a 25 MHz. Com esta unidade de cálculo seria assim conseguida uma rapidez dupla da apresentada pela implementação realizada em FPGA com os multiplicadores *pipelined*. No entanto, com aquela frequência de trabalho surgiriam limitações resultantes do sistema de memória considerado na arquitectura actual. A implementação deste processador dedicado como um só circuito integrado, contendo também as memórias *cache* do ProHos, permitirá contornar essas limitações, já que as memórias que podem ser incluídas num circuito integrado realizado naquela tecnologia podem ser usadas com taxas de acesso significativamente mais elevadas do que com as memórias externas usadas no CVR. Esta é naturalmente uma via a explorar para conseguir ultrapassar as limitações de rapidez resultantes da utilização de circuitos FPGA e

circuitos discretos de memória, e pode ser necessária para conseguir satisfazer requisitos de aplicações exigentes de processamento em tempo real.

Para comparar o tempo de execução do ProHos com o de uma implementação em *software*, o algoritmo para estimação dos momentos de ordem superior apresentado na figura 6.1 foi codificado em C e compilado com o compilador Visual C++ 4.0, gerando código para o processador Pentium. Para não ser excedida a gama de representação de inteiros (*overflow*) nas operações realizadas durante o cálculo dos componentes dos momentos de ordem 3 e 4 para vectores com elementos de 8 *bits*, não é suficiente realizar operações inteiras em 32 *bits*, correspondentes ao tipo `long` da implementação de C utilizada. Considerando segmentos com elementos de 8 *bits* e comprimento igual a 1024 elementos, os valores calculados pelos segundo e terceiro acumuladores, correspondentes aos componentes dos momentos de ordem 3 e 4, antes da divisão pelo comprimento do segmento, têm de ser representados por números com 34 e 42 *bits*, respectivamente. Uma vez que o compilador de C disponível não suporta dados inteiros com mais de 32 *bits* que possam garantir resultados correctos naquelas condições, optou-se por programar esse algoritmo usando operações aritméticas sobre dados daquele tipo, mas considerar que os elementos do segmento de dados são representados com um número inferior de *bits*. Assim, para garantir resultados correctos para o momento de ordem 4 em segmentos com 1024 elementos e usando operações aritméticas em 32 *bits*, só podem ser processados segmentos que apenas contenham dados representados por números de 6 *bits* em complemento para dois. Os vectores de dados usados para teste da implementação em *software* foram obtidos com o gerador de números aleatórios da biblioteca de C, produzindo uma distribuição uniforme de números inteiros no intervalo $[-31, +31]$, com valor médio próximo de zero.

A implementação em *software* foi executada num computador pessoal PC-Pentium a 120MHz e foram usados segmentos de dados com 1024 e 4096 elementos inteiros de 8 *bits*, conduzindo a valores para $P4$ iguais a 6 e 10, respectivamente, pela relação 6.15. Na tabela 6.1 comparam-se as características dos dois exemplos experimentados e os tempos de execução obtidos para o PC-Pentium e para o ProHos. O tempo de execução apresentado para o ProHos é calculado com base na duração do ciclo de relógio de 12.5 MHz e no número de ciclos necessários para completar o processamento. Nos valores apresentados na tabela é mostrado o tempo total requerido pelo ProHos, resultante da soma do tempo de processamento e do tempo necessário à transferência dos dados e resultados entre o ProHos e o processador principal.

Como os componentes dos momentos de ordem 2 e 3 são obtidos durante o cálculo dos momentos de ordem 4, o número de ciclos requeridos para completar o processamento realizado pelo ProHos é igual ao número de ciclos necessários para enviar os elementos

Tabela 6.1: Comparação dos tempos de execução para o ProHos e para uma implementação em *software* executada num PC-Pentium a 120 MHz. Os dois valores adicionados para o ProHos são o tempo de execução e o tempo de transferência de dados, respectivamente.

nº de elementos	nº componentes			nº operações ($\times 10^6$)	Tempo de execução (ms)	
	m_2	m_3	m_4		Pentium a 120 MHz	ProHos a 12.5 MHz
1024	7	28	84	0.81	54	$6.96 + 0.86 = 7.82$
4096	11	66	286	10.9	790	$94.0 + 3.2 = 97.2$

do segmento de dados para as entradas da unidade de cálculo, de forma a calcular todos os componentes do momento de ordem 4. Pelas razões que foram apresentadas antes, o número de elementos do segmento de dados que são efectivamente usados para calcular cada componente dos momentos difere do seu comprimento pelo valor de τ_1. Como o valor de τ_1 é muito menor do que o comprimento do segmento de dados ($\tau_1 \leq P4$), pode ser obtida uma boa aproximação para o número total de ciclos necessários, multiplicando o número de componentes calculados do momento de ordem 4 pelo número de elementos do segmento. Note-se que esse valor conduz a um número de ciclos superior àquele que é realmente necessário, embora nos exemplos considerados a diferença seja inferior a 0.25%. Ao número de ciclos gastos pelo processamento propriamente dito, é acrescentado o número de ciclos necessários para carregar o segmento da memória principal do ProHos para as memórias *cache*, que é igual ao número de elementos desse segmento.

O tempo necessário à transferência de dados e resultados pelo barramento ISA foi obtido medindo o tempo de execução necessário ao PC-Pentium para transferir os segmentos de dados considerados e o conjunto de resultados produzidos pelo ProHos, considerando que cada componente dos momentos ocupa 4 *bytes*. Como foi referido antes, essa operação é efectuada recorrendo às instruções máquina do processador para ler ou escrever blocos de dados em portas de entrada ou saída de 16 *bits*, conseguindo uma taxa efectiva de transferência de dados igual a 1.75 M*bytes*/s. Como se mostra na tabela 6.1, os tempos gastos com a comunicação de dados representam apenas 11% e 3.3% do tempo total de processamento no ProHos, respectivamente para os segmentos de 1024 e 4096 elmentos.

Os resultados obtidos mostram que o ProHos a trabalhar com uma frequência de relógio de 12.5 MHz consegue um ganho em rapidez de cerca de 8.1 vezes, comparando com uma realização em *software* num PC-Pentium a 120MHz. No entanto, pelas razões apontadas, esta implementação efectua operações sobre variáveis de 32 *bits*, o que não é suficiente para processar os segmentos de dados considerados pelo ProHos, com elementos

representados por números de 8 *bits* em complemento para dois. Uma implementação em *software* que realize aquele conjunto de operações nessas condições conduzirá naturalmente a tempos de execução mais longos do que os apresentados.

6.6 Conclusões

Neste capítulo apresentou-se a concepção e implementação de um processador auxiliar dedicado, que foi desenvolvido de raiz para uma aplicação bem definida na área de processamento digital de sinal. Tendo por primeiro objectivo a implementação desse processador dedicado, foi também desenvolvido e construído um novo sistema digital reconfigurável, destinado à implementação de processadores auxiliares para computadores pessoais do tipo PC. A realização do processador dedicado desenvolvido nesta plataforma reconfigurável mostrou conseguir ganhos importantes de desempenho, em relação a implementações em *software* executadas no mesmo tipo de computador a que o processador dedicado se destina.

No processador dedicado desenvolvido para o problema estudado, conseguiu-se explorar de forma eficaz as características de paralelismo e regularidade apresentadas pelo problema a resolver, traduzindo-se em circuitos lógicos para processamento e controlo que foram optimizados para a aplicação. A abordagem que foi apresentada com o desenvolvimento e implementação do ProHos na plataforma reconfigurável construída, mostra o caminho seguido na optimização e transformação de um algoritmo numa arquitectura de processamento dedicada. Ilustra um número grande de situações e de compromissos que ocorrem em desenvolvimentos desta natureza, que como se demonstrou é um meio eficaz de criar processadores dedicados eficientes para aplicações específicas.

O sistema reconfigurável CVR desenvolvido e construído no âmbito deste trabalho foi criado tendo por objectivo inicial a implementação do processador dedicado que foi criado para a aplicação estudada. O seu projecto não foi no entanto limitado às necessidades requeridas para o ProHos, já que a sua arquitectura foi idealizada tendo também em vista a implementação de outros processadores dedicados, que apresentem características de processamento semelhantes às identificadas na aplicação estudada. A realização física de processadores auxiliares nesta plataforma permite obter ganhos importantes de rapidez, como mostram os resultados obtidos com a implementação do processador dedicado desenvolvido. Além disso, o sistema reconfigurável construído é também interessante como um meio para prototipagem de novas arquitecturas, permitindo experimentar e afinar diferentes opções arquitecturais, tendo em vista realizações de processadores dedicados como ASICs.

A reconfigurabilidade do sistema CVR permitirá desenvolver e experimentar arquitecturas para processadores dedicados destinados a problemas variados, quer no âmbito da estimação de EOS quer noutras áreas de aplicação. Na área de aplicação estudada, outros problemas podem ser resolvidos de forma eficaz com arquitecturas de processamento dedicadas derivadas da que foi proposta para o ProHos. Exemplos são o cálculo dos cumulantes e poli-espectros a partir dos momentos ou a estimação dos momentos em regiões reduzidas dos seus domínios. Outra aplicação em desenvolvimento para ser implementada neste sistema reconfigurável é a de um processador dedicado ao cálculo de intersecções entre polígonos, destinado a acelerar programas para optimização de empacotamentos de figuras irregulares.

Capítulo 7

Conclusões

Sistemas digitais reconfiguráveis são um meio provado e eficaz para construir processadores dedicados, que permite aliar a rapidez conseguida por circuitos lógicos desenhados para aplicações específicas com a flexibilidade oferecida pela reconfiguração do *hardware*. O factor rapidez pode ser quantificado de forma objectiva e é um dos aspectos determinantes na opção pela construção de processadores para aplicações específicas como circuitos lógicos dedicados. Com efeito, num grande número de situações são usados como complemento de processadores convencionais para executar de forma mais rápida partes seleccionadas de uma aplicação. Flexibilidade, não podendo ser quantificada de forma objectiva, é o grande trunfo dos sistemas digitais reconfiguráveis: permitir re-usar o mesmo suporte de realização física para criar circuitos lógicos à medida das necessidades específicas de diferentes aplicações. A reconfiguração rápida e o custo nulo associado à modificação estrutural de um circuito, faz com que sistemas digitais reconfiguráveis sejam vistos cada vez mais como um suporte atraente para a realização rápida de circuitos lógicos, com especial interesse no estudo, avaliação e afinação de soluções arquitecturais para um novo sistema.

Este meio de implementação surgiu como consequência natural do aparecimento dos agregados programáveis de portas lógicas (FPGA). A flexibilidade que oferecem, permitindo a programação da estrutura do circuito lógico implementado pelo dispositivo com a mesma facilidade e rapidez com que são programados processadores convencionais, possibilita a sua re-utilização de forma arbitrária para um sem número de aplicações. Naturalmente que realizações como circuitos integrados fabricados para aplicações específicas apresentam desempenhos melhores do que realizados em FPGAs. No entanto, os custos elevados associados à realização de um novo ASIC e os tempos normalmente longos desde a sua concepção até ao produto final são factores impeditivos importantes que em muitos casos restringem o recurso a este meio de realização física. Por esses motivos, a fabricação

de diferentes circuitos integrados para aplicações variadas pode não ser praticável. É, no entanto, atraente a ideia de re-utilizar um mesmo sistema reconfigurável como meio de construir circuitos lógicos dedicados para diferentes aplicações específicas, sem custo e com risco associado muito reduzido.

A realização de processadores dedicados para aplicações específicas em sistemas reconfiguráveis requer competências num conjunto de áreas de conhecimento, que incluem o domínio da tecnologia alvo, de técnicas de projecto de sistemas digitais, de arquitecturas de computadores e compiladores, e naturalmente um profundo conhecimento do problema a resolver. Esta é por isso uma actividade interdisciplinar que requer uma abordagem integrada em que decisões arquitecturais são ditadas pelas características do problema a resolver e não podem ser completamente desligadas de aspectos concretos de realização física, resultantes de condicionamentos impostos pela tecnologia de implementação empregue.

O trabalho apresentado nesta dissertação revelou variados aspectos, problemas e soluções que surgem no decorrer da concepção e desenvolvimento de processadores dedicados em sistemas reconfiguráveis, e que foram aqui abordados com estudos realizados em torno de casos concretos. Os problemas tratados abarcam a construção de compiladores de linguagens de alto nível, selecção e construção de operadores dedicados, optimização da configuração para uma arquitectura programável e reconfigurável e as várias etapas envolvidas na construção de raiz de um processador dedicado para uma aplicação específica, passando pelo estudo do problema, concepção da arquitectura e a sua implementação num novo sistema digital reconfigurável construído propositadamente para essa aplicação.

A criação de suportes de *software* para programação de arquitecturas dedicadas programáveis com linguagens de programação de alto nível é um problema que foi abordado com o trabalho desenvolvido em torno de um processador dedicado. Este estudo teve como base um processador vectorial construído especificamente para uma aplicação de controlo em tempo real, que foi desenvolvido como um circuito com funcionalidade fixa não exibindo por isso reconfigurabilidade estrutural do seu circuito lógico. Foi adaptado um compilador de C configurável (GNU C) para essa arquitectura e foi implementada uma técnica de sequenciamento para permitir compactar o microcódigo executado pelo processador. O sistema construído permite traduzir programas numa linguagem de alto nível para instruções executáveis pelo processador, possibilitando desse modo experimentar e comparar os resultados obtidos por diferentes algoritmos sem requerer a sua codificação manual com operações de baixo nível. O problema tratado neste estudo ilustra a tarefa

fundamental que consiste na compilação para instruções específicas de processadores dedicados programáveis. Como os resultados obtidos demonstraram, o recurso a compiladores configuráveis que foram concebidos para servirem arquitecturas convencionais de processadores, permite simplificar uma parte significativa do trabalho de construir um novo compilador para um processador específico. No entanto, as particularidades que arquitecturas programáveis não convencionais podem exibir, obrigam a desenvolver e implementar técnicas que complementem as ferramentas construídas com base em compiladores concebidos para processadores convencionais.

Inspirado na arquitectura do processador anterior e procurando generalizar o âmbito da sua aplicação foi proposta uma arquitectura reconfigurável para um processador programável. Este é baseado em unidades funcionais constituídas por circuitos lógicos programáveis do tipo FPGA, que permitem criar os operadores necessários para tarefas específicas encontradas numa aplicação, construídos como circuitos desenhados de forma optimizada para essas tarefas. Deste modo foi acrescentada reconfigurabilidade estrutural à flexibilidade já oferecida pela arquitectura anterior. Foram desenhados e implementados circuitos optimizados para operadores destinados a tarefas seleccionadas e foram comparados os seus desempenhos com implementações optimizadas em *software*. Os exemplos apresentados ilustram situações paradigmáticas onde o recurso a operadores realizados como circuitos desenhados expressamente para tarefas seleccionadas pode conseguir benefícios importantes de rapidez.

Com a flexibilidade acrescida pela introdução das unidades funcionais reconfiguráveis, a adaptação da arquitectura programável proposta para diferentes aplicações apresenta problemas importantes que transcendem a compilação de código de alto nível. Na realidade, a selecção do conjunto óptimo de operadores a implementar num número limitado de recursos programáveis é uma tarefa decisiva para optimizar a arquitectura considerada para uma aplicação específica. Com a liberdade permitida para a escolha da funcionalidade a colocar nas unidades funcionais, o processo de tradução de uma sequência de operações para uma configuração dessa arquitectura engloba os passos nucleares de optimização em síntese de alto nível de circuitos digitais, que têm de ser abordados numa perspectiva de optimização conjunta. Com efeito, para além de construir e seleccionar os operadores que melhor satisfazem as necessidades de uma aplicação, é também necessário determinar uma ordenação temporal das operações do problema e uma sua atribuição aos operadores criados nas unidades funcionais. Foram considerados dois cenários que traduzem dois modos diferentes de reconfiguração suportados pelas unidades funcionais: reconfiguração estática e reconfiguração dinâmica. No primeiro considerou-se que as unidades funcionais mantêm um conjunto de operadores durante a resolução do problema.

Considerando reconfiguração dinâmica, permite-se que a função realizada por uma unidade funcional seja objecto de alteração durante diferentes estágios de resolução de um problema. Neste cenário, a complexidade do problema de optimização considerado é acrescida de forma significativa, pela necessidade de encadear de modo adequado as fases de reconfiguração com os períodos de execução das unidades funcionais.

Para resolver esse problema foi desenvolvida uma solução baseada na técnica de optimização conhecida por *Simulated Annealing*. Para um problema dado, permite optimizar de forma simultânea o conjunto de operadores a colocar nas unidades funcionais, a ordenação temporal das operações e a atribuição das operações aos operadores seleccionados. Embora tenham já sido propostas por outros autores técnicas para optimização global do problema considerado no contexto de síntese de alto nível de circuitos digitais, não são contempladas por essas abordagens algumas características fundamentais dos modelos propostos, nomeadamente no que se referem à característica de reconfiguração dinâmica considerada para as unidades funcionais. Este problema de optimização tem carácter geral e não é específico da arquitectura alvo que foi considerada neste trabalho. Surge igualmente em situações similares onde seja colocado o problema da avaliação de compromissos resultantes de variadas alternativas de realizações para operadores específicos. A solução desenvolvida dá resposta a estes problemas, e tem um âmbito de aplicação que ultrapassa a arquitectura alvo considerada.

O estudo de um problema concreto na área do processamento digital de sinal conduziu ao desenvolvimento de um processador auxiliar dedicado, destinado a acelerar uma tarefa bem definida e computacionalmente intensiva: estimação de momentos de ordem superior. Este trabalho surgiu da necessidade de efectuar aqueles cálculos de forma mais rápida do que um processador convencional, para permitir usar em tempo real os valores calculados, por algoritmos de codificação e compressão de vídeo digital para baixas cadências em aplicações destinadas a serem executadas em máquinas de baixo custo. Para isso foi desenvolvido o processador dedicado ProHos, desenhado expressamente para aquela tarefa. A natureza do problema estudado conduziu a uma arquitectura que realiza processamento vectorial e explora de forma eficiente o paralelismo exibido pelas operações realizadas. Apesar disso, apresenta complexidade moderada adequada à implementação sobre circuitos digitais programáveis FPGA.

A realização física do processador dedicado desenvolvido, motivou a concepção e construção de um novo sistema reconfigurável baseado em circuitos FPGA, que foi criado tendo em vista a implementação de processadores dedicados com requisitos de processamento vectorial. Apesar da arquitectura criada para o processador dedicado ProHos ter influenciado de forma decisiva a concepção deste sistema, foi mantida uma estrutura

flexível de forma a possibilitar a realização física de outras arquitecturas destinadas a diferentes aplicações. A realização física do ProHos neste sistema reconfigurável mostrou conseguir um aumento de rapidez superior a 8 vezes para a aplicação seleccionada, quando comparado com uma implementação optimizada em *software* e executada no mesmo tipo de computador a que o ProHos se destina. A reconfigurabilidade oferecida pelo sistema desenvolvido permitirá naturalmente a sua re-utilização para implementar outros processadores auxiliares para aplicações específicas que exibam requisitos de processamento vectorial análogos aos identificados no problema resolvido pelo ProHos.

7.1 Perspectivas de trabalho futuro

Desenvolvimentos futuros perspectivados no seguimento do trabalho que se desenvolveu e apresentou nesta dissertação seguirão a linha de aplicação de sistemas digitais reconfiguráveis na implementação de unidades de processamento auxiliar para aplicações específicas. Neste sentido, duas áreas de trabalho fundamentais se prefiguram: uma diz respeito ao desenvolvimento de ferramentas de *software* para apoiar a implementação de processadores dedicados na plataforma reconfigurável construída. A outra tem a ver com a exploração de aplicações em diferentes campos de Engenharia que possam tirar partido das vantagens oferecidas por processadores dedicados, realizados nesta ou noutro tipo de plataforma.

Em torno do sistema reconfigurável construído, alguns desenvolvimentos são importantes para tornar efectiva a sua utilização noutras aplicações:

- Construir modelos e ferramentas de *software* para apoio ao desenvolvimento de processadores dedicados na plataforma construída. Um trabalho fundamental consiste na construção de modelos abstractos da plataforma reconfigurável desenvolvida, que permitam a simulação de um sistema nela realizado a um nível elevado de abstracção. Isto é importante para avaliar eficazmente os compromissos resultantes de diferentes alternativas para a divisão da funcionalidade de um sistema pelos recursos disponíveis na plataforma reconfigurável.

- Desenvolver modelos para processadores auxiliares, adequados à implementação nos recursos deste sistema, e construir ferramentas para a sua personalização, partindo de problemas especificados em linguagens de programação de alto nível. Um modelo de arquitectura a considerar é semelhante à que foi proposta no capítulo 4, e que beneficiará naturalmente das técnicas de optimização que foram desenvolvidas no âmbito deste trabalho.

- Explorar arquitecturas de processadores dedicados que tirem partido da reconfiguração dinâmica dos circuitos FPGA. Apesar de neste sistema isso ser condicionado pelos tempos de reconfiguração integral dos FPGAs, poderá mesmo assim ser interessante em aplicações de estimação de Estatísticas de Ordem Superior, re-utilizando o mesmo *hardware* físico para diferentes etapas requeridas pelos algoritmos de estimação.

Para além do desenvolvimento de ferramentas de apoio, uma linha de intervenção importante será identificar outras aplicações que possam retirar proveito de processadores dedicados. Neste sentido, foram já seleccionadas duas áreas onde se identificaram problemas que podem beneficiar de soluções baseadas em processadores auxiliares realizados nesta plataforma reconfigurável. Um foi já referido nesta dissertação e consiste na determinação de sobreposições entre listas de polígonos, operação usada intensivamente por processos de optimização de empacotamentos de peças irregulares, com vista à minimização de desperdícios. Outra área de aplicação é a que motivou a construção deste sistema: codificação e compressão de vídeo. Vários problemas conhecidos nessa área são computacionalmente pesados quando realizados em processadores convencionais, mas pela natureza de paralelismo exibido pelas operações envolvidas sugerem ser adequados à implementação nesta plataforma (por exemplo, operações para comparação de blocos de imagem usadas em técnicas de estimação de movimento).

Futuros desenvolvimentos conduzirão inevitavelmente ao nascimento de novos sistemas reconfiguráveis, de forma a satisfazer necessidades de aplicações com outros níveis de exigência e a acompanhar os progressos tecnológicos da microelectrónica a que os circuitos FPGA não são alheios. No que se refere à evolução do sistema construído, uma versão futura será realizada com interface para o barramento PCI disponível nos PCs, o que permitirá contornar as limitações resultantes do fraco desempenho do barramento ISA suportado presentemente.

Um ambiente universitário onde se reúnem competências relacionadas com diferentes ramos de Engenharia constitui uma boa fonte de problemas em áreas muito diversas que podem vir a beneficiar das vantagens de processadores dedicados realizados sobre plataformas reconfiguráveis. Os trabalhos já realizados em colaboração com grupos de investigação na área da codificação de audio e vídeo, e do teste de circuitos electrónicos, bem como outros presentemente em curso ligados a problemas concretos de Investigação Operacional, ilustram aspectos de interdisciplinaridade que interessa salientar.

O trabalho desenvolvido e apresentado nesta dissertação mostra a influência recíproca

que pode existir entre áreas aparentemente não relacionadas. Pretende ser uma contribuição para que relações deste tipo possam ser intensificadas conduzindo a formas inovadoras de aplicação dos avanços da tecnologia microelectrónica e da arquitectura de processadores dedicados a novas áreas e novos problemas.

Referências

[AA95] Peter M. Athanas e A. Lynn Abbot, "Real-Time Image Processing on a Custom Computing Platform", *Computer*, vol. 28, nº 2, pp. 16–24, Fevereiro 1995.

[AACE94] A. Lynn Abbott, Peter M. Athanas, Luna Chen e Robert L. Elliott, "Finding Lines and Building Pyramids with Splash 2", *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 155–163, Abril 1994.

[Ach93] Hans Achatz, "Extended 0/1 LP Formulation for the Scheduling Problem in High-Level Synthesis", *Proceedings of the European Design Automation Conference, EURODAC*, pp. 226–231, 1993.

[AG95] John M. M. Anderson e Georgios B. Giannakis, "Image Motion Estimation Algorithm Using Cumulants", *IEEE Transactions on Image Processing*, vol. 4, nº 3, pp. 346–357, Março 1995.

[AHG94] J. C. Alves, M. Held e M. Glesner, "A Code Generator for an Application Specific Pipelined Processor", *Proceedings of the Mediterrean Electrotechnical Conference, MELECON'94*, pp. 154–157, Abril 1994.

[ALT96] ALTERA, *Data Book*, 1996.

[Alv89] José Carlos Alves, Atega—software gráfico para o controlador EGA, Provas de Aptidão Pedagógica e Capacidade Científica, Faculdade de Engenharia da Universidade do Porto, 1989.

[Alv93] J. C. Alves, Configuring gcc for an application specific pipelined processor, Relatório Interno, Institute für Datentechnik, FG Mikroelektronische Systeme, Technische Hochschule Darmstadt, Abril 1993.

[And92] Thomas N. Anderson, *TASM—A Table Driven Cross Assembler for the MSDOS Environment, User's Manual V 2.9*, 1992.

[APCRM96] J. C. Alves, A. Puga, L. Corte-Real e J. S. Matos, "ProHos-1—A vector processor for the efficient estimation of higher-order moments", *Proceedings of the Second International Meeting on Vector and Parallel Processing (Systems and Applications)—VecPar96*, Setembro 1996.

[APCRM97] J. C. Alves, A. Puga, L. Corte-Real e J. S. Matos, "A Vector Architecture for Higher-Order Moments Estimation", *Proceedings of the International Conference on Acoustics, Speech and Signal Processing, ICASSP'97,*, pp. 4145–4148, Abril 1997.

[Arn93] Jeffrey M. Arnold, "The Splash 2 Software Environment", *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 88–93, Abril 1993.

[AS93] Peter M. Athanas e Harvey F. Silverman, "Processor Reconfiguration Through Instruction-Set Metamorphosis", *Computer*, vol. 26, nº 3, pp. 11–17, 1993.

[ASU86] A. V. Aho, R. Sethi e J. D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986.

[ATM95] Atmel Corporation, *Configurable Logic - Design and Applications Book*, 1995.

[AWG94] Lalit Agarwal, Mike Wazlowski e Sumit Ghosh, "An Asynchronous Approach to Efficient Execution of Programs on Adaptative Architectures Utilizing FPGAs", *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 101–110, 1994.

[BAK96] Duncan A. Buell, Jeffrey M. Arnold e Walter J. Kleinfelder, *Splash 2—FPGAs in a Custom Computing Machine*, IEEE Computer Society Press, 1996.

[BIJ+95] Luiz André Barroso, Sasan Iman, Jaeheon Jeong, Koray Öner e Michel Dubois, "RPM: A Rapid Prototyping Engine for Multiprocessor Systems", *Computer*, vol. 28, pp. 26–34, 1995.

[BMT+92] David E. Van Den Bout, Joseph N. Morris, Douglas Thomae, Scott Labrozzi, Scot Wingo e Dean Hallman, "AnyBoard: An FPGA-Based, Reconfigurable System", *IEEE Design and Test of Computers*, vol. 9, nº 3, pp. 21–30, Setembro 1992.

[BRV89] Patrice Bertin, Didier Roncin e Jean Vuillemin, Introduction to Programmable Active Memories, Relatório Interno, Digital Paris Research Laboratory, Junho 1989.

[BRV92] Patrice Bertin, Didier Roncin e Jean Vuillemin, "Programmable Active Memories: a Performance Assessment", *First International ACM/SIGDA Workshop on FPGAs*, pp. 1–10, Janeiro 1992.

[BT94] Patrice Bertin e Hervé Touati, "PAM Programming Environments: Practice and Experience", *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 133–138, 1994.

[CA94] William H. Crouse e Donald L. Anglin, *Automotive Engines, (eighth edition)*, McGraw-Hill International Editions, 1994.

[CH96] David A. Clark e Brad L. Hutchings, "The DISC Programming Environment", *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, Abril 1996.

[Cha94] Pak K. Chan, A field-programmable prototyping board: XC4000 BORG, user's guide, Relatório Interno, Board of Studies in Computer Engineering, University of California, Santa Cruz, 1994.

[CPGP94] Joël Champeau, Luc Le Pape, Eric Gautrin e Laurent Perraudeau, "Flexible Parallel FPGA-Based Architectures with ArMen", *Proceedings of the Hawai International Conference of System Sciences*, Janeiro 1994.

[CSM92] Pak K. Chan. Martine D. F. Schlag e Marcelo Martin, "BORG: A Reconfigurable Prototyping Board Using Field-Programmable Gate Arrays", *Proceedings of the 1st ACM/SIGDA Workshop on FPGAs*, pp. 45–51, 1992.

[CWM94] Samit Chaudhuri, Robert Walker e John Mitchell, "Analysing and Exploiting the Structure of the Constraints in the ILP Approach to the Scheduling Problem", *IEEE Transactions on Very Large Scale Integration*, vol. 2, nº 4, pp. 456–471, Dezembro 1994.

[dB93] David E. Van den Bout, "The AnyBoard: Programming and Enhancements", *Proceedings of the IEEE Workshop fo FPGAs for Custom Computing Machnies*, pp. 68–77, Abril 1993.

[DeH94] André DeHon, "DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century", *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 31–39, Abril 1994.

[DLSM81] S. Davidson, D. Landskov, B. D. Shriver e P. W. Mallett, "Some experiments in local microcode compaction for horizontal machines", *IEEE Transactions on Computers*, vol. C-30, nº 7, pp. 460–477, Julho 1981.

[DN89] S. Devadas e A. R. Newton, "Algorithms for Hardware Allocation in Data Path Synthesis", *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 8, nº 7, pp. 768–781, Julho 1989.

[dSLAM97] José Machado da Silva, Ana C. Leão, José C. Alves e José S. Matos, "Implementation of Mixed-Current/Voltage Testing Using IEEE P1149.4", *a publicar em Proceedings of the International Test Conference, ITC'97*, Novembro 1997.

[EH94] James G. Eldredge e Brad L. Hutchings, "Density Enhancement of a Neural Network Using FPGAs and Run-Time Reconfiguration", *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 180–188, 1994.

[EH96] James G. Eldredge e Brad L. Hutchings, "Run-Time Reconfigurarion: A Method for Enhancing the Functional Density of SRAM-based FPGAs", *Journal of VLSI Sinal Processing*, vol. 12, pp. 67–86, 1996.

[EHB93] Rolf Ernst, Jörg Henkel e Thomas Benner, "Hardware-Software Cosynthesis for Microcontrollers", *IEEE Design & Test of Computers*, pp. 64–75, Setembro 1993.

[ES295] European Silicon Structures, *ECPD07 Library Databook*, 1995.

[Fer92] José Manuel Martins Ferreira, *O Teste de Cartas de Circuito Impresso com BST: Arquitectura de um Controlador Residente e Geração Automática do Programa de Teste*, Tese de Doutoramento, Faculdade de Engenharia de Universidade do Porto, 1992.

[Fou93] Patrick W. Foulk, "Data-folding in SRAM configurable FPGAs", *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 163–171, Abril 1993.

[GE93] Catherine H. Gebotys e Mohamed I. Elmasry, "Global Optimization Approach for Architectural Synthesis", *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 12, nº 9, pp. 1266–1278, Setembro 1993.

[GHB93] Jonathan Greene, Esmat Hamdy e Sam Beal, "Antifuse Field Programmable Gate Arrays", *Proceedings of the IEEE*, vol. 81, nº 7, pp. 1042–1056, Julho 1993.

[GHK+91] Maya Gokhale, William Holmes, Andrew Kosper, Sara Lucas, Ronald Minnich, Douglas Sweely e Daniel Lopresti, "Building and Using a Highly Parallel Programmable Logic Array", *Computer*, vol. 24, nº 1, pp. 81–89, Janeiro 1991.

[GJM94] Rajesh K. Gupta, Claudionor N. Coelho Jr. e Giovanni De Micheli, "Program Implementation Schemes for Hardware-Software Systems", *IEEE Computer*, pp. 48–55, Janeiro 1994.

[GM93] Maya Gokhale e Ron Minnich, "FPGA Computing in a Data Parallel C", *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 94–101, Abril 1993.

[GWDL92a] Daniel Gajski, Allen Wu, Nikil Dutt e Steve Lin, *High-Level Synthesis—Introduction to Chip and Systems Design*, capítulo 7, Scheduling, pp. 213–258, Kluwer Academic Publishers, 1992.

[GWDL92b] Daniel Gajski, Allen Wu, Nikil Dutt e Steve Lin, *High-Level Synthesis—Introduction to Chip and Systems Design*, Kluwer Academic Publishers, 1992.

[Hey88] John B. Heywood, *Internal Combustion Engine Fundamentals*, capítulo 12, Engine Heat Transfer, McGRAW-HILL International Editions, Automotive Technology Series, 1988.

[HH95] J. D. Hadley e Brad L. Hutchings, "Designing a partially reconfigured system", *SPIE's Photonics East: FPGAs for Fast Board Development and Reconfigurable Computing*, pp. 210–220, Outubro 1995.

[HKR94] Reiner W. Hartenstein, Rainer Kress e Helmut Reinig, "A Reconfigurable Data-Driver ALU for Xputers", *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 139–146, 1994.

[HL90] Frederick S. Hillier e Gerald J. Lieberman, *Introduction to Operations Research*, McGRAW-HILL, 1990.

[HLH91] Cheng-Tsung Hwang, Jiahn-Hung Lee e Yu-Chin Hsu, "A Formal Approach to the Scheduling Problem in High-Level Synthesis", *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 10, nº 4, pp. 464–475, apr 1991.

[Hoa93] Dzung T. Hoang, "Searching Genetic Databases on Splash 2", *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 185–181, Abril 1993.

[Höf95] Wolfgang Höflich, *The Programmable Logic Data Book, Xapp 015.000: Using the XC4000 Readback Capability,*, XILINX, 1995.

[HP83] L. J. Hafer e A. C. Parker, "A Formal Method for the Specification, Analysis and Design of Register-Transfer Level Digital Logic", *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 2, nº 1, pp. 4–18, Janeiro 1983.

[HP90] J. Hennessy e D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Palo Alto, California, 1990.

[HW97] John R. Hauser e John Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor", *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 24–33, Abril 1997.

[Hwa90] Kai Hwang, *Computer Arithmetic—Principles, Architecture and Design*, capítulo Chapter 6—Iterative Cellular Array Multipliers, pp. 161–211, John Wiley&Sons, 1990.

[HWGG93] H.-J. Herpel, N. Wehn, M. Gasteier e M. Glesner, "A Reconfigurable Computer for Embedded Control Applications", *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 111–120, Abril 1993.

[IJ95] Tarek Ben Ismail e Ahmed Amine Jerraya, "Synthesis Steps and Design Models for Codesign", *IEEE Computer*, pp. 44–52, Fevereiro 1995.

[IS93] Christian Iseli e Eduardo Sanchez, "Spyder: A Reconfigurable VLIW Processor using FPGAs", *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 17–24, Abril 1993.

[IS95] Christian Iseli e Eduardo Sanchez, "A C++ Compiler for FPGA Custom Execution Units Synthesis", *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 173–179, Abril 1995.

[Ise96] Christian Iseli, *A Reconfigurable Processor Development System*, Tese de Doutoramento, École Polytechnique Féderale de Lausanne, 1996.

[JEÖH94] Axel Jantsch, Peter Ellervee, Johny Öberg e Ahmed Hemani, "A Case Study on Hardware/Software Partitioning", *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 111–118, 1994.

[Kay88] S. M. Kay, *Modern Spectral Estimation*, Prentice Hall, Englewood Cliffs, NJ, 1988.

[KB92] Tom Kean e Irene Buchanan, "The Use of FPGAs in a Novel Computing Subsystem", *Proceedings of the First International ACM/SIGDA Workshop on FPGAs*, pp. 60–66, Fevereiro 1992.

[KGV83] S. Kirkpatrick, C. D. Gellat e M. P. Vecchi, "Optimization by Simulated Annealing", *Science*, vol. 220, pp. 671–680, 1983.

[KM95] John N. Kalamatianos e Elias S. Manolakos, "Parallel Computation of Higher-Order Moments on the MasPar-1 Machine", *Proceedings of the International Conference on Acoustics Speech and Signal Processing, ICASSP'95*, Maio 1995.

[KWK85] S. Y. Kung, H. J. Whitehouse e T. Kailath, *VLSI and Modern Signal Processing*, Prentice Hall, 1985.

[LA89] P. J. M. Van Laarhoven e E. H. L. Aarts, *Simulated Annealing: Theory and Applications*, Kluwer Academic Publishers, 1989.

[LA93] X.-P. Ling e H. Amano, "WASMII: A Data Driven Computer on a Virtual Hardware", *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 33–42, Abril 1993.

[LGW91a] A. Laudenbach, M. Glesner e N. Wehn, "VLSI System Design for the Control of High Performance Combustion Engines", *Proceedings of the IFIP VLSI'91 Conference*, pp. 6b1.1–6b1.10, 1991.

[LGW+91b] A. Laudenbach, M. Glesner, P. Windirsch, J. Plahl e W. Clemens, "VLSI Chip Set for Floating Point Vector Processing", *Proceedings of the Euro ASIC'91 Conference*, pp. 154–157, Maio 1991.

[LHL89] Jiahn-Hung Lee, Yu-Chin Hsu e Youn-Long Lin, "A New Integer Linear Programming Formulation for the Scheduling Problem in Data Path Synthesis", *Proceedings of ICCAD*, pp. 20–23, 1989.

[LM86] M. Lundy e A. Mees, "Convergence of an Annealing Algorithm", *Mathematic Programming*, vol. 34, pp. 111–124, 1986.

[LM95] Eric Lemoine e David Merceron, "Run-Time Reconfiguration of FPGA for Scanning Genomic DataBases", *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 90–98, Abril 1995.

[LvIW93] David M. Lewis, Marcus H. van Ierssel e Daniel H. Wong, "A Field Programmable Accelerator for Compiled-Code Applications", *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 60–67, Abril 1993.

[MMF+90] M. A. Matos, V. P. Miranda, J. R. Ferreira, J. C. Alves, J. N. Fidalgo, M. Isidoro e A. Azevedo, "GRADES/EXPLOR—Um sistema Gráfico Interactivo para Apoio à Gestão de Redes de Distribuição", *Actas das primeiras Jornadas Hispano-Lusas de Ingenieria Electrica*, pp. 793–800, 1990.

[Mor92] Joseph Norman Morris, Hardware design of a rapid prototyping reconfigurable logic system, Tese de Mestrado, North Carolina State University, 1992.

[MRR+53] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller e E. Teller, "Equation of state calculation by fast computing machines", *Journal of Chemistry and Physics*, vol. 21, pp. 1087–1091, 1953.

[Näh95] Stephan Näher, *The LEDA User Manual - version 3.2*, Max-Plank Institut für Informatik, 1995.

[NM93] C. L. Nikias e J. M. Mendel, "Signal Processing with Higher-Order Spectra", *IEEE Signal Processing Magazine*, vol. 10, pp. 10–37, 1993.

[NP93] Chrysostomos L. Nikias e Athina P. Petropulu, *Higher-Order Spectra Analysis—a Nonlinear Signal Processing Framework*, Prentice Hall, Signal Processing Series, Englewood Cliffs, NJ, 1993.

[OD95] John V. Oldfield e Richard C. Dorf, *Field Programmable Gate Arrays—Reconfigurable Logic for Rapid Prototyping and Implementation of Digital Systems*, John Wiley & Sons, Inc., 1995.

[PA95] A. T. Puga e A. P. Alves, "Higher-order Statistics Symmetry Properties", *Proceedings of 7th Portuguese Conference on Pattern Recognition- REC-PAD'95*, 1995.

[PC73] Robert H. Perry e Cecil H. Chilton, *Chemical Engineer's Handbook, fifth edition*, capítulo 4, Reaction Kinectics, Reactor Design and Termodynamics, McGRAW-HILL International Student Editions, Kogakusha, Ltd, Tokyo, 1973.

[PCI92] PCI Special Interest Group, *PCI Local Bus Specification (revision 2.0)*, 1992.

[PK89] Pierre G. Paulin e John P. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's", *IEEE Transactions on Computer Aided Design*, vol. 8, nº 6, pp. 661–679, Junho 1989.

[PTS93] Daniel V. Pryor, Mark R. Thistle e Nabeel Shirazi, "Text Searching on Splash 2", *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 172–177, Abril 1993.

[QKSZ94] G. M. Quénot, I. C. Kraljić, J. Sérot e B. Zavidovique, "A Reconfigurable Compute Engine for Real-Time Vision Automata Prototyping", *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 91–100, 1994.

[Raz94] Rahul Razdan, *PRISC: Programmable Reduced Instruction Set Computers*, Tese de Doutoramento, Harvard University, Cambridge, Massachusetts, 1994.

[Ree95] Colin R. Reeves, *Modern Heuristic Techniques for Combinatorial Problems*, McGRAW-HILL Book Company, 1995.

[RGSV93] Jonhatan Rose, Abbas El Gamal e Alberto Sangiovanni-Vincentelli, "Architectures of Field-Programmable Gate-Arrays", *Proceedings of the IEEE*, vol. 81, nº 7, pp. 1013–1029, Julho 1993.

[RJR95] Nalini K. Ratha, Anil K. Jain e Diane T. Rover, "Convolution on Splash 2", *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 204–213, Abril 1995.

[RKJ93] Arjun Rajagopal, Belliappa Kuttanna e Balaji Janakiraman, "A Reconfigurable Arithmetic Processor", *Proceedings of the 6th International Conference on VLSI Design*, pp. 172–175, Janeiro 1993.

[RLRP93] F. Raimbault, D. Lavenier, S. Rubini e B. Pottier, "Fine grain Parallelism on a MIMD machine using FPGAs", *Proceedings of the IEEE Workshop of FPGAs for Custom Computing Machines*, pp. 2–8, Abril 1993.

[RMJL94] Minjoong Rim, Ashutosh Mujumdar, Rajiv Jain e Renato De Leone, "Optimal and Heuristic Algoritms for Solving the Binding Problem", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 2, nº 2, pp. 211–225, Junho 1994.

[RS94] Rahul Razdan e Michael D. Smith, "A High-Performance Microarchitecture with Hardware-Programmable Functional Units", *Proceedings of the 27th Annual Workshop on Microprogramming and Microarchitecture - MICRO-27*, pp. 172–180, Novembro 1994.

[SA95] Tom Shanley e Don Anderson, *PCI System Architecture - Third Edition*, Addison-Wesley Publishing Company, 1995.

[SA96] Tom Shanley e Don Anderson, *ISA System Architecture - Third Edition*, Addison-Wesley Publishing Company, 1996.

[SjV95] Brian Schoner, Chris jones e John Villasenor, "Issues in Wireless Video Coding using Run-Time-Reconfigurable FPGAs", *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 85–89, Abril 1995.

[SL91] Harald Sihorsch e A. Laudenbach, Entwicklung eines makro-assemblers für einen vektor prozessor mit fliesskommarechenwreken, Relatório Interno, Institute für Datentechnik, FG Mikroelektronische Systeme, Technische Hochschule Darmstadt, Abril 1991.

[SM94] Haris M. Stellakis e Elias S. Manolakos, "An Array of Processors for the real-time estimation of fourth- and lower-order moments", *Signal Processing*, n º 36, pp. 341–354, 1994.

[SM95] Haris M. Stellakis e Elias S. Manolakos, "Adaptative Computation of Higher-Order Moments and its Systolic Realization", *International Journal of Adaptative Control and Signal Processing*, nº 36, pp. 341–354, 1995.

[SP89] Jong-Jiann Shieh e C. Papachristou, "On Reordering Instruction Streams for Pipelined Computers", *Proceedings of the 22nd Annual International Workshop on Microprogramming and Microarchitecture*, pp. 199–206, Agosto 1989.

[SS92] Chin-Foo See e Kewal K. Saluja, "An Efficient Method for Computation of Signatures", *Proceedings of the 5th International Conference on VLSI Design*, pp. 254–250, Janeiro 1992.

[SSAM93] Anton Shishkov, Sérgio Soares, C. Beltrán Almeida e Fernando Moreira, "SEMCI - Sistema de EMulação de Circuitos Integrados", *Actas do encontro ENDIEL 93*, 1993.

[Sta92] Richard M. Stallman, *Using and Porting GNU CC (for version 2.3)*, Free Software Foundation, 1992.

[SWSS95] Nathan Sitkoff, Mike Wazlowski, Aaron Smith e Harvey Silverman, "Implementing a Genetic Algorithm on a Parallel Custom Computing Machine", *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 180–187, Abril 1995.

[SZ90] Abdelhakim Safir e Bertrand Zavidovique, "Towards a Global Solution to High Level Synthesis Problems", *Proceedings of the 1990 European Design Automation Conference*, pp. 283–288, 1990.

[Tes90] Test Technology Technical Committee of the IEEE Computer Society, IEEE Standards Board, *IEEE Standard Test Access Port and Boundary-Scan Architecture*, Fevereiro 1990.

[TG92] Michail K. Tsatsanis e Georgios B. Giannakis, "Object and Texture Classification Using Higher-Order Statistics", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 14, nº 7, pp. 733–750, Julho 1992.

[Tur97] Ray Turner, "Using Emulation for Hardware/Software Co-Design", *Quickturn Design Systems Quarterly Newsletter*, vol. 1, nº 3, pp. 6–7, 1997.

[VBR+96] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati e P. Boucard, "Programmable Active Memories: Reconfigurable Systems Come of Age", *IEEE Transactions on VLSI Systems*, Março 1996.

[VMS97] John Villasenor e William H. Mangione-Smith, "Configurable Computing", *Scientific American*, pp. 54–59, jun 1997.

[Wal91] S. Walters, "Computer-Aided Prototyping for ASIC-Based Systems", *IEEE Design & Test of Computers*, pp. 4–10, Junho 1991.

[WAL+93] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman e S. Ghosh, "PRISM-II Compiler and Architecture", *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 9–16, Abril 1993.

[WC96] Ralph D. Wittig e Paul Chow, "OneChip: An FPGA Processor with Reconfigurable Logic", *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 126–135, Abril 1996.

[WH96a] Michael J. Wirthlin e Brad L. Hutchings, "A Dynamic Instruction Set Computer", *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 126–135, Abril 1996.

[WH96b] Michael J. Wirthlin e Brad L. Hutchings, "Sequencing Run-Time Reconfigured Hardware with Software", *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 122–128, Abril 1996.

[WHG94] Michael J. Wirthlin, Brad L. Hutchings e Kent L. Gilson, "The Nano Processor: a Low Resource Reconfigurable Processor", *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 23–30, Abril 1994.

[Wil89] Ifor Williams, "The MUSHROOM Machine – An Architecture for Symbolic Processing", *Proceedings of the IEE Colloquium on VLSI and Symbolic Processing*, Março 1989.

[Wil91] Ifor Williams, *Using FPGAs to Prototype New Computer Architectures*, pp. 373–382, Abingdom EE & CS Books, 1991.

[WMGB93] T. C. Wilson, N. Mukherjee, M. K. Garg e D. K. Banerji, "An Integrated and Accelerated ILP Solution for Scheduling, Module Allocation and Binding in Datapath Synthesis", *Proceedings of the 6th International Conference on VLSI Design*, pp. 192–197, 1993.

[WZM95] Richard W. Wieler, Zaifu Zhang e Robert McLeod, "Emulating Static Faults Using a Xilinx Based Emulator", *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 110–115, Abril 1995.

[XBL94] XILINX, *XBLOX User Guide*, 1994.

[XIL95] XILINX, *The Programmable Logic Data Book*, 1995.

Apêndice A

Programa de teste para o processador THD

Apresenta-se neste apêndice a listagem de uma codificação em linguagem C do algoritmo para cálculo da libertação de calor, usado na aplicação apresentada no capítulo 3. O código listado foi traduzido de uma representação numa linguagem de descrição de *hardware*, e foi usado como teste da configuração do GNU C construída para o processador THD.

```
#define max(a,b) ( (a) > (b) ? (a) : (b) )
#define MAXITER   128

float  Qmax, p[128], V[128], dV[128], Ki[128], Qb[128], Mrestgas, Mkraft,
            Mluft, Rende, Rstart, RM;

float  p260, V260, T260, Rg0, Lmin, lam1, Mgas, M0;

float  Su[128], r[128], A[128], B[128], TC[128], Rg[128], M[128],
          Rm[128], T[128], lam[128], lam075[128], lam080[128],
          lam093[128], n1[128], n2[128], n3[128], Tc[128], U[128],
          alp1[128], alp2[128], alp[128], dQw[128], MU[128],
          dMU[128], dQb[128], Cv[128];

inline void initialize(void);
inline void startvalue(void);
inline void iteration(void);


inline void
initialize()
{
  Rg0      =   292;
  Lmin     =   14.7;
  lam1     =   1.1;
  Mgas     = p260 * V260/(Rg0 * T260);
  Mrestgas =   1.1E5 * 0.069E-3/(Rg0 *  650);
```

```
  Mluft    = Mgas - Mrestgas;
  Mkraft   = Mluft/Lmin;
  M0       = Mluft + Mkraft + Mrestgas;
  Rende    = (lam1 -  1)/(lam1 +  1/Lmin);
  Rstart   = (Rende * Mrestgas + Mluft)/(Mrestgas + Mluft);
  RM       = Rg0 * M0;
  return;
}

inline void
startvalue()
{
  unsigned int i;

  for(i=0; i<MAXITER; i++)
  {

    T[i]  = p[i]*V[i]/RM;
    Cv[i] =  2.3961 -  5.47812E-9*p[i] +
      ( 9.273640E-4 +
      Rende*( -4.108109E-4))/ (float)2 * T[i];
    MU[i] = Cv[i] * RM * T[i];
    Qb[0] =  0;
    Qmax  =  0;
    switch(i)
    {
      case  0:
         break;
      case  1:
         dMU[0] = MU[1] - MU[0];
         dQb[0] = dMU[0] + p[0]*dV[0];
         Qmax = max( Qmax, Qb[0]);
         break;
      default: dMU[i-1] = (MU[i] - MU[i-2])/ (float)2;
         dQb[i-1] = dMU[i-1] + p[i-1]*dV[i-1];
         Qb[i-1]  = Qb[i-2] + (dQb[i-1] + dQb[i-2])/ (float)2;
         Qmax = max( Qmax, Qb[i-1]);
         if (i==MAXITER-1)
         {
           dMU[MAXITER-1] = MU[MAXITER-1] - MU[MAXITER-1 - 1];
           dQb[MAXITER-1] = dMU[MAXITER-1] +
                            p[MAXITER-1]*dV[MAXITER-1];
           Qb[MAXITER-1]  = Qb[MAXITER-1 - 1] +
                            (dQb[MAXITER-1] + dQb[MAXITER-1 - 1])/ (float)2;
           Qmax = max( Qb[MAXITER-1], Qmax );
         }
    }
  }
}


inline void
iteration()
{
  unsigned int i;
```

```
Qb[0] =  0;
for(i=0; i<MAXITER; i++)
{
  Su[i] = Qb[i]/Qmax;
  r[i]  = Su[i] * (Rende - Rstart) + Rstart;
  T[i]  = p[i]*V[i]/Rm[i];
  A[i]  =  3.4675901E-1 -  5.8277721E-2 * r[i] +
     1.1822300E-2 * r[i] *  2;
  B[i]  =  1.9100219E+1 +  1.7071225E+1 * r[i] -
     5.9537173 * r[i]* 2;
  TC[i] =  6.4785028 +  1.0208707E+1 * r[i] +
     8.2227097E-3 * T[i] -
     1.1142846E-6 * T[i] *  2 +
     3.8813399E-3 * r[i] * T[i] -
     5.0641063 * r[i]* 2;
  Rg[i] =  9.31 * ( 29 + A[i] +
    (p[i]/ 1E+7) * (B[i]/TC[i] - A[i]));
  M[i]  = Su[i] * Mkraft + Mluft + Mrestgas;
  Rm[i] = M[i] * Rg[i];
  lam[i] = ( 1 +  0.0698 * r[i])/( 1 - r[i]);
  lam075[i] =  3.2282957E-1 +
   7.2952567E-1*lam[i] -
   3.1138489E-2*lam[i]* 2 +
   1.3624943E-3*lam[i]* 3 -
   2.4439919E-5*lam[i]* 4;
  lam080[i] =  2.5963138E-1 +
   7.8609894E-1*lam[i] -
   2.8024497E-2*lam[i]* 2 +
   1.2053270E-3*lam[i]* 3 -
   2.1483051E-5*lam[i]* 4;
  lam093[i] =  9.1570749E-2 +
   9.2843110E-1*lam[i] -
   1.3044151E-2*lam[i]* 2 +
   5.3283207E-4*lam[i]* 3 -
   9.3259132E-6*lam[i]* 4;
  n1[i] =   6.7383810E+1/lam093[i];
  n2[i] =   4.8795170E-2/lam080[i];
  n3[i] =   -7.0433507E-6/lam075[i];
  Tc[i] = T[i] -  273.14;
  U[i]  = ((( -1.415931E-5 + n3[i])*Tc[i] +
      1.1280979E-1 + n2[i])*Tc[i] +
      7.110153E+3 + n1[i])*Tc[i] +
      1.970395E+5;
  alp1[i] =  8.0220847E+1 +
       7.8517971E-4*p[i] -
       6.7093635E-11*p[i]* 2 +
       6.3173436E-18*p[i]* 3 -
       2.3691233E-25*p[i]* 4;
  alp2[i] =  8.5621887 +  8.0326056E-3*T[i] -
       9.0750866E-7*T[i]* 2;
  alp[i]  = alp1[i]/alp2[i]*( 6 +  1.4)*0.8;
  dQw[i]  = alp[i] * K1[i] * (T[i] -  423)/ 12000;
  MU[i] = M[i] * U[i];
  switch(i)
```

```
    {
      case  0: break;
      case  1: dMU[0] = MU[1] - MU[0];
         dQb[0] = dMU[0] + p[0]*dV[0] + dQw[0];
         Qmax = max(Qmax, Qb[0]);
         break;
      default: dMU[i-1] = MU[i] - MU[i-2];
         dQb[i-1] = dMU[i-1] + p[i-1]*dV[i-1] + dQw[i-1];
         Qb[i-1]  = Qb[i-2] + (dQb[i-1] + dQb[i-2])/ 2;
         Qmax = max(Qb[i-1], Qmax);
         if (i==MAXITER-1)
         {
           dMU[MAXITER-1] = MU[MAXITER-1] - MU[MAXITER-1 - 1];
           dQb[MAXITER-1] = dMU[MAXITER-1] + p[MAXITER-1]*dV[MAXITER-1] +
                            dQw[MAXITER-1];
           Qb[MAXITER-1]  = Qb[MAXITER-1 - 1] +
                            (dQb[MAXITER-1] + dQb[MAXITER-1 - 1])/ 2;
           Qmax = max( Qmax, Qb[MAXITER-1]);
         }
    }
  }
}


void _main()
{
  initialize();
  startvalue();
  iteration();
  iteration();
}
```

Apêndice B

Configuração do GNU C para o processador THD

Apresenta-se neste apêndice o ficheiro de configuração que foi construído para configurar o GNU C para o processador THD, onde são representados os modelos das instruções suportadas pelo processador destino e certas regras de optimização (ficheiro de descrição da máquina, `procthd.md`). O ficheiro de definição de *macros* não é incluído nesta listagem devido à sua extensão e por ter sido obtido à custa de um número reduzido de modificações num dos ficheiros originais da distribuição do `gcc`, `we32K.h`. A configuração construída invoca funções que residem num ficheiro em C adicional, que são responsáveis pela formatação do código *assembly* produzido e pela emissão de mensagens de erro quando são encontradas instruções não suportadas pelo processador.

```
; gcc configuration for THD vector processor (procthd.md)
;
; J.C.Alves, Darmstadt, July 1993
;
;-----------------------------------------------------------------
; Invalid machine modes (other that SF or DF) generate an error message
; via function Invalidmachmode(). The definitions of operation id
; are in file invopcode.h, included by procthd.c
;
; Invalid opcodes (not supported by the processor) generate an error message
; calling function InvalidOpcode().
;
; data movement instructions:
;
(define_insn "movsi"
  [(set (match_operand:SI 0 "general_operand"  "")
        (match_operand:SI 1 "general_operand"  ""))]
  ""
  "*
  { Invalidmachmode(OP_MOV);
```

```
    return \"\";
  }")
;
(define_insn "movdf"
  [(set (match_operand:DF 0 "general_operand"  "")
        (match_operand:DF 1 "general_operand"  ""))]
  ""
  "*
  { output_mov( operands , insn);
    return \"\";
  }")
;
(define_insn "movsf"
  [(set (match_operand:SF 0 "general_operand"  "")
        (match_operand:SF 1 "general_operand"  ""))]
  ""
  "*
  { output_mov( operands , insn);
    return \"\";
  }")
;
;-----------------------------------------------------------------------
; Single arithmetic instructions:
;
; Single integer machine mode: this is invalid for THD
; processor, so an error message will be emited and no assembly
; code will be produced
;
(define_insn "addsi3"
  [(set (match_operand:SI 0 "nonimmediate_operand" "")
        (plus:SI (match_operand:SI 1 "general_operand" "")
                 (match_operand:SI 2 "general_operand" "")
        )
   )]
  ""
  "*
  { Invalidmachmode(OP_ADD);
    return \"\";
  }")
;
(define_insn "subsi3"
  [(set (match_operand:SI 0 "nonimmediate_operand" "")
        (minus:SI (match_operand:SI 1 "general_operand" "")
                  (match_operand:SI 2 "general_operand" "")))]
  ""
  "*
  { Invalidmachmode(OP_SUB);
    return \"\";
  }")
;
(define_insn "mulsi3"
  [(set (match_operand:SI 0 "nonimmediate_operand" "")
        (mult:SI (match_operand:SI 1 "general_operand" "")
                 (match_operand:SI 2 "general_operand" "")))]
  ""
```

```
  "*
  { Invalidmachmode(OP_MUL);
    return \"\";
  }")
;
(define_insn "divsi3"
  [(set (match_operand:SI 0 "nonimmediate_operand" "")
        (div:SI (match_operand:SI 1 "general_operand" "")
                (match_operand:SI 2 "general_operand" "")))]
  ""
  "*
  { Invalidmachmode(OP_DIV);
    return \"\";
  }")
;
 (define_insn "smaxsi3"
   [(set (match_operand:SI 0 "nonimmediate_operand" "")
         (smax:SI (match_operand:SI 1 "general_operand" "")
                  (match_operand:SI 2 "general_operand" "")))]
    ""
    "*
    { Invalidmachmode(OP_MAX);
      return \"\";
    }")
;
;-------------------------------------------------------------------
; Single float and double float machine mode
;
(define_insn "addsf3"
  [(set (match_operand:SF 0 "nonimmediate_operand" "")
        (plus:SF (match_operand:SF 1 "general_operand" "")
                 (match_operand:SF 2 "general_operand" "")
        )
   )]
  ""
  "*
  { output_add( operands , insn);
    return \"\";
  }")
;
(define_insn "subsf3"
  [(set (match_operand:SF 0 "nonimmediate_operand" "")
        (minus:SF (match_operand:SF 1 "general_operand" "")
                  (match_operand:SF 2 "general_operand" "")))]
  ""
  "*
  { output_sub( operands , insn);
    return \"\";
  }")
;
(define_insn "mulsf3"
  [(set (match_operand:SF 0 "nonimmediate_operand" "")
        (mult:SF (match_operand:SF 1 "general_operand" "")
                 (match_operand:SF 2 "general_operand" "")))]
  ""
```

```
  "*
  { output_mul(operands, insn);
    return \"\";
  }")
;
(define_insn "divsf3"
  [(set (match_operand:SF 0 "nonimmediate_operand" "")
        (div:SF (match_operand:SF 1 "general_operand" "")
                (match_operand:SF 2 "general_operand" "")))]
  ""
  "*
  { output_div(operands, insn);
    return \"\";
  }")
;
 (define_insn "smaxsf3"
   [(set (match_operand:SF 0 "nonimmediate_operand" "")
         (smax:SF (match_operand:SF 1 "general_operand" "")
                  (match_operand:SF 2 "general_operand" "")))]
    ""
    "*
    { output_max(operands, insn);
      return \"\";
    }")
;
(define_insn "adddf3"
  [(set (match_operand:DF 0 "nonimmediate_operand" "")
        (plus:DF (match_operand:DF 1 "general_operand" "")
                 (match_operand:DF 2 "general_operand" "")
        )
   )]
  ""
  "*
  { output_add( operands , insn);
    return \"\";
  }")
;
(define_insn "subdf3"
  [(set (match_operand:DF 0 "nonimmediate_operand" "")
        (minus:DF (match_operand:DF 1 "general_operand" "")
                  (match_operand:DF 2 "general_operand" "")))]
  ""
  "*
  { output_sub( operands , insn);
    return \"\";
  }")
;
(define_insn "muldf3"
  [(set (match_operand:DF 0 "nonimmediate_operand" "")
        (mult:DF (match_operand:DF 1 "general_operand" "")
                 (match_operand:DF 2 "general_operand" "")))]
  ""
  "*
  { output_mul(operands, insn);
    return \"\";
```

```
  }")
;
(define_insn "divdf3"
  [(set (match_operand:DF 0 "nonimmediate_operand" "")
        (div:DF (match_operand:DF 1 "general_operand" "")
                (match_operand:DF 2 "general_operand" "")))]
  ""
  "*
  { output_div(operands, insn);
    return \"\";
  }")
;
 (define_insn "smaxdf3"
   [(set (match_operand:DF 0 "nonimmediate_operand" "")
         (smax:DF (match_operand:DF 1 "general_operand" "")
                  (match_operand:DF 2 "general_operand" "")))]
    ""
    "*
    { output_max(operands, insn);
      return \"\";
    }")
;
;----------------------------------------------------------------------
; insn patterns for instruction combination:
;
(define_insn ""
  [(set (match_operand:SF 0 "general_operand" "")
        (plus:SF (match_operand:SF 3 "general_operand" "")
                 (mult:SF (match_operand:SF 1 "general_operand" "")
                          (match_operand:SF 2 "general_operand" "")
                 )
        )
   )
  ]
  ""
  "*
  { output_madd(operands, insn);
    return \"\";
  }")
;
(define_insn ""
  [(set (match_operand:SF 0 "general_operand" "")
        (plus:SF (mult:SF (match_operand:SF 1 "general_operand" "")
                          (match_operand:SF 2 "general_operand" "")
                 )
                 (match_operand:SF 3 "general_operand" "")
        )
   )
  ]
  ""
  "*
  { output_madd(operands, insn);
    return \"\";
  }")
;
```

```
(define_insn ""
  [(set (match_operand:SF 0 "general_operand" "")
        (minus:SF (match_operand:SF 3 "general_operand" "")
                  (mult:SF (match_operand:SF 1 "general_operand" "")
                           (match_operand:SF 2 "general_operand" "")
                  )
        )
   )
  ]
  ""
  "*
  { output_msub(operands, insn);
    return \"\";
  }")
;
(define_insn ""
  [(set (match_operand:SF 0 "general_operand" "")
        (minus:SF (mult:SF (match_operand:SF 1 "general_operand" "")
                           (match_operand:SF 2 "general_operand" "")
                  )
                  (match_operand:SF 3 "general_operand" "")
        )
   )
  ]
  ""
  "*
  { output_msub(operands, insn);
    return \"\";
  }")
;
(define_insn ""
  [(set (match_operand:SF 0 "general_operand" "")
        (plus:SF (match_operand:SF 3 "general_operand" "")
                 (div:SF (match_operand:SF 1 "general_operand" "")
                         (match_operand:SF 2 "general_operand" "")
                 )
        )
   )
  ]
  ""
  "*
  { output_dadd(operands, insn);
    return \"\";
  }")
;
(define_insn ""
  [(set (match_operand:SF 0 "general_operand" "")
        (plus:SF (div:SF (match_operand:SF 1 "general_operand" "")
                         (match_operand:SF 2 "general_operand" "")
                 )
                 (match_operand:SF 3 "general_operand" "")
        )
   )
  ]
  ""
```

```
  "*
  { output_dadd(operands, insn);
    return \"\";
  }")
;
(define_insn ""
  [(set (match_operand:SF 0 "general_operand" "")
        (minus:SF (match_operand:SF 3 "general_operand" "")
                  (div:SF (match_operand:SF 1 "general_operand" "")
                          (match_operand:SF 2 "general_operand" "")
                  )
        )
   )
  ]
  ""
  "*
  { output_dsub(operands, insn);
    return \"\";
  }")
;
(define_insn ""
  [(set (match_operand:SF 0 "general_operand" "")
        (minus:SF (div:SF (match_operand:SF 1 "general_operand" "")
                          (match_operand:SF 2 "general_operand" "")
                  )
                  (match_operand:SF 3 "general_operand" "")
        )
   )
  ]
  ""
  "*
  { output_dsub(operands, insn);
    return \"\";
  }")
;
;-------------------------------------------------------------------
; The following are not supported by THD processor, but
; they must be defined in any configuration
;
;-------------------------------------------------------------------
; Compare and test:
;
(define_insn "tstsi"
  [(set (cc0) (match_operand:SI 0 "nonimmediate_operand" "mri"))]
  ""
  "*
  {
  InvalidOpcode(OP_TST);
  return \"\";
  } ")
;
(define_insn "cmpsi"
  [(set (cc0) (compare (match_operand:SI 0 "general_operand" "mri")
                       (match_operand:SI 1 "general_operand" "mri")))]
  ""
```

```
  "*
  {
  InvalidOpcode(OP_CMP);
  return \"\";
  } ")
;
(define_insn "cmpsf"
  [(set (cc0) (compare (match_operand:SF 0 "general_operand" "")
                       (match_operand:SF 1 "general_operand" "")))]
  ""
  "*
  {
  InvalidOpcode(OP_CMP);
  return \"\";
  } ")
;
;-----------------------------------------------------------------
; Branch instructions (conditional and unconditional):
;
(define_insn "beq"
  [(set (pc) (if_then_else (eq (cc0) (const_int 0))
                           (label_ref (match_operand 0 "" ""))
                           (pc)))]
  ""
  "*
  {
  InvalidOpcode(OP_COND_JMP);
  return \"\";
  } ")
;
(define_insn "bne"
  [(set (pc) (if_then_else (ne (cc0) (const_int 0))
                           (label_ref (match_operand 0 "" ""))
                           (pc)))]
  ""
  "*
  {
  InvalidOpcode(OP_COND_JMP);
  return \"\";
  } ")
;
(define_insn "bgt"
  [(set (pc) (if_then_else (gt (cc0) (const_int 0))
                           (label_ref (match_operand 0 "" ""))
                           (pc)))]
  ""
  "*
  {
  InvalidOpcode(OP_COND_JMP);
  return \"\";
  } ")
;
(define_insn "bgtu"
  [(set (pc) (if_then_else (gtu (cc0) (const_int 0))
                           (label_ref (match_operand 0 "" ""))
```

```
                          (pc)))]
  ""
  "*
  {
  InvalidOpcode(OP_COND_JMP);
  return \"\";
  } ")
;
(define_insn "blt"
  [(set (pc) (if_then_else (lt (cc0) (const_int 0))
                           (label_ref (match_operand 0 "" ""))
                           (pc)))]
  ""
  "*
  {
  InvalidOpcode(OP_COND_JMP);
  return \"\";
  } ")
;
(define_insn "bltu"
  [(set (pc) (if_then_else (ltu (cc0) (const_int 0))
                           (label_ref (match_operand 0 "" ""))
                           (pc)))]
  ""
  "*
  {
  InvalidOpcode(OP_COND_JMP);
  return \"\";
  } ")
;
(define_insn "bge"
  [(set (pc) (if_then_else (ge (cc0) (const_int 0))
                           (label_ref (match_operand 0 "" ""))
                           (pc)))]
  ""
  "*
  {
  InvalidOpcode(OP_COND_JMP);
  return \"\";
  } ")
;
(define_insn "bgeu"
  [(set (pc) (if_then_else (geu (cc0) (const_int 0))
                           (label_ref (match_operand 0 "" ""))
                           (pc)))]
  ""
  "*
  {
  InvalidOpcode(OP_COND_JMP);
  return \"\";
  } ")
;
(define_insn "ble"
  [(set (pc) (if_then_else (le (cc0) (const_int 0))
                           (label_ref (match_operand 0 "" ""))
```

```
                          (pc)))]
  ""
  "*
  {
  InvalidOpcode(OP_COND_JMP);
  return \"\";
  } ")
;
(define_insn "bleu"
  [(set (pc) (if_then_else (leu (cc0) (const_int 0))
                           (label_ref (match_operand 0 "" ""))
                           (pc)))]
  ""
  "*
  {
  InvalidOpcode(OP_COND_JMP);
  return \"\";
  } ")
;
(define_insn "indirect_jump"
  [(set (pc) (match_operand:SI 0 "address_operand" "p"))]
  ""
  "*
  {
  InvalidOpcode(OP_JMP);
  return \"\";
  } ")
;
(define_insn "jump"
  [(set (pc) (label_ref (match_operand 0 "" "")))]
  ""
  "*
  {
  InvalidOpcode(OP_JMP);
  return \"\";
  } ")
;
;-----------------------------------------------------------------------
; call instruction:
;
(define_insn "call"
  [(call (match_operand:QI 0 "memory_operand" "m")
         (match_operand:SI 1 "immediate_operand" "i"))]
  ""
  "*
  {
  InvalidOpcode(OP_CALL);
  return \"\";
  } ")
;
;-----------------------------------------------------------------------
; No-op instruction:
;
(define_insn "nop"
  [(const_int 0)]
```

```
""
"*
{
InvalidOpcode(OP_NOP);
return \"\";
} ")
```

Apêndice C

Biblioteca de unidades funcionais

Neste apêndice é listado o ficheiro que descreve a biblioteca de unidades funcionais considerada para os problemas de teste ensaiados com a aplicação de optimização apresentada no capítulo 5. A matriz de custos de reconfiguração, listada no fim da definição de cada unidade funcional, é apenas usada quando a aplicação considera unidades funcionais reconfiguráveis dinamicamente, i.e., requerendo tempo para trocar entre os vários operadores que pode implementar. No caso contrário, a matriz de custos de reconfiguração é ignorada e todos esses custos são considerados nulos.

```
#####################################################################
# Para cada unidade funcional:
# - nome, custo area, numero de configuracoes
#   para cada configuracao:
#    - nome, caracter_id, duracao, latencia, numero de operandos
#    - para cada operando:
#       ciclo de consumo
#    - ciclo de producao resultado
# - matriz dos custos de reconfiguacao
#####################################################################
# Para as operacoes MOV e EXIT:
#####################################################################
EXITMOV 0 2
# Lista de configuracoes:
EXIT . 1 0 0
0
0
MOV  = 1 1 1
0
0
# Custos de reconfiguracao:
0 0
0 0
#####################################################################
# ALU generica com custo area 600 e 10 configuracoes:
# ADD, SUB nao-pipelined, MUL, DIV dois estagios pipeline.
#####################################################################
```

```
ALU  600  10
# Lista de configuracoes:
ADD  + 2 2 2
0   0
1
SUB - 3 3 2
0   1
2
MUL * 8 4 2
0    0
7
DIV / 10 5 2
0    0
9
MULADD m 10 4 3
0   0   8
9
MULSUB s 11 4 3
0   0   8
10
DIVADD d 12 5 3
0   0   10
11
DIVSUB u 13 5 3
0   0   10
12
MAX X 3 3 2
0   1
2
MIN x 3 3 2
0   1
2
# Custos de reconfiguracao:
 0  5  8 10  8 12 10 14  5  5
 5  0  8 10 12  8 12  8  3  3
 8  8  0  8  5  8 10 12  5  5
12  8  8  0 12 14  5  6  5  5
10 12  5 12  0  5 10 12  5  5
12 10  5 12  5  0 12 10  4  4
 8 10 10  6  0  5  0 12  5  5
10  8 13  6 12 10  6  0  4  4
 5  5  8 10 10 12 12 14  0  2
 5  5  8 10 12 10 12 14  2  0
#####################################################################
# ALU generica com custo area 400 e 10 configuracoes:
# ADD e SUB pipelined, MUL e DIV lentos
#####################################################################
GENERICALU  400  10
# Lista de configuracoes:
ADD  + 1 1 2
0    0
0
SUB - 2 2 2
0    0
1
```

```
MUL * 16 16 2
0    0
15
DIV / 16 16 2
0    0
15
MULADD m 18 16 3
0   0   16
17
MULSUB s 18 16 3
0   0  16
17
DIVADD d 18 16 3
0   0   16
17
DIVSUB u 18 16 3
0   0   16
17
MAX X 3 3 2
0   0
2
MIN x 3 3 2
0   0
2
# Custos de reconfiguracao:
 0  3  5  7  5  8  6  9  3  3
 3  0  5  7  8  5  9  5  3  3
 5  5  0  5  4  6  8  9  3  3
 8  6  6  0  8 10  3  4  3  3
 8  9  4  9  0  4  8  9  3  3
 8  7  3  8  3  0  9  8  2  2
 6  9  7  4  8  4  0  8  5  5
 8  6  9  4  8  8  4  0  3  3
 4  4  6  8  8  9  9 10  0  2
 4  4  6  8  9  8  9 10  2  0

#####################################################################
# MULPIPE: multiplicador e somador/subtractor, 1 ciclo de latencia
#####################################################################
MULPIPE 400  7
# Lista de configuracoes:
ADD  + 2 1 2
0    0
1
SUB - 2 1 2
0    0
1
MUL * 10 1 2
0    0
9
MULADD m 12 1 3
0   0   10
11
MULSUB s 12 1 3
0   0   10
```

```
11
MAX X 2 1 2
0   0
1
MIN x 2 1 2
0   0
1
# Custos de reconfiguracao:
 0  7  9  9 14  3  3
 7  0  8 12  8  3  3
 9  8  0  5  8  5  5
 9 12  5  0  5  5  5
14  9  5  5  0  4  4
 4  5  8 12 13  0  2
 3  5  8 14 12  2  0
#####################################################################
# MULDIVPIPE: multiplicador e divisor rapidos e pipelined,
#             tempos de reconfiguracao longos
#####################################################################
MULDIVPIPE 500  3
# Lista de configuracoes:
ADD +  1 1 2
0    0
0
MUL *  7 1 2
0    0
6
DIV /  9 1 2
0    0
8
# Custos de reconfiguracao:
 0 15 16
15  0 12
16 12  0
#####################################################################
# ADDSUBMUL: somador e subtractor pipelined, multiplicador lento
#            tempos de reconfiguracao curtos
#####################################################################
ADDSUBMUL 300  5
# Lista de configuracoes:
ADD  + 3 1 2
0    0
2
SUB - 3 1 2
0    0
2
MUL * 16 16 2
0    0
15
MAX X 3 1 2
0   0
2
MIN x 3 1 2
0   0
2
```

```
# Custos de reconfiguracao:
 0  2  8  3  3
 2  0  8  3  3
 9  9  0  8  5
 3  3  8  0  1
 3  3  8  1  0
```

Apêndice D

Problemas ensaiados com a aplicação desenvolvida no capítulo 5

Neste apêndice são listados os problemas exemplo que foram ensaiados com a aplicação de optimização baseada no algoritmo *Simulated Annealing*, apresentada no capítulo 5. Os três primeiros exemplos foram escritos directamente na linguagem *assembly* considerada para especificar os programas a optimizar. O quarto é um segmento do código obtido pela compilação do programa listado no apêndice A, usando o compilador GNU C que foi construído para o processador THD.

D.1 HAL

Este é um problema de teste geralmente usado em aplicações de síntese de alto nível de circuitos digitais, originalmente apresentado em [PK89]. O número reduzido de operações (9 operações, 21 ramos entre os nós do grafo de dependências de dados) conduz a tempos curtos de execução para a técnica de optimização implementada, o que permitiu experimentar variadas combinações dos parâmetros de controlo do algoritmo *Simulated Annealing* de forma a afiná-los para a aplicação desenvolvida.

```
MUL U2,FIVE,X
MUL U3,THREE,Y
MUL Y1,U,DX
ADD X,X,DX
MUL U4,Y1,U2
MUL U5,DX,U3
ADD Y,Y,Y1
SUB U6,U,U4
SUB U,U6,U5
```

D.2 ellip

Como o anterior, este é também um exemplo corrente usado em aplicações de síntese de alto nível de circuitos digitais. A sequência de operações implementa um filtro digital elíptico de 5ª ordem [KWK85]. O grafo de dependências de dados que representa este problema tem 34 nós (número de operações) e 80 ramos.

```
ADD A,I,T2
ADD B,A,T13
ADD X1,B,T26
ADD G,T33,T39
ADD E,G,X1
MUL X2,M21,E
MUL X3,M24,E
ADD D,X2,B
ADD F,X3,G
ADD X4,B,D
ADD X9,F,D
MUL X6,M9,X4
ADD X5,F,G
ADD T26,X9,E
MUL X7,M30,X5
ADD C,A,X6
ADD X12,C,D
ADD H,T39,X7
ADD X8,A,C
ADD X10,F,H
MUL X11,M6,X8
ADD J,T18,X12
ADD X13,H,T39
MUL X16,M16,J
MUL O,M40,X13
ADD K,T38,X10
ADD X14,I,X11
MUL X15,M36,K
ADD T18,T18,X16
ADD T39,O,H
ADD T2,X14,C
ADD T38,T38,X15
ADD T13,T18,J
ADD T33,T38,K
```

D.3 inter

Este programa foi escrito directamente na linguagem *assembly* considerada para o processador reconfigurável apresentado no capítulo 4 e calcula nas variáveis `xi` e `yi` as coordenadas do ponto de intersecção das rectas definidas pelos pares de pontos `(Xa,Ya)`, `(Xb,Yb)` e `(X0,Y0)`, `(X1,Y1)`. Esta operação é intensivamente usada em aplicações de optimização de empacotamentos de peças representadas por polígonos irregulares, para determinar sobreposições entre eles. O problema tem 19 operações e o GDD que o representa tem 47 ramos.

```
MUL R0, Y0, X1
MUL R1, X0, Y1
SUB R0, R0, R1
SUB R2, X1, X0
DIV R3, R0, R2
MUL R0, Ya, Xb
MUL R1, Xa, Yb
SUB R0, R0, R1
SUB R4, Xb, Xa
DIV R5, R0, R4
SUB R0, Y1, Y0
SUB R1, Yb, Ya
SUB R3, R5, R3
DIV R0, R0, R2
DIV R1, R1, R4
SUB R0, R0, R1
DIV xi, R3, R0
MUL R1, R1, xi
ADD yi, R1, R5
```

D.4 hd1l

Este segmento de código foi obtido com a configuração do gcc apresentada no capítulo 3, e é o resultado da compilação de uma iteração do ciclo `for(...)` para `i=2`, da função `iteration()` do programa listado no apêndice A. O problema tem 106 operações e o grafo de dependências de dados 294 ramos.

```
; i = 2
DIV R8, Qb+8, Qmax
MOV Su+8, R8
SUB R0, Rende, Rstart
MADD R3, R0, R8, Rstart
MOV r+8, R3
MOV R9, p+8
MUL R0, R9, V+8
DIV R6, R0, Rm+8
MOV T+8, R6
MOV R0, R3
MSUB R1, R0, #0.0582777, #0.346759
MADD R7, R0, #0.0236446, R1
MOV A+8, R7
MADD R1, R0, #17.0712, #19.1002
MSUB R5, R0, #11.9074, R1
MOV B+8, R5
MADD R0, R3, #10.2087, #6.4785
MADD R0, R6, #0.00822271, R0
MSUB R1, R6, #2.22857E-06, R0
MUL R0, R3, #0.00388134
MADD R0, R0, R6, R1
MSUB R2, R3, #10.1282, R0
MOV TC+8, R2
MOV R0, R7
ADD R4, R0, #29
DIV R1, R9, #1E+07
DSUB R0, R5, R2, R0
MADD R0, R1, R0, R4
MUL R1, R0, #9.31
MOV Rg+8, R1
MADD R0, R8, Mkraft, Mluft
ADD R7, R0, Mrestgas
MOV M+8, R7
MUL Rm+8, R7, R1
MOV R0, R3
MADD R1, R0, #0.0698, #1
SUB R0, #1, R0
DIV R1, R1, R0
MOV lam+8, R1
MADD R0, R1, #0.729526, #0.32283
MSUB R2, R1, #0.062277, R0
MUL R0, R1, #0.00136249
MADD R2, R0, #3, R2
MUL R0, R1, #2.44399E-05
MSUB R5, R0, #4, R2
MOV lam075+8, R5
MADD R0, R1, #0.786099, #0.259631
```

```
MSUB R2, R1, #0.056049, R0
MUL R0, R1, #0.00120533
MADD R2, R0, #3, R2
MUL R0, R1, #2.14831E-05
MSUB R3, R0, #4, R2
MOV lam080+8, R3
MADD R0, R1, #0.928431, #0.0915707
MSUB R2, R1, #0.0260883, R0
MUL R0, R1, #0.000532832
MADD R2, R0, #3, R2
MUL R0, R1, #9.32591E-06
MSUB R0, R0, #4, R2
MOV lam093+8, R0
DIV R4, #67.3838, R0
MOV n1+8, R4
DIV R2, #0.0487952, R3
MOV n2+8, R2
DIV R0, #-7.04335E-06, R5
MOV n3+8, R0
MOV R3, R6
SUB R1, R3, #273.14
MOV Tc+8, R1
ADD R0, R0, #-1.41593E-05
MADD R0, R0, R1, #0.11281
ADD R0, R0, R2
MADD R0, R0, R1, #7110.15
ADD R0, R0, R4
MADD R4, R0, R1, #197040
MOV U+8, R4
MOV R1, R9
MADD R0, R1, #0.00078518, #80.2208
MSUB R2, R1, #1.34187E-10, R0
MUL R0, R1, #6.31734E-18
MADD R2, R0, #3, R2
MUL R0, R1, #2.36912E-25
MSUB R1, R0, #4, R2
MOV alp1+8, R1
MOV R2, R3
MADD R0, R2, #0.00803261, #8.56219
MSUB R0, R2, #1.81502E-06, R0
MOV alp2+8, R0
DIV R0, R1, R0
MUL R0, R0, #7.4
MUL R0, R0, #0.8
MOV alp+8, R0
MUL R1, R0, K1+8
SUB R0, R2, #423
MUL R0, R1, R0
DIV dQw+8, R0, #12000
MUL R0, R7, R4
MOV MU+8, R0
; case: i == 2
SUB R0, R0, MU
MOV dMU+4, R0
MADD R0, p+4, dV+4, R0
```

```
ADD R0, R0, dQw+4
MOV dQb+4, R0
ADD R0, R0, dQb
DADD Qb+4, R0, #2, Qb
MAX Qmax, Qb+4, Qmax
```

Apêndice E

Implementação do ProHos no CVR

Apresentam-se neste apêndice os esquemas das partes fundamentais da implementação do processador dedicado ProHos no sistema reconfigurável CVR, apresentados no capítulo 6. É apresentada a implementação realizada para a unidade de cálculo e para a unidade geradora de endereços para as memórias *cache*.

E.1 Unidade de cálculo

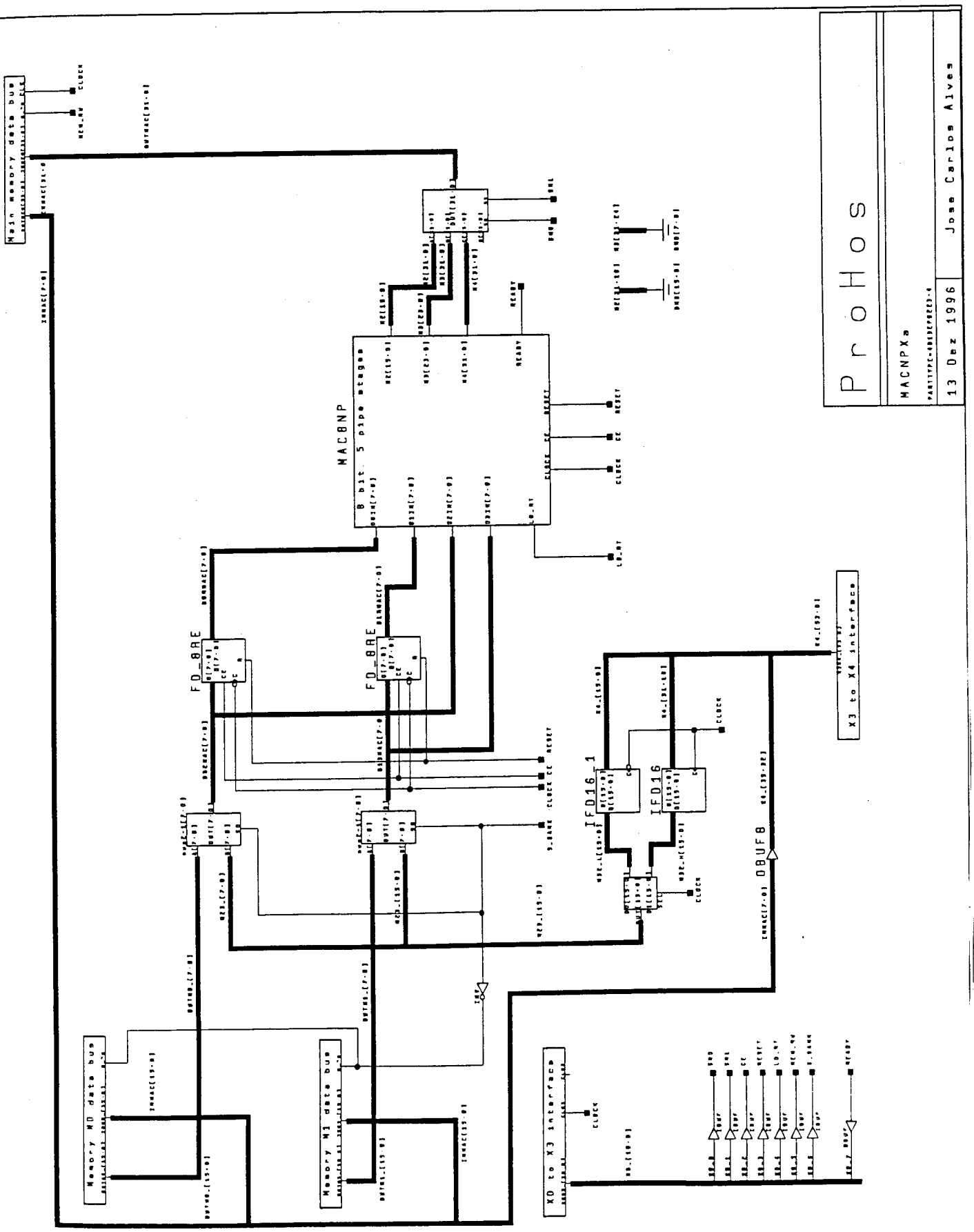
A unidade de cálculo apresentada foi implementada no FPGA X3 do CVR e inclui os circuitos fundamentais que são listados nas páginas seguintes, de forma descendente da hierarquia do circuito construído:

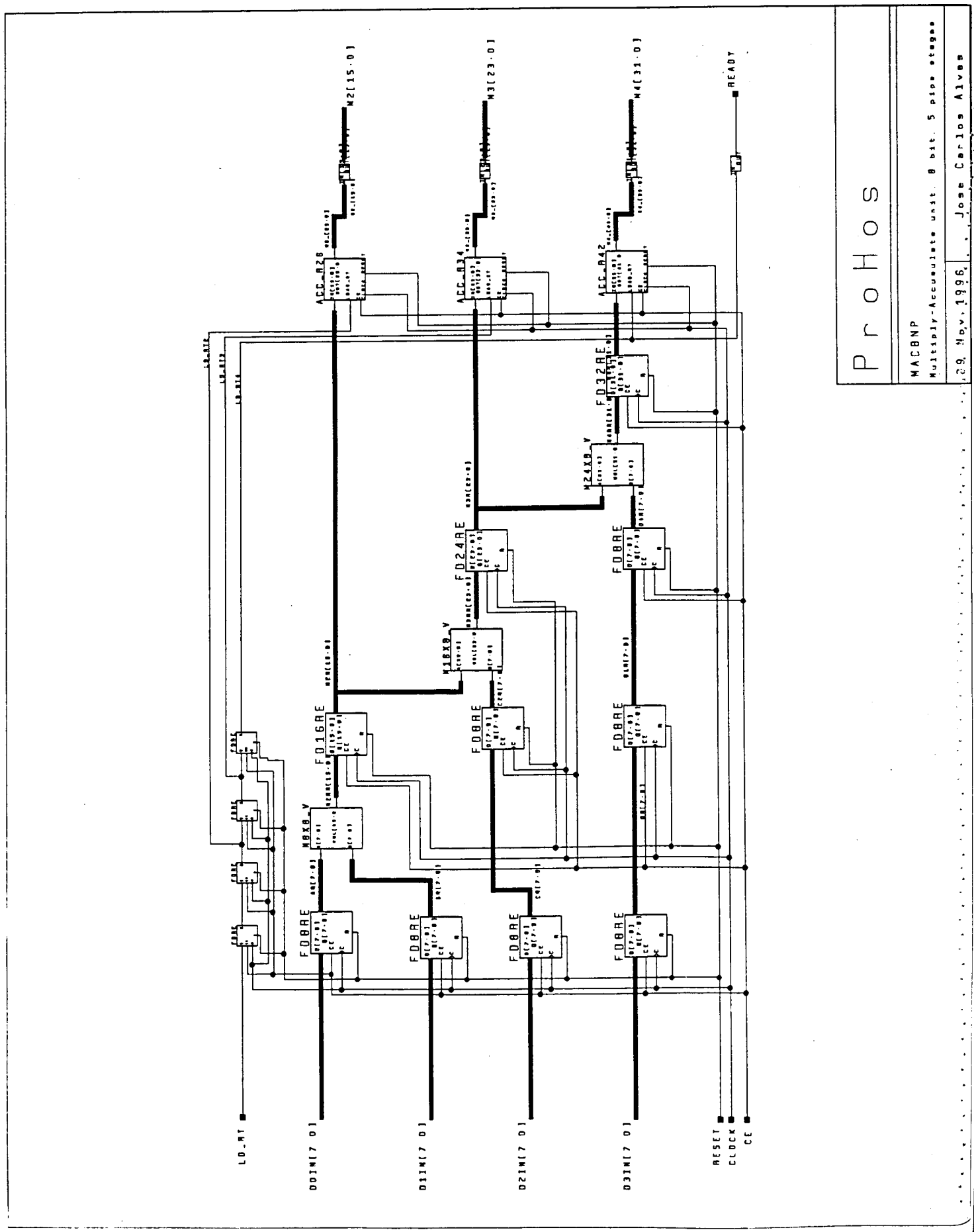
- `MACNPXA` é o circuito implementado no FPGA X3, que inclui a unidade de cálculo propriamente dita, `MAC8NP`.

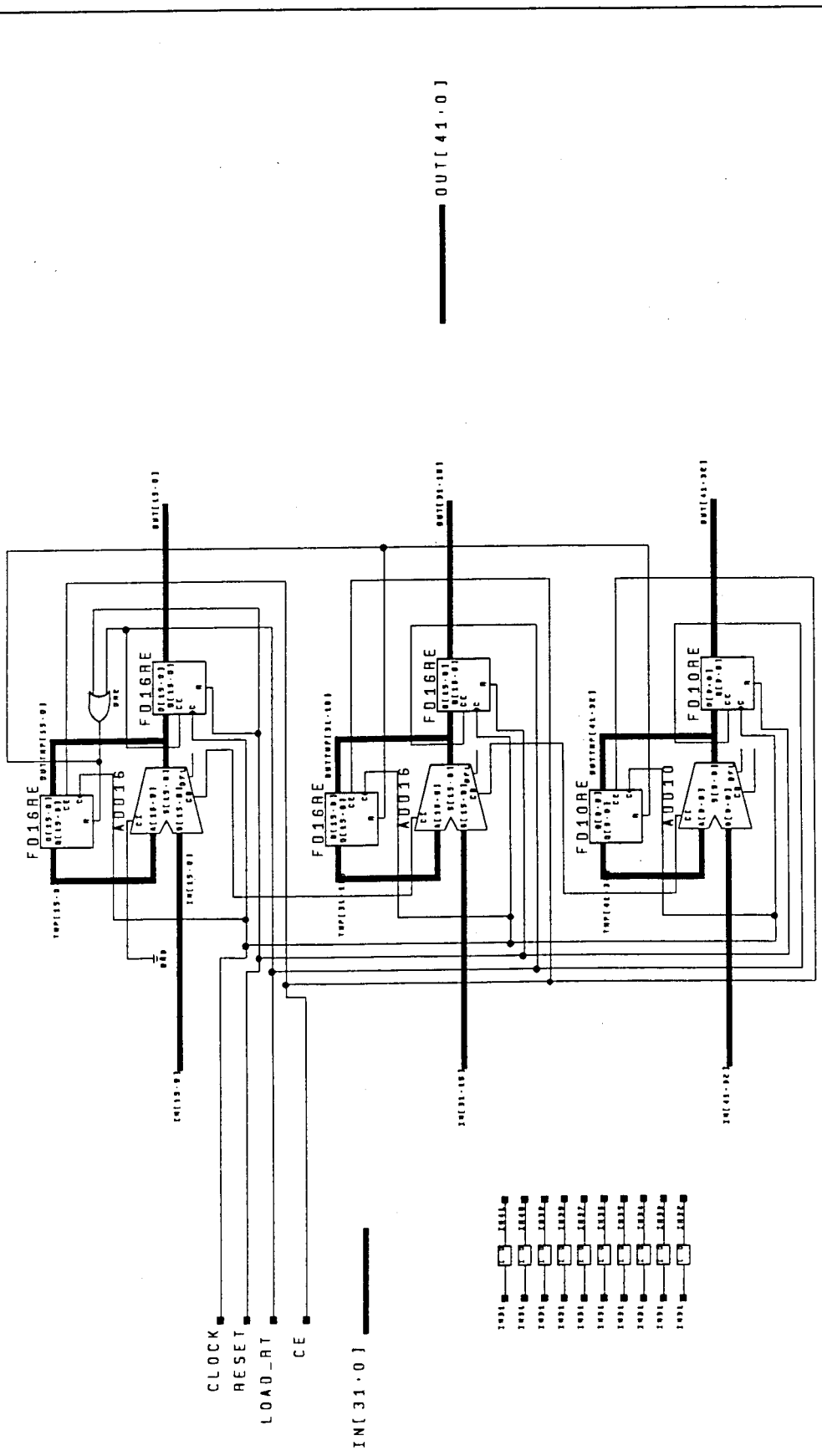
- Os elementos fundamentais da unidade de cálculo (`MAC8NP`) são os 3 multiplicadores e os 3 acumuladores existentes na saída de cada multiplicador. Os acumuladores correspondem aos símbolos `ACC_R26`, `ACC_R34` e `ACC_R42`, e os multiplicadores são os símbolos `M8X8_V`, `M8x16_V` e `M8X24_V`. Destes 6 circuitos apresentam-se apenas os dois mais complexos de cada tipo, `M8X24_V` e `ACC_R42`.

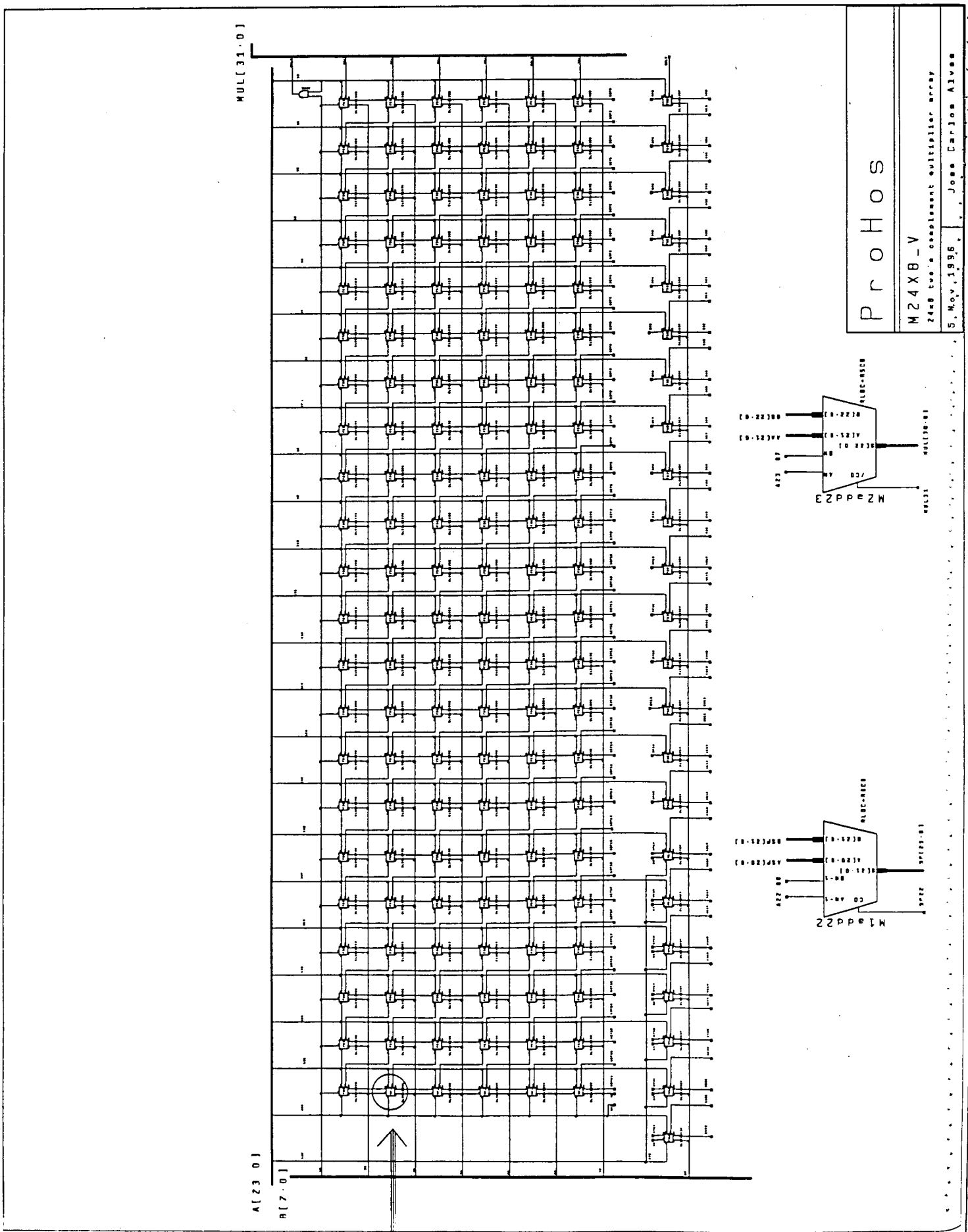
- Os multiplicadores foram desenvolvidos como estruturas regulares de células semelhantes, que foram optimizadas para usar eficazmente os recursos programáveis da família de FPGA usada neste desenvolvimento. A título de exemplo, apresenta-se a implementação da célula `FA7_3`, assinalada com uma seta no esquema lógico do multiplicador `M8X24_V`.

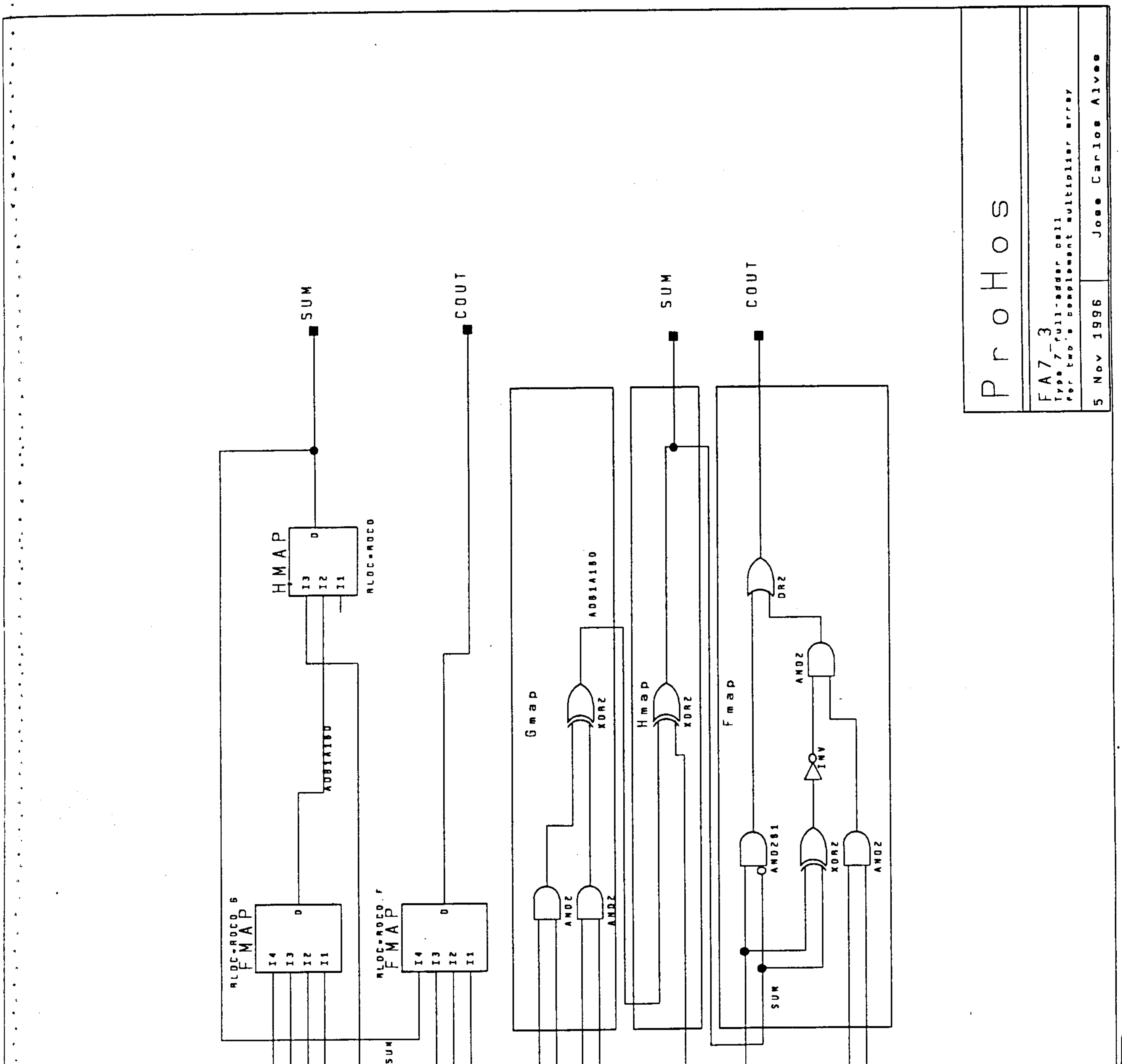
- Os dois símbolos existentes na parte inferior do esquema lógico do multiplicador são circuitos somadores que usam recursos que os FPGAs contêm, dedicados à construção de cadeias eficientes para propagação do *bit* de transporte em circuitos somadores. Estes circuitos somadores foram modificados de outros existentes na biblioteca de componentes da XILINX, de forma a trabalharem sobre operandos de dimensão adequada às necessidades de cada multiplicador, incluindo também algumas funções lógicas adicionais. Como a estrutura dos 2 circuitos deste tipo é semelhante, é apresentado apenas o circuito lógico do somador `M1ADD22` usado no multiplicador `M8X24_V`.

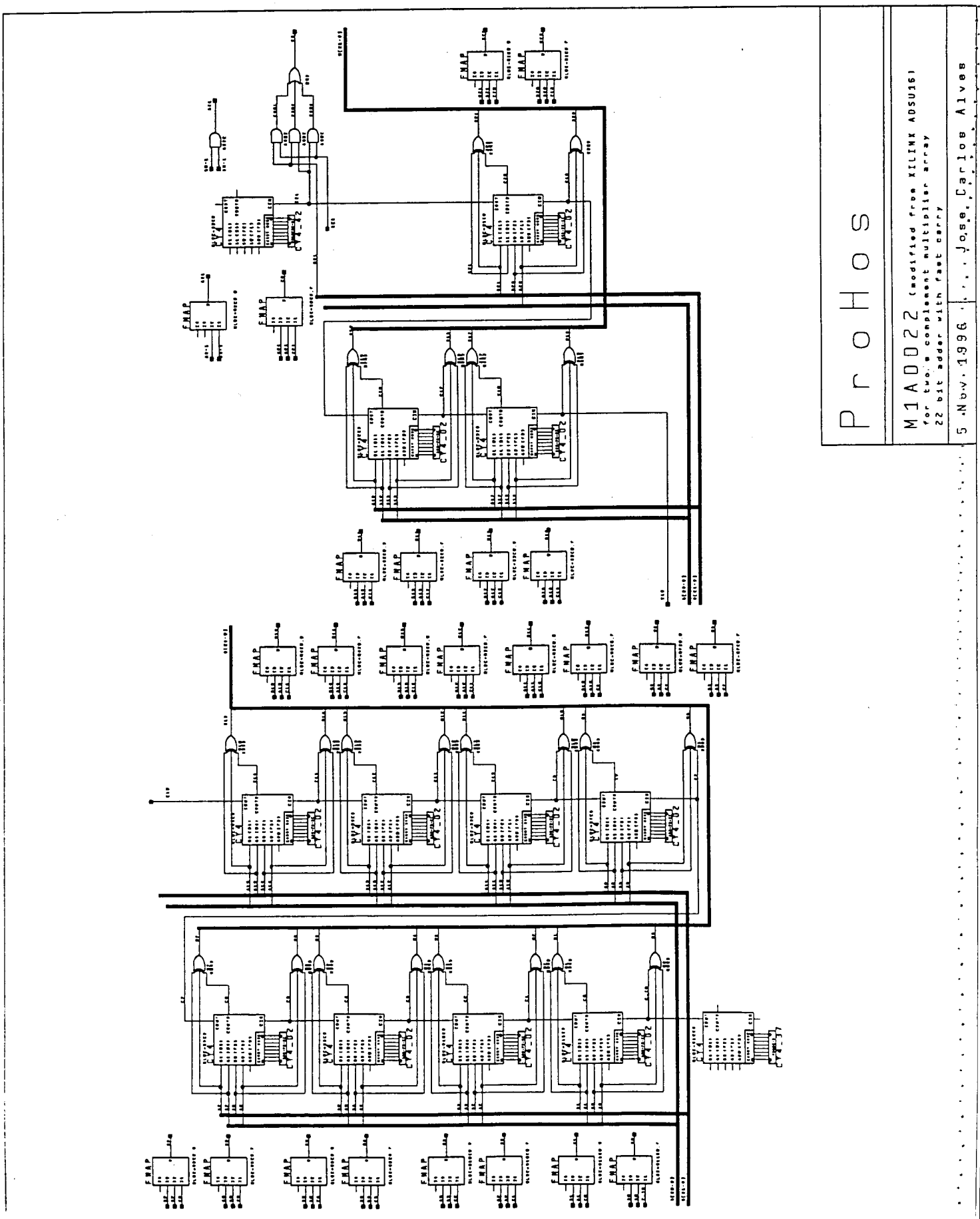

ProHos
MACNPXa
13 Dez 1996
Jose Carlos Alves
Main memory data bus
MAC8NP
8 bit 5 pipe stages
FD_8RE
IFD16_1
IFD16
OBUF8
X3 to X4 interface
X0 to X3 interface
Memory M0 data bus
Memory M1 data bus


ProHos
MAC8NP
Multiply-Accumulate unit. 8 bit. 5 pipe stages
29 Nov 1996
Jose Carlos Alves
ACC_R28
ACC_R34
ACC_R42
M2[15:0]
M3[23:0]
M4[31:0]
READY
FD32RE
M24X8_V
FD24RE
M16X8_V
FD16RE
M8X8_V
FD8RE
LD_MT
DD1N[7:0]
D11N[7:0]
D21N[7:0]
D31N[7:0]
RESET
CLOCK
CE


OUT[41:0]
FD16RE
FD16RE
FD16RE
FD10RE
ADD16
ADD16
ADD10
CLOCK
RESET
LOAD_AT
CE
IN[31:0]
ProHos
ACC_R42
23 Nov 1996
Jose Carlos Alves


MUL[31:0]
A[23:0]
B[7:0]
ProHos
M24X8_V
24x8 two's complement multiplier array
5, May, 1996
Jose Carlos Alves
M2add23
M1add22

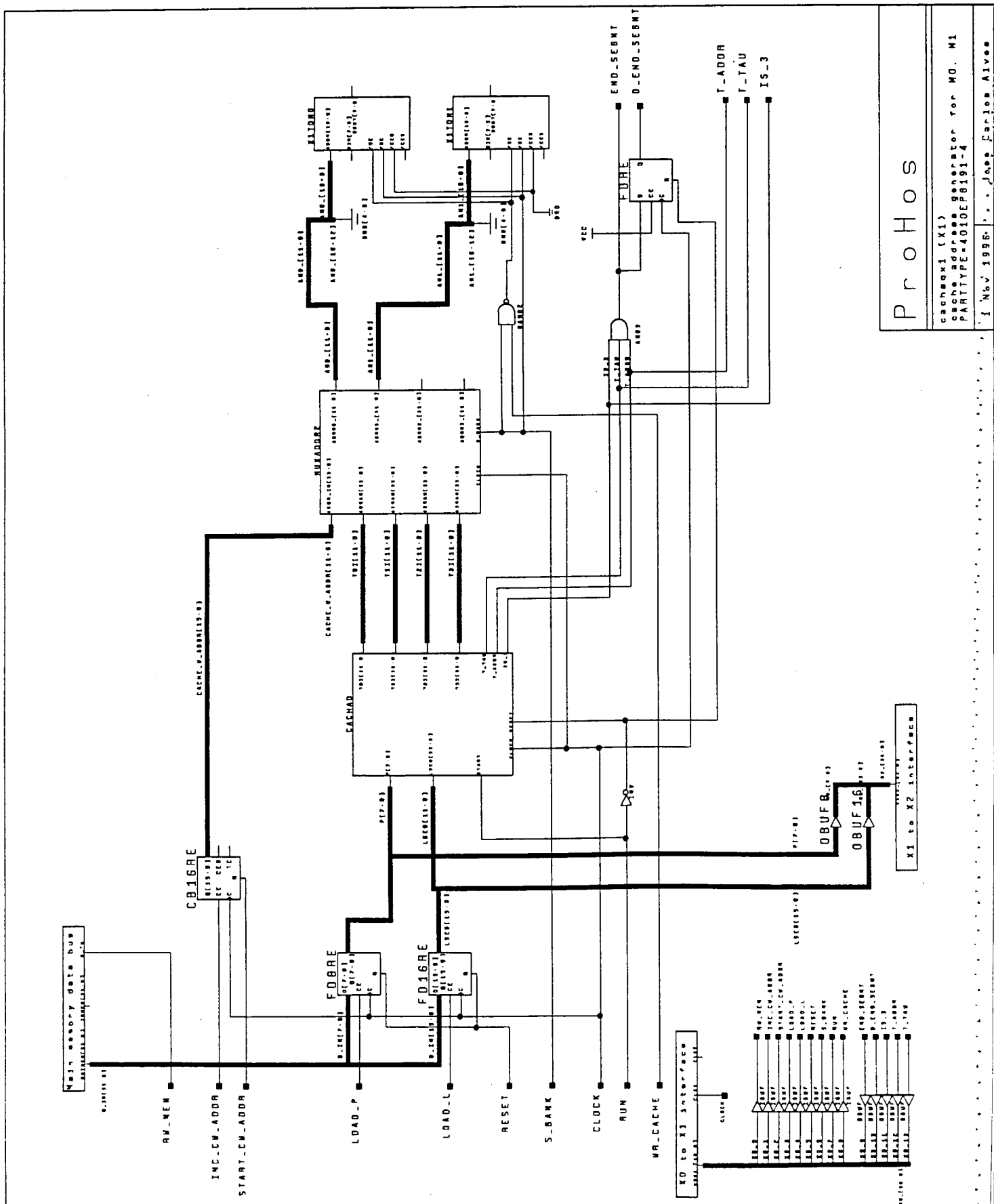

ProHoS
FA7_3
Type 7 full-adder cell
for two's complement multiplier array
5 Nov 1996
Jose Carlos Alves
SUM
COUT
HMAP
FMAP
Gmap
Hmap
Fmap
A1
B0
A0
B1
CIN

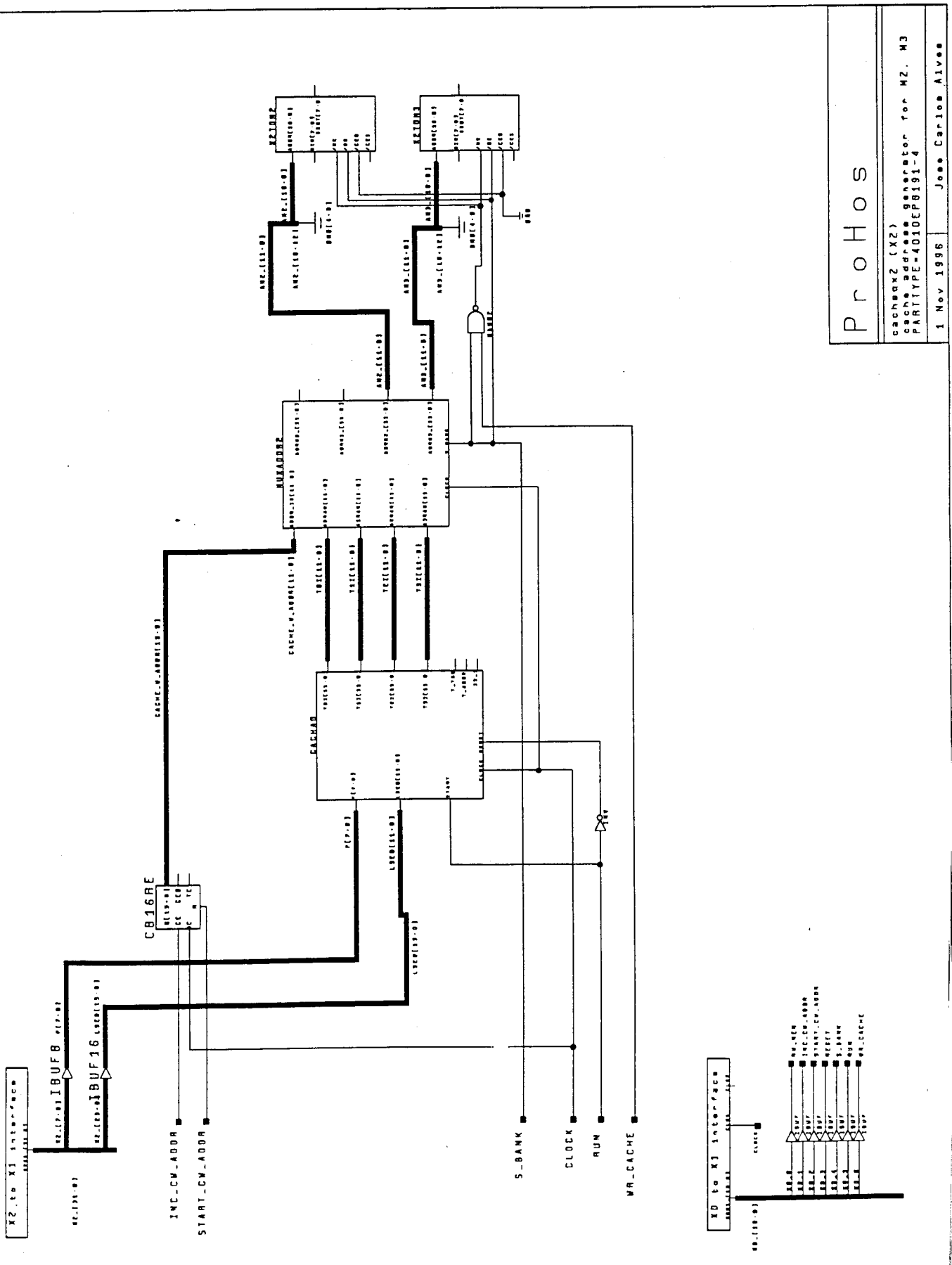

ProHos
M1A0022 (modified from XILINX ADSU161)
for two's complement multiplier array
22 bit adder with fast carry
5 Nov 1996
José Carlos Alves

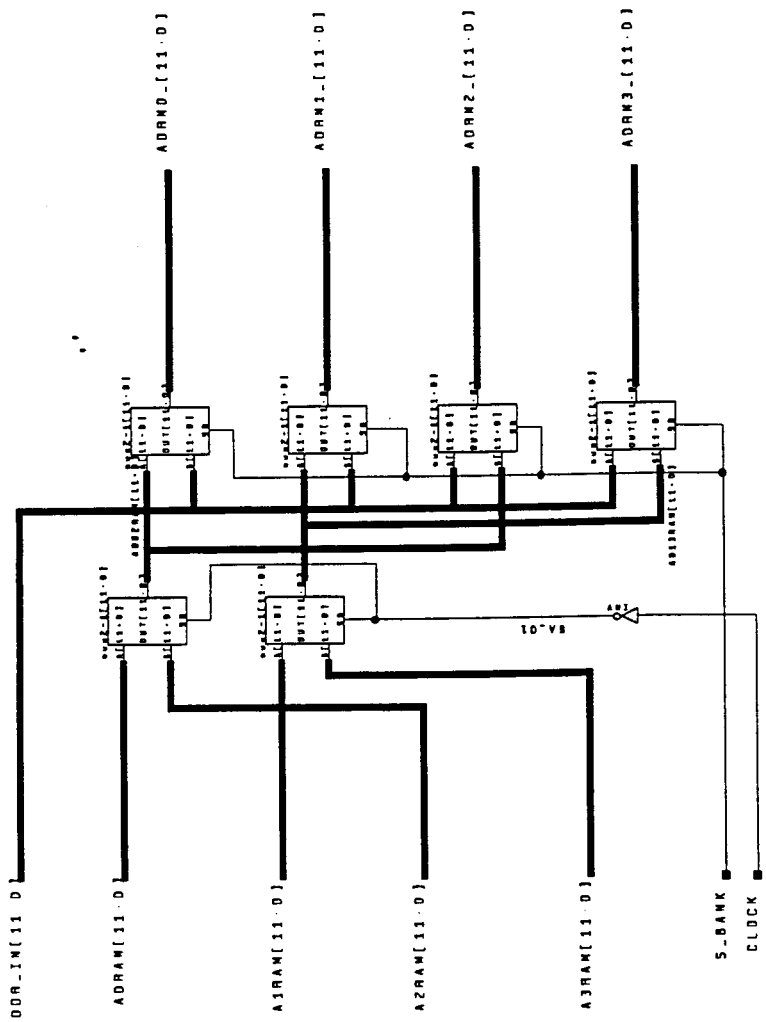
E.2 Geradores de endereços das memórias *cache*

Apresentam-se os circuitos lógicos desenvolvidos para os FPGAs X1 e X2 do CVR, que incluem os circuitos geradores de endereços para as memórias *cache* do ProHos. Nas páginas seguintes são listados, de forma descendente na hierarquia, os circuitos fundamentais que compõem cada um:

- `CACHSQX1` e `CACHSQX2` são os circuitos desenvolvidos para os FPGAs X1 e X2, respectivamente. Os 2 circuitos principais que os formam são o sequenciador de endereços para as memórias *cache* (`CACHAD`) e o circuito selector de endereços (`MUXADDR2`). Os dois símbolos existentes do lado direito em cada esquema (`X1TOM0`, `X1TOM1`, `X2TOM2` e `X2TOM3`) constituem a interface com os bancos de memória rápida do CVR.
- O circuito gerador de endereços para as memórias *cache* (`CACHAD`) é formado pelos 2 componentes principais: gerador de domínio `TGEN` e sequenciador de endereços `ADGEN`.


ProHos
cachex1 (X1)
cache address generator for MO. M1
PARTTYPE=4010EP81191-4
1 Nov 1996
END_SEGMT
D_END_SEGMT
T_ADDR
T_TAU
IS_3
CB16RE
FDRE
FD16RE
OBUF8
OBUF16
X1 to X2 interface
X0 to X1 interface
RW_MEM
INC_CW_ADDR
START_CW_ADDR
LOAD_P
LOAD_L
RESET
S_BANK
CLOCK
RUN
WR_CACHE

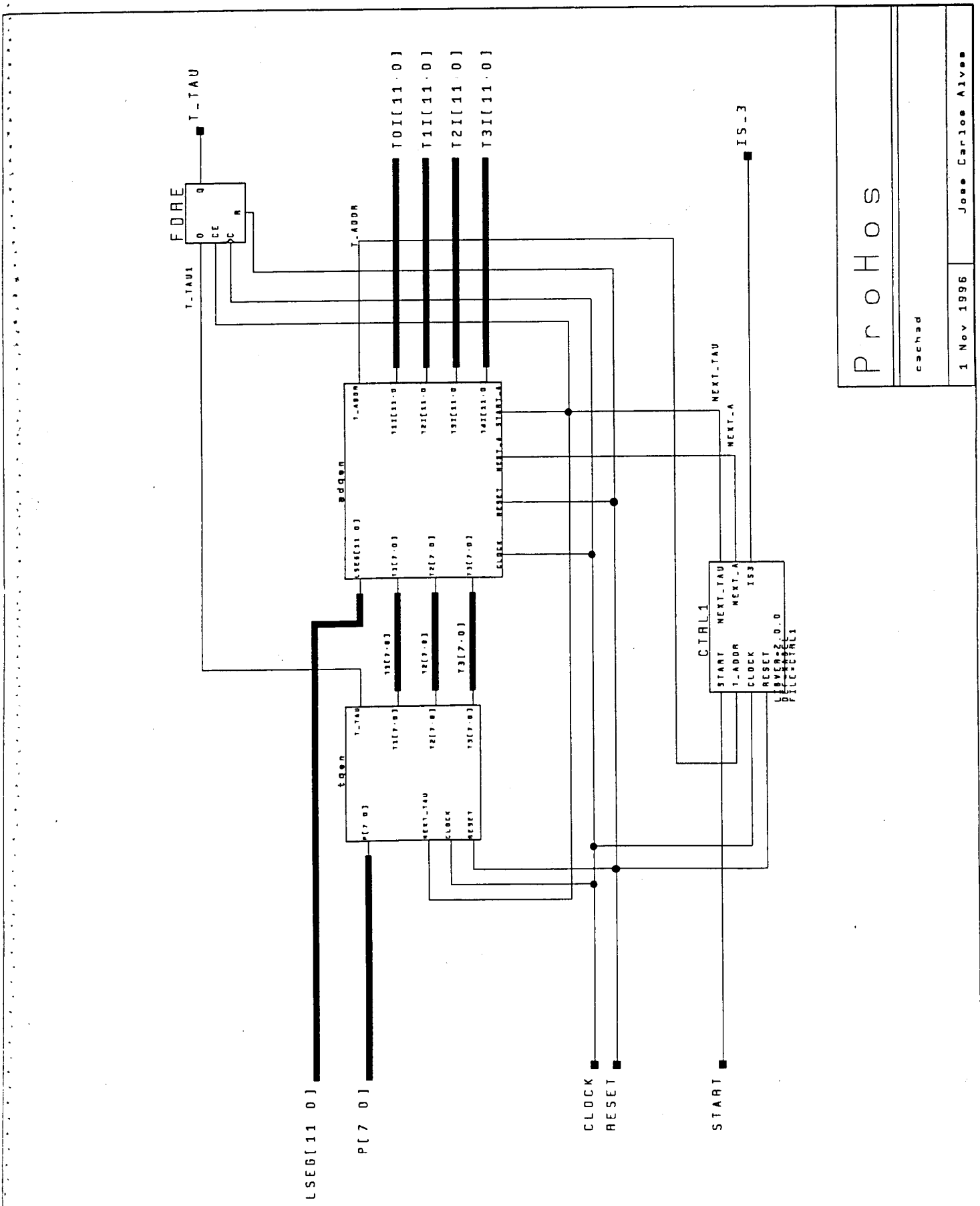

ProHos
cache2 (X2)
cache address generator for M2, M3
PARTTYPE=4010EP81191-4
1 Nov 1996
Jose Carlos Alves
CB16RE
IBUF8
IBUF16
INC_CW_ADDR
START_CW_ADDR
S_BANK
CLOCK
RUN
WR_CACHE

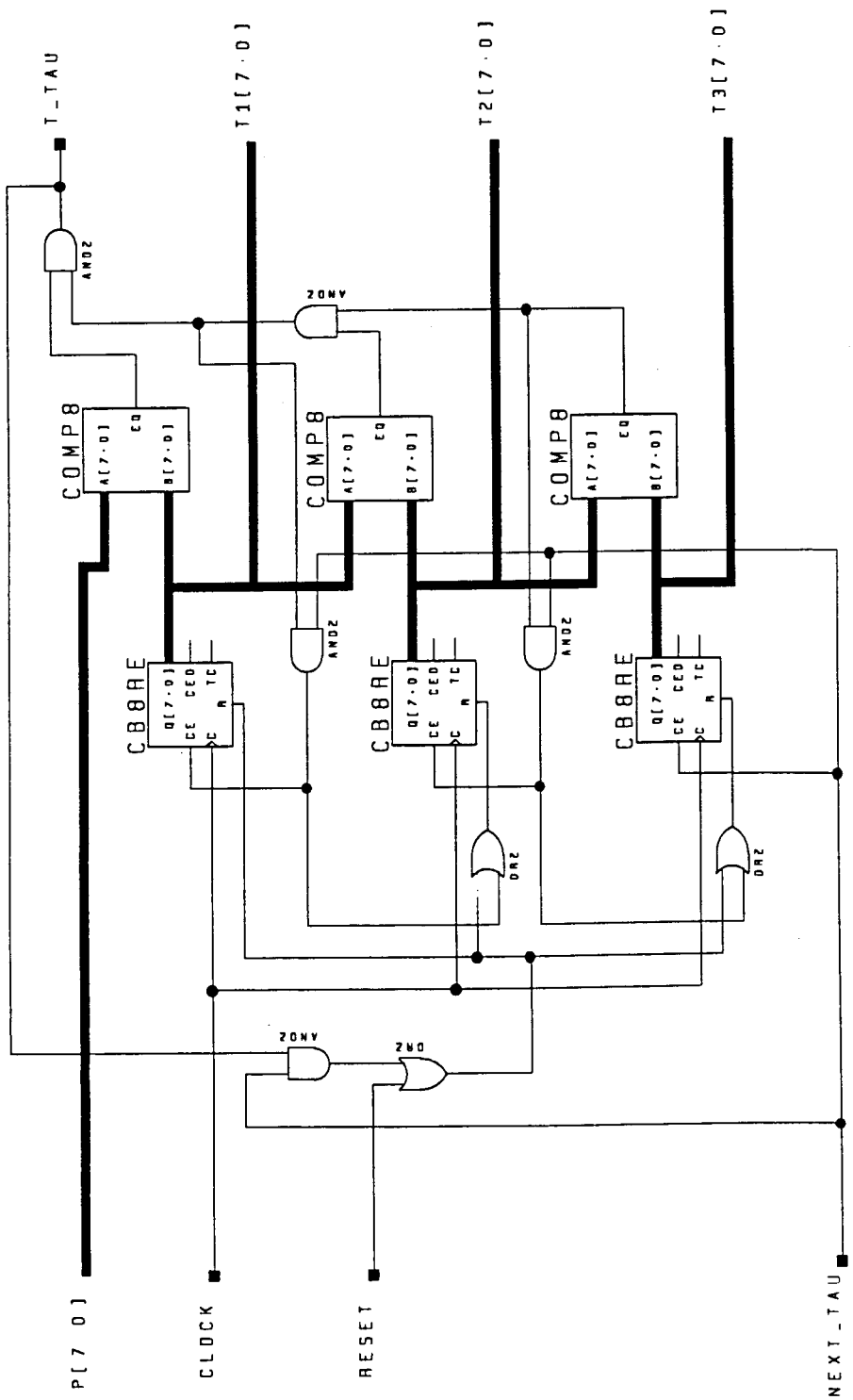


ProHos

MUXADDR2

1 Nov 1996, Joao Carlos Alves

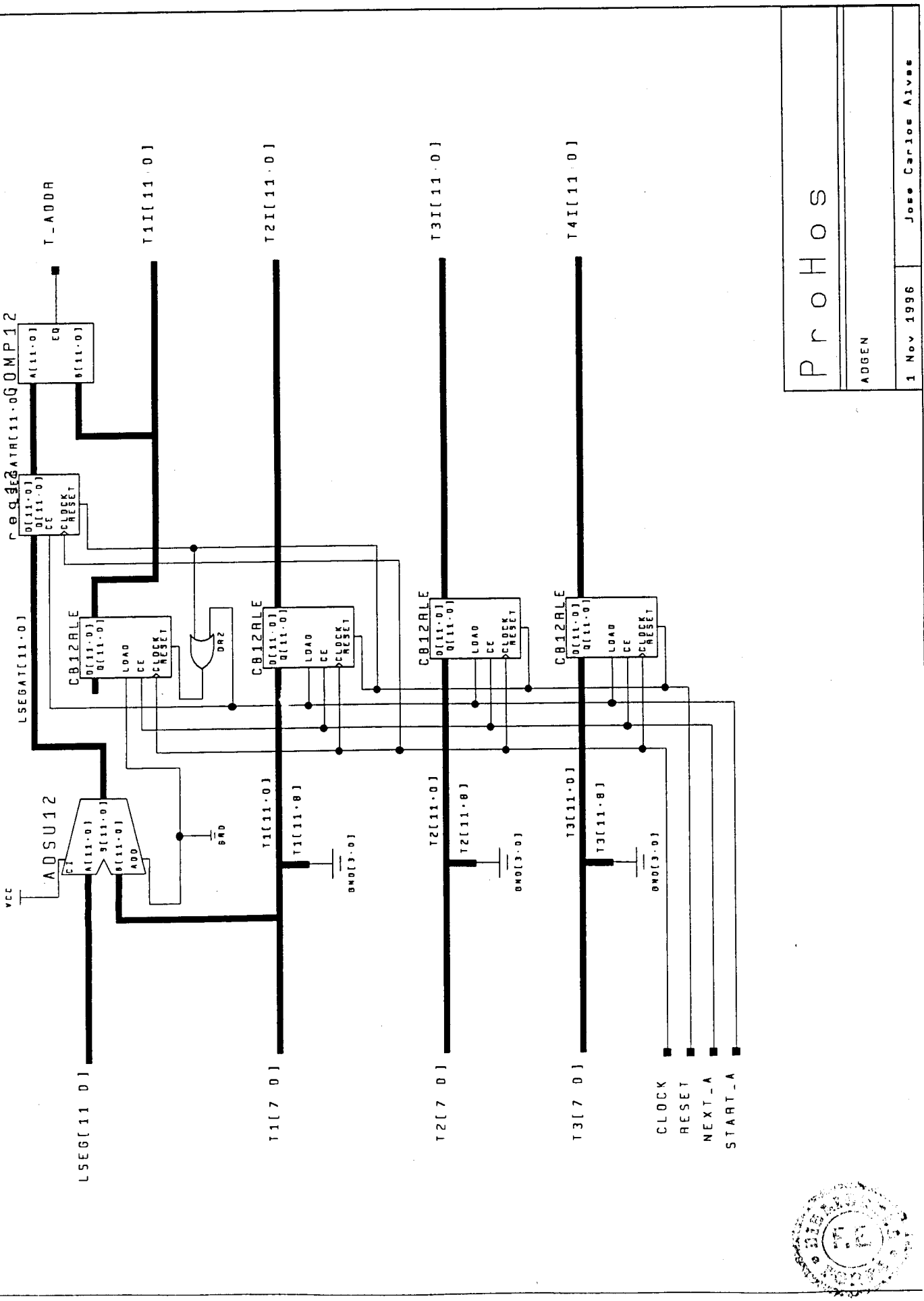

ProHos
cachad
1 Nov 1996
Jose Carlos Alves
T_TAU
FORE
T_TAU1
T_ADDR
T0I[11..0]
T1I[11..0]
T2I[11..0]
T3I[11..0]
IS_3
NEXT_TAU
NEXT_A
adgen
T_ADDR
LSEG[11..0]
T1[7..0]
T2[7..0]
T3[7..0]
CLOCK
RESET
CTRL1
START
T_ADDR
CLOCK
RESET
LSEG[11..0]
P[7..0]
START



ProHos

TGEN

1 Nov, 1996 | Jose Carlos Alves


ProHos
ADGEN
1 Nov 1996
Jose Carlos Alves
T_ADDR
T1I[11:0]
T2I[11:0]
T3I[11:0]
T4I[11:0]
COMP12
CB12RLE
DR2
ADSU12
VCC
GND
LSEGAT[11:0]
LSEG[11:0]
T1[7:0]
T2[7:0]
T3[7:0]
CLOCK
RESET
NEXT_A
START_A