Universidade de Aveiro
2004

Departamento de Electrónica e Telecomunicações

Iouliia Skliarova

Arquitecturas reconfiguráveis para problemas de optimização combinatória

Universidade de Aveiro
2004

Departamento de Electrónica e Telecomunicações

Iouliia Skliarova

Arquitecturas reconfiguráveis para problemas de optimização combinatória

dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Doutor em Engenharia Electrotécnica, realizada sob a orientação científica do Dr. António Manuel de Brito Ferrari Almeida, Professor Catedrático do Departamento de Electrónica e Telecomunicações da Universidade de Aveiro

o júri

presidente

Prof. Dr. António José Venâncio Ferrer Correia
professor catedrático da Universidade de Aveiro

Prof. Dr. António Manuel de Brito Ferrari de Almeida
professor catedrático da Universidade de Aveiro (orientador)

Prof. Dr. António Manuel Melo de Sousa Pereira
professor catedrático da Universidade de Aveiro

Prof. Dr. António Rui Oliveira Silva Borges
professor associado da Universidade de Aveiro

Prof. Dr. José Carlos dos Santos Alves
professor auxiliar da Faculdade de Engenharia da Universidade do Porto

Prof. Dr. José João Henriques Teixeira de Sousa
professor auxiliar do Instituto Superior Técnico da Universidade Técnica de Lisboa

Prof. Dr. João Manuel Paiva Cardoso
professor auxiliar da Faculdade de Ciências e Tecnologia da Universidade do Algarve

agradecimentos

Gostaria de expressar os meus agradecimentos ao meu orientador, Professor António B. Ferrari, por me ter ajudado a conduzir este trabalho até ao fim e por comentários e sugestões que fez ao longo da elaboração desta tese que contribuíram para o aperfeiçoamento da sua qualidade.

Agradeço também aos meus colegas do Laboratório de Sistemas Computacionais, em particular ao Arnaldo Oliveira, à Andreia Melo e ao Artur Pereira por todas as sugestões, em particular a nível do português, feitas ao longo da realização deste trabalho.

Reconheço o apoio financeiro da Fundação para a Ciência e a Tecnologia que suportou este trabalho através da Bolsa de Doutoramento com a referência PRAXIS XXI/BD/21353/99. A apresentação dos resultados deste trabalho em várias conferências internacionais foi possível graças ao apoio da Fundação para a Ciência e a Tecnologia, da Fundação Calouste Gulbenkian e do Instituto de Engenharia Electrónica e Telemática de Aveiro.

Fico muito grata à minha família, em particular aos meus pais, Tatiana e Valeri, por ajuda, suporte e encorajamento que me deram.

Apoio financeiro da Fundação para a Ciência e a Tecnologia e do Fundo Social Europeu no âmbito do III Quadro Comunitário de Apoio.

resumo

Os problemas combinatórios têm uma gama extremamente ampla de aplicações numa variedade de áreas de engenharia, incluindo teste de circuitos electrónicos, reconhecimento de padrões, síntese lógica, etc. Muitos dos problemas de interesse pertencem às classes *NP-hard* e *NP-complete*, o que implica que os algoritmos relevantes têm no pior caso complexidade exponencial. Este facto impede a solução de muitos problemas práticos com a ajuda de computadores convencionais. As implementações em circuitos integrados específicos também não são viáveis, em particular por causa da própria heterogeneidade dos problemas combinatórios. Uma solução alternativa consiste no uso de dispositivos reconfiguráveis que podem ser personalizados para um algoritmo específico e reutilizados para outros algoritmos via uma simples reprogramação da sua estrutura interna. As implementações baseadas em hardware reconfigurável permitem optimizar a execução dos algoritmos relevantes com a ajuda de técnicas tais como processamento paralelo, unidades funcionais personalizadas, etc. Tais implementações possibilitam conter o efeito de crescimento exponencial do tempo de computação, permitindo deste modo a solução de problemas combinatórios complexos.

Recentemente foram desenvolvidos vários sistemas reconfiguráveis destinados a resolver problemas combinatórios. Estes são principalmente baseados na ideia de hardware específico para a instância, em que para cada instância do problema é gerado um circuito particular. Nesta tese exploramos duas abordagens alternativas. A primeira é orientada para o domínio e permite processar uma variedade de problemas da área da computação combinatória. Para tal é projectado e implementado um processador combinatório reconfigurável e são desenvolvidos métodos e ferramentas que asseguram a sua reconfiguração dinâmica parcial. A segunda abordagem é orientada para a aplicação e é destinada a resolver um problema combinatório específico. Em particular, é proposta uma arquitectura inovadora para a solução do problema de satisfação booleana com a ajuda de uma combinação de software e de hardware reconfigurável. A técnica adoptada elimina a compilação de hardware específica à instância e permite processar problemas que excedem os recursos lógicos disponíveis. São também exploradas as possibilidades de implementação em hardware reconfigurável de estratégias evolutivas para o caso do problema do caixeiro viajante.

Esta tese estende o domínio de aplicação da computação reconfigurável ao demonstrar que esta é capaz de acelerar algoritmos com fluxos de controlo complexos.

abstract

Combinatorial problems have an extremely wide range of practical applications in a variety of engineering areas, including the testing of electronic circuits, pattern recognition, logic synthesis, etc. Many of the problems of interest belong to the classes *NP-hard* and *NP-complete*, which implies that the relevant algorithms have an exponential worst-case complexity. This fact precludes the solution of many practical problems with conventional computers. ASIC-based implementations are also not viable, in particular because of the inherent heterogeneity of combinatorial problems. Reconfigurable devices offer an alternative solution, which can be customized to the requirements of a specific algorithm and reutilized for other algorithms via a simple reprogramming of their internal structure. Implementations based on reconfigurable hardware permit the execution of the relevant algorithms to be optimized with the aid of such techniques as parallel processing, personalized functional units, etc. Such implementations allow the effect of exponential growth in the computation time to be delayed, thus enabling more complex problem instances to be solved.

Recently, a few reconfigurable engines for combinatorial problems have been developed. They are mainly based on the idea of *instance-specific* hardware, which assumes that a particular circuit is generated for each problem instance. In this thesis we explore two alternative approaches. The first, *domain-specific*, approach enables a variety of problems in the area of combinatorial computation to be addressed. For this purpose, a reconfigurable combinatorial processor has been designed and implemented and a number of methods and tools that support its partial dynamic reconfiguration have been developed. The second, *application-specific*, approach is oriented towards solving individual combinatorial problems. In particular, a novel architecture is proposed for solving the Boolean satisfiability problem with the aid of software and reconfigurable hardware. The adopted technique avoids instance-specific hardware compilation and permits problems that exceed the available logic resources to be solved. The possibility of implementing evolutionary strategies for the traveling salesman problem in reconfigurable hardware is also explored.

This thesis extends the application domain of reconfigurable computing by demonstrating that it is effective in accelerating algorithms with complex control flows.

резюме

Комбинаторные задачи имеют множество практических областей применения, включая тестирование электронных схем, распознавание образов, логический синтез и т.д. Многие важные комбинаторные задачи являются НП-трудными и НП-полными, что подразумевает в худшем случае экспоненциальную сложность соответствующих алгоритмов. Этот факт не позволяет решать многие практические комбинаторные задачи с помощью компьютеров общего назначения. Использование специализированных интегральных схем также не является рациональным, в основном по причине разнородности комбинаторных задач. Альтернативный подход заключается в использовании реконфигурируемых устройств, которые способны имитировать выполнение некоторого алгоритма, после чего они могут быть перепрограммированы на выполнение других алгоритмов путем простого внесения изменений в их конфигурацию. Реализация на основе реконфигурируемых устройств позволяет оптимизировать выполнение различных алгоритмов за счёт использования методов параллельной обработки данных, а также специализированных функциональных узлов. Такие реализации позволяют задержать экспоненциальный рост времени вычислений, предоставляя таким образом возможность для решения более сложных задач.

В последнее время было предложено несколько реконфигурируемых систем, предназначенных для решения комбинаторных задач. Большинство таких систем основаны на идее *индивидуальной схемы*, которая подразумевает создание специфической конфигурации устройства для каждого примера задачи. В этой диссертации исследуются два альтернативных подхода. Первый из них, отражающий специфику конкретной области, позволяет рассматривать сразу некоторое множество комбинаторных задач. Для этого был спроектирован и реализован реконфигурируемый комбинаторный процессор, и были разработаны методы и средства, позволяющие осуществлять его частичную динамическую реконфигурацию. Второй подход, являющийся проблемно-ориентированным, предназначен для решения индивидуальных комбинаторных задач. В качестве конкретного примера, предлагается архитектура системы для решения задачи выполнимости булевой функции, базирующаяся на комбинировании программных и реконфигурируемых аппаратных средств. Предлагаемый метод полностью отменяет необходимость компиляции специфических для каждого примера задачи конфигураций и позволяет решать задачи, чьи характеристики превосходят ресурсы имеющегося в наличии устройства.

Кроме того, исследуются возможности реализации в реконфигурируемых устройствах эволюционных методов для задачи коммивояжера.

Эта диссертация расширяет область применения реконфигурируемой вычислительной техники, продемонстрировав её способность ускорять выполнение алгоритмов, характеризующихся сложной последовательностью команд.

Índice

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

ABEL	*Advanced Boolean Expression Language*
ADN	Ácido Desoxirribonucleico
ADRES	*Architecture for Dynamically Reconfigurable Embedded System*
ALU	*Arithmetic and Logic Unit*
ASIC	*Application-Specific Integrated Circuit*
CAD	*Computer-Aided Design*
CCM	*Cube Calculus Machine*
CLB	*Configurable Logic Block*
CNF	*Conjunctive Normal Form*
CPLD	*Complex Programmable Logic Device*
DCM	*Digital Clock Manager*
DIMACS	*Center for Discrete Mathematics & Theoretical Computer Science*
DLL	*Delay-Locked Loop*
DMA	*Direct Memory Access*
DNF	*Disjunctive Normal Form*
DSP	*Digital Signal Processor*
EA	*Evolutionary Algorithms*
EDIF	*Electronic Design Interchange Format*
EEPROM	*Electrically Erasable PROM*
EPROM	*Erasable PROM*
FIR	*Finite Impulse Response*
FPGA	*Field-Programmable Gate Array*
FSM	*Finite State Machine*
GRASP	*Generic seaRch Algorithm for the Satisfiability Problem*
GRM	*General Routing Matrix*
GS	*Graph-Scheme*
HCPLD	*High-Capacity PLD*
HDL	*Hardware Description Language*
HT	*Hardware Template*
I/O	*Input/Output*
IOB	*Input/Output Block*
IP	*Intellectual Property*
ISE	*Integrated Software Environment*
JTAG	*Joint Test Action Group*
LAN	*Local Area Network*
LC	*Logic Cell*

LSI	*Large-Scale Integration*
LUT	*Look-Up Table*
MGT	*Multi-Gigabit Transceiver*
MIMD	*Multiple Instruction streams, Multiple Data streams*
MIPS	*Million Instructions Per Second*
MPGA	*Mask-Programmable Gate Array*
MSI	*Medium-Scale Integration*
NAPA	*National Adaptive Processing Architecture*
NRE	*Non-Recurring Engineering*
ORCA	*Optimized Reconfigurable Cell Array*
PAL	*Programmable Array Logic*
PAM	*Programmable Active Memories*
PCI	*Peripheral Component Interconnect*
PFU	*Programmable Functional Unit*
PLA	*Programmable Logic Array*
PLD	*Programmable Logic Device*
PMC	*PCI Mezzanine Card*
PMX	*Partially Matched Crossover*
PODEM	*Path-Oriented DEcision Making*
POS	*Product-of-Sums*
POSIT	*PrOpositional SatIsfiability Testbed*
PRISC	*Programmable Instruction Set Computers*
PRISM	*Processor Reconfiguration through Instruction Set Metamorphosis*
PROM	*Programmable Read-Only Memory*
PSM	*Programmable Switch Matrix*
RAM	*Random Access Memory*
RAW	*Reconfigurable Architecture Workstation*
RCP	*Reconfigurable Combinatorial Processor*
RCU	*Reconfigurable Control Unit*
RENCO	*REconfigurable Network Computer*
RFU	*Reconfigurable Functional Unit*
RISC	*Reduced Instruction Set Computer*
ROM	*Read-Only Memory*
RSA	*Rivest-Shamir-Adelman*
RTL	*Register Transfer Level*
SAT	*Boolean Satisfiability*
SCSI	*Small Computer System Interface*
SIMD	*Single Instruction stream, Multiple Data streams*
SPLD	*Simple PLD*
SPYDER	*REconfigurable Processor Development System*
SRAM	*Static Random Access Memory*
SSI	*Small-Scale Integration*

TSP	*Traveling Salesman Problem*
TSPLIB	*Library of Traveling Salesman and related Problems*
VGA	*Video Graphics Array*
VHDL	*VHSIC Hardware Description Language*
VHSIC	*Very High Speed Integrated Circuit*
VIRTEX-EM	*Virtex-E Extended Memory*
VLIW	*Very Long Instruction Word*
VLSI	*Very Large-Scale Integration*

1 Introdução

Sumário

Neste capítulo dá-se uma motivação e estabelecem-se os objectivos principais do trabalho. É também feita uma introdução à organização da tese.

1.1 Motivação

1.1.1 Problemas de optimização combinatória

Os problemas de optimização combinatória surgem em diversas áreas, nomeadamente na síntese, optimização e teste de circuitos digitais [Micheli94, Breuer00], no mapeamento, colocação e interligação de componentes de projecto de circuitos integrados [Togawa98], em topologia e cartografia [Tambouratzis98], em inteligência artificial [Zakrevski98], etc. São exemplos típicos de problemas de optimização combinatória determinar o caminho mais curto e o mais longo em grafos, coloração de grafos, optimização de funções booleanas, problemas de cobertura, problemas de codificação, etc.

Estes problemas estão bem estudados e podem ser formulados sobre entidades matemáticas diversas como grafos, matrizes, conjuntos, funções booleanas. Frequentemente é possível formalizar o problema em mais do que um destes modelos básicos ou algumas das suas variedades como hipercubos [Handbook00], diagramas de decisão binários [Bryant86], etc., podendo-se obter a representação num dos modelos através de transformações aplicadas a um outro.

Muitos dos problemas que surgem em optimização combinatória são de elevada complexidade, não sendo resolúveis em tempo polinomial (pertencem às classes *NP-complete* e *NP-hard*) [Garey79], exigindo meios computacionais extremamente poderosos e caros. Prestam-se no entanto à exploração de algoritmos de solução com um considerável grau de paralelismo.

É assim possível explorar a utilização de unidades de processamento especializadas para tarefas de optimização combinatória, investigando nomeadamente as seguintes vias:

1) exploração de estruturas dedicadas para problemas de optimização combinatória que possam ser usadas como núcleos de processamento combinatórios;
2) concepção de blocos funcionais para problemas específicos de optimização combinatória;
3) exploração de estruturas de computação que modelem directamente o problema.

Estas tarefas podem ser abordadas implementando as estruturas relevantes em software ou em hardware. Na abordagem de software utiliza-se um conjunto de instruções predefinidas que servem para mapear o algoritmo desenvolvido num computador de uso geral. Dado que os processadores convencionais não são optimizados para problemas combinatórios, o desempenho resultante será escasso. A abordagem de hardware pode ser, ao contrário, talhada à medida dum algoritmo particular garantindo o desempenho óptimo. Contudo, um circuito de hardware especializado só é capaz de executar a tarefa para a qual foi concebido, enquanto um computador de uso geral pode ser reutilizado para várias tarefas via uma simples modificação da sequência de instruções. Este compromisso de software/hardware obriga tradicionalmente os projectistas a optarem entre desempenho e flexibilidade.

A construção de hardware dedicado para problemas e domínios específicos envolve custos e tempos de desenvolvimento consideráveis. A experiência mostra que os benefícios resultantes são frequentemente escassos ou mesmo inexistentes devido ao ritmo de evolução da tecnologia dos processadores convencionais que leva a que estruturas de computação menos eficientes para o

domínio de aplicação em causa suplantem o desempenho de estruturas de computação especializadas, optimizadas para a aplicação, através da utilização de tecnologia mais recente. Acresce que no domínio deste trabalho a própria heterogeneidade dos problemas de optimização combinatória desaconselha a construção de coprocessadores especializados.

A invenção de dispositivos lógicos programáveis de alta capacidade, tais como FPGAs (*Field-Programmable Gate Array*) gerou um método de computação alternativo. As FPGAs reprogramáveis em sistema fornecem a base necessária para atingir o desempenho do hardware e a flexibilidade do software, dado que podem ser optimizadas para um algoritmo específico e reutilizadas para outros algoritmos via uma simples reprogramação da sua estrutura interna. Assim, pode-se construir processadores que sejam optimizados para cada aplicação específica através da reprogramação da funcionalidade das células lógicas básicas das FPGAs utilizadas , isto é, sem que tal envolva alterações a nível do "hardware".

1.1.2 Arquitecturas reconfiguráveis

O potencial de estruturas de computação reconfiguráveis foi já considerado na década de sessenta por G. Estrin [Estrin60] que advogava a construção de um computador cuja estrutura incluísse uma parte fixa e uma outra variável. Ao tempo porém a tecnologia disponível impedia uma realização eficiente desta ideia, que durante muitos anos permaneceu esquecida.

Os primeiros trabalhos que se enquadram neste novo paradigma de computação apareceram nos finais da década oitenta, de que é exemplo o PAM (*Programmable Active Memories*) desenvolvido nos laboratórios de investigação da *Digital Equipment* em Paris [Vuillemin96]. Na primeira metade da década passada surgiram vários protótipos de investigação de que se podem apontar como exemplos significativos os desenvolvidos na Universidade de Brown (PRISM, [Athanas93]), na *École Polytechnique Fédérale* de Lausanne (Spyder [Sanchez99]), e uma máquina, o Splash, que veio a ser explorada como um dos recursos computacionais do *Center for Computing Sciences* (anteriormente, *Supercomputing Research Center*) em Maryland, EUA, e de que foi desenvolvido um modelo de segunda geração, o Splash-2 [Buell96]. Todos eles utilizavam FPGAs disponíveis no mercado, tipicamente a última geração de FPGAs da Xilinx [Xilinx].

Nos anos mais recentes surgiram grupos que começaram a explorar a integração numa mesma pastilha de silício de um núcleo de processamento tradicional e de um bloco lógico reconfigurável, de modo a tirar partido da evolução da tecnologia integrada para aumentar a largura de banda das comunicações entre o sistema hospedeiro e o hardware reconfigurável, problema entretanto identificado como estando na origem das principais limitações do desempenho da primeira geração de processadores reconfiguráveis. Enquadram-se nesta linha projectos como o GARP da Universidade de Berkeley [Hauser00], MATRIX do *Massachusetts Institute of Technology* [Mirsky96], MorphoSys [Singh00] da Universidade de Irvine, OneChip da Universidade de Toronto [Jacob98], etc.

As aplicações em que tem sido testada a utilização de sistemas reconfiguráveis, abrangem domínios muito diversos, como processamento de imagem, processamento de vídeo, motores de busca em bases de dados genéticas, reconhecimento de padrões, redes neuronais, aplicações em física de altas energias, etc. As aplicações em que os níveis de desempenho atingidos foram mais elevados foram

aquelas que exibem um elevado grau de paralelismo e/ou utilizam formatos não-*standard* de representação da informação. Os problemas de optimização combinatória possuem ambas estas características e aparecem por isso à partida como um domínio onde será possível obter bons resultados na utilização de sistemas reconfiguráveis.

Deste modo torna-se viável a construção de um coprocessador, ou acelerador, reconfigurável para problemas combinatórios. A ideia de utilização de FPGAs como coprocessadores para algoritmos computacionalmente intensivos não é nova. Neste contexto, é importante fazer uma distinção entre os três seguintes tipos de aceleradores:

1) *os aceleradores orientados para a instância* exigem a geração de uma configuração de FPGA que é só capaz de resolver uma única instância específica dum dado problema;
2) *os aceleradores orientados para a aplicação* podem, em princípio, processar qualquer instância dum dado problema;
3) *os aceleradores orientados para o domínio* são destinados a resolver um conjunto de problemas e das suas instâncias de uma dada área.

Até ao início deste trabalho, na aceleração de problemas combinatórios com a ajuda de hardware reconfigurável foi essencialmente explorada a primeira alternativa, enquanto nesta tese serão averiguadas as duas últimas.

Embora a ideia de uso de aceleradores baseados em hardware reconfigurável pareça ser óbvia, existe uma série de entraves significativos. Por exemplo, nos aceleradores orientados para a instância o desafio principal é reduzir o tempo de geração de configurações específicas; nos aceleradores orientados para a aplicação a questão essencial é como comutar entre instâncias rapidamente e como processar instâncias que excedem os recursos do próprio acelerador; nos aceleradores orientados para o domínio o desafio importante é como acomodar uma variedade de problemas diferentes de uma maneira eficiente, sem que os recursos necessários expludam.

1.2 Objectivos

Os objectivos principais deste trabalho são os seguintes:

1) *Exploração do potencial da computação reconfigurável.* Com os avanços da tecnologia, a capacidade lógica de dispositivos programáveis tem aumentado significativamente. Para além da lógica pura, estes dispositivos começaram a incluir várias estruturas adicionais, tais como blocos de memória embutidos, etc. Por estas razões, torna-se necessário averiguar como estas facilidades novas podem contribuir para a consecução de um desempenho elevado. As capacidades crescentes de dispositivos lógicos programáveis estendem também o domínio de aplicações possíveis. Neste contexto, a utilização eficiente da computação reconfigurável para a implementação de algoritmos com fluxos de controlo complexos, tais como os empregues na solução de problemas combinatórios, é muito importante pois muitas aplicações práticas envolvem algoritmos complicados.
2) *Análise de modelos matemáticos.* Para a especificação e solução de problemas combinatórios podem ser usados vários modelos matemáticos. Dado que frequentemente estes modelos são

mutuamente convertíveis, torna-se necessário efectuar um estudo e uma análise comparativa de vários modelos a fim de estimar a sua adequabilidade para implementações baseadas em hardware reconfigurável. Deste modo será possível seleccionar um único modelo que servirá de base para um núcleo de problemas específicos.

3) *Desenvolvimento de algoritmos eficientes para arquitecturas reconfiguráveis.* A fim de propor uma arquitectura reconfigurável dedicada para a solução de problemas combinatórios é necessário identificar um núcleo de problemas básicos em optimização combinatória. O passo seguinte consiste na concepção de algoritmos eficientes para sistemas reconfiguráveis, em que um dos objectivos importantes é explorar o grau de paralelismo do problema.

4) *Exploração de estratégias de reconfiguração dinâmica parcial.* Neste contexto, torna-se interessante explorar metodologias de projecto orientadas para a reutilização de componentes, aproveitando para o efeito as técnicas características da programação orientada por objectos. É indispensável averiguar as possibilidades de reconfiguração parcial e dinâmica das implementações baseadas em dispositivos programáveis estaticamente.

5) *Concepção de arquitecturas reconfiguráveis orientadas para o domínio.* Este objectivo consiste no desenvolvimento de ferramentas de apoio à concepção e realização de processadores reconfiguráveis dedicados a problemas de optimização combinatória. Os alvos principais a atingir são os seguintes: especificação e descrição de um conjunto de operações comuns em problemas de optimização combinatória; implementação em hardware reconfigurável do conjunto de operações definidas; realização da reconfiguração dinâmica das operações do conjunto predefinido; concepção e implementação de circuitos de controlo reconfiguráveis para os algoritmos de optimização combinatória; análise de estratégias de utilização de reconfiguração dinâmica; desenvolvimento de ferramentas de apoio à realização da reconfiguração dinâmica parcial. A fim de facilitar o processo de desenvolvimento de processadores combinatórios reconfiguráveis é essencial implementar bibliotecas de unidades funcionais e de circuitos de controlo reutilizáveis.

6) *Concepção de arquitecturas reconfiguráveis orientadas para a aplicação.* Este objectivo é uma alternativa ao anterior e consiste na exploração de técnicas de desenvolvimento orientadas para a aplicação e destinadas a reduzir o tempo da reconfiguração, eliminar a compilação de hardware específica para a instância do problema e permitir processar problemas que excedem a capacidade lógica do dispositivo programável disponível. Para atingir este fim terão de ser seleccionados problemas particulares, para os quais deverão ser desenvolvidos e implementados circuitos que satisfaçam todas as restrições impostas.

7) *Avaliação dos resultados.* O último objectivo consiste na avaliação do desempenho das soluções propostas para os problemas de optimização combinatória em estudo. Terão que ser feitos os três tipos seguintes de avaliação: a) Comparação do desempenho dos circuitos desenvolvidos com o das implementações dos mesmos algoritmos em software; b) Avaliação dos circuitos contra os melhores programas resolutores existentes, aproveitando para tal as instâncias de teste disponíveis publicamente; c) Comparação com outras arquitecturas reconfiguráveis propostas no âmbito da investigação académica e destinadas a acelerar a solução de problemas combinatórios.

1.3 Organização da tese

Esta tese está dividida em oito capítulos. Os capítulos 2, 3 e 4 possuem um carácter introdutório, enquanto os capítulos restantes estão directamente relacionados com o trabalho desenvolvido.

No capítulo 2, que segue esta introdução, são discutidos os dispositivos lógicos programáveis, é descrita a evolução destes e são consideradas as tecnologias básicas que asseguram a sua reprogramação. É também apresentado o fluxo de projecto típico utilizado para os dispositivos lógicos programáveis de densidade elevada. Sendo as FPGAs a tecnologia utilizada neste trabalho apresenta-se uma classificação destas de acordo com a granulosidade dos respectivos elementos lógicos e com a arquitectura de encaminhamento implementada. Descrevem-se ainda as arquitecturas de FPGAs da Xilinx porque serviram de base para o desenvolvimento deste trabalho.

O capítulo 3 é dedicado à computação reconfigurável. Este capítulo contém uma revisão de vários tipos de organização de sistemas reconfiguráveis, nomeadamente são considerados modos de reconfiguração, mecanismos de interacção entre os componentes principais, modelos de programação de todo o sistema, etc. A seguir, são analisadas técnicas utilizadas na computação reconfigurável para atingir desempenho elevado. Apresentam-se também alguns exemplos de sistemas reconfiguráveis mais conhecidos e as áreas típicas de aplicação da computação reconfigurável. Finalmente, são descritas duas placas de desenvolvimento baseadas em FPGAs que foram utilizadas no âmbito deste trabalho.

No capítulo 4 é dada uma definição de problemas combinatórios. São apresentados alguns exemplos de problemas combinatórios, classificados de acordo com os modelos matemáticos utilizados normalmente para a sua especificação. A seguir são abordados vários tipos de algoritmos que podem ser utilizados para resolver problemas de optimização combinatória.

O capítulo 5 é dedicado à exploração de aceleradores orientados para o domínio de computações combinatórias. Para tal são apresentadas várias arquitecturas de coprocessadores reconfiguráveis que podem ser utilizados para a solução de problemas diferentes. São propostas técnicas e ferramentas que asseguram a comutação rápida entre vários problemas combinatórios. É também descrito um modelo de computação que se baseia na interacção de software e hardware reconfigurável e visa abordar o problema da capacidade lógica limitada.

No capítulo 6 é explorada a possibilidade de aceleração do problema de satisfação booleana (SAT - *Boolean Satisfiability*) com a ajuda de hardware reconfigurável. Para tal é sugerida uma técnica de desenvolvimento orientada para a aplicação que elimina completamente a compilação de hardware específica à instância e permite processar problemas que excedem a capacidade lógica da FPGA disponível. É também efectuada uma análise comparativa de outras implementações de vários algoritmos de solução de SAT baseadas em hardware reconfigurável e propostas no âmbito da investigação académica.

O capítulo 7 explora a possibilidade de aceleração de algoritmos evolutivos para o caso do problema do caixeiro viajante com a ajuda de hardware reconfigurável. É proposto o mapeamento para FPGA de uma das operações do algoritmo concebido.

Finalmente, o capítulo 8 sumaria as contribuições principais deste trabalho e expressa algumas ideias para investigação futura.

2 Dispositivos lógicos programáveis

Sumário

Graças ao desenvolvimento de novos tipos de dispositivos lógicos programáveis (PLD – *Programmable Logic Devices*), o processo de projecto de sistemas digitais sofreu grandes alterações durante as últimas décadas. Actualmente, muitos sistemas digitais são implementados com a ajuda de PLDs de densidade elevada. O mercado de PLDs continua a crescer o que resulta num grande número de dispositivos disponíveis. Estes distinguem-se por várias características as mais importantes das quais são a tecnologia de programação utilizada e a granulosidade dos elementos lógicos principais.

Neste capítulo introduzimos o conceito de dispositivo lógico programável, descrevemos a evolução destes desde as PROMs (*Programmable Read-Only Memories*) até às FPGAs e comparamo-los com outras metodologias de projecto tais como *full custom*, *standard cell* e *gate arrays*. Seguidamente, consideramos as tecnologias básicas que asseguram a reprogramação de PLDs, nomeadamente, os fusíveis, os antifusíveis, as células de memória SRAM (*Static Random Access Memory*) e os transístores de porta flutuante, realçando as suas vantagens e desvantagens relativas.

Para implementarem uma função lógica, os PLDs precisam de ser configurados com os dados apropriados que são normalmente gerados pelas ferramentas de projecto assistido por computador. Portanto, descrevemos o fluxo de projecto típico utilizado para os PLDs de densidade elevada.

Sendo as FPGAs a tecnologia utilizada neste trabalho apresenta-se uma classificação destas de acordo com a granulosidade dos respectivos elementos lógicos e com a arquitectura de encaminhamento implementada. A seguir, descrevem-se as arquitecturas de FPGAs da Xilinx porque serviram de base para o desenvolvimento deste trabalho.

2.1 Introdução

Os sistemas digitais são construídos com a ajuda de circuitos integrados. De acordo com o nível de integração de componentes electrónicos, os circuitos integrados são divididos em quatro categorias [Mano00]:

- Dispositivos SSI (*Small-Scale Integration*) contêm até uma dezena de portas lógicas independentes cujas entradas e saídas são ligadas aos pinos externos.
- Dispositivos MSI (*Medium-Scale Integration*) possuem complexidade de 10-100 portas lógicas que em conjunto executam uma dada função elementar.
- Dispositivos LSI (*Large-Scale Integration*) incorporam entre 100 e alguns milhares de portas lógicas. Estes incluem sistemas digitais assim como memórias pequenas e dispositivos programáveis elementares.
- Dispositivos VLSI (*Very Large-Scale Integration*) podem conter milhões de portas lógicas. Exemplos deste tipo de dispositivos são processadores e dispositivos programáveis complexos.

Ao contrário das gerações anteriores da tecnologia, nas quais os projectos incluíam um grande número de pastilhas de SSI, praticamente cada projecto digital desenvolvido hoje em dia utiliza os dispositivos de densidade elevada.

Para projectar circuitos digitais utilizam-se quatro tipos básicos de metodologias: *full custom*, *standard cell*, *gate array* e programável (*programmable*). O estilo *full custom* requer que cada função lógica primitiva seja projectada e optimizada à mão. Isto resulta numa boa utilização da área do silício e em características (tais como desempenho e consumo de potência) óptimas. Contudo, os projectistas devem manipular os detalhes de transístores individuais conduzindo assim a custos NRE (*Non-Recurring Engineering*) demasiado elevados. Por causa dos custos de desenvolvimento altos e de tempos de desenvolvimento longos, esta metodologia é aplicada hoje em dia só para componentes *standard* produzidos em grandes volumes, tais como memórias.

No estilo *standard cell*, os projectistas dispõem de bibliotecas de células que incluem vários blocos lógicos e funcionais que vão desde as portas lógicas simples até aos componentes mais complexos tais como somadores, descodificadores, multiplicadores, etc. A utilização de bibliotecas de células simplifica bastante o projecto, mas, ao mesmo tempo, reduz a flexibilidade e o nível de optimização que se pode alcançar. Para além disso, as células devem ser periodicamente renovadas para as adequar aos progressos na tecnologia de semicondutores.

Ambas as metodologias (*full custom* e *standard cell*) requerem um processo de fabrico personalizado usando um conjunto completo de máscaras únicas. Como resultado, os custos para a criação das máscaras bem como o tempo de fabrico são bastante elevados. Em alternativa, um circuito pode ser produzido usando um padrão de ligações personalizado aplicado a uma matriz de portas lógicas que inicialmente estão desconexas. MPGAs (*Mask-Programmable Gate Array*) são um exemplo deste tipo de dispositivos. A especialização das MPGAs efectua-se nas últimas etapas do processo de fabrico interligando os transístores relevantes com pistas especializadas reduzindo deste modo os custos e o tempo de produção.

Todas as metodologias descritas acima pressupõem que a função do dispositivo é definida durante a sua fabricação não podendo ser alterada posteriormente. Portanto os dispositivos produzidos destes modos *não são programáveis*[1]. Caso o utilizador do dispositivo possa alterar a sua função através de modificações estruturais, o dispositivo diz-se *programável*. Este capítulo é dedicado à descrição de dispositivos programáveis.

O termo *dispositivo lógico programável* (PLD) abrange todos os circuitos integrados que podem ser configurados pelo utilizador (i.e. alguém que usa o dispositivo para projectar um sistema) a fim de implementar a funcionalidade pretendida. A programação consiste em modificar a estrutura interna do PLD de acordo com a função desejada. Dispositivos lógicos programáveis caracterizam-se por grande flexibilidade de utilização mantendo os custos e o tempo de desenvolvimento reduzidos. Por estas razões a maioria dos protótipos bem como muitos projectos de produção são construídos actualmente utilizando PLDs. É de notar que a flexibilidade e programabilidade de PLDs têm como contrapartida desempenho e densidade reduzidos em comparação com os circuitos desenvolvidos usando outras metodologias.

A tabela 2.1 compara as características mais relevantes das quatro metodologias de projecto consideradas. A selecção da metodologia de projecto apropriada para cada caso depende de muitos factores tais como a funcionalidade desejada, custos e o tempo de desenvolvimento disponível. Os estilos *semi-custom* possuem características importantes que os torna adequados para os projectos lógicos de densidade elevada. As maiores desvantagens destes estilos são o tempo de desenvolvimento e os custos NRE associados. Em comparação, a desvantagem principal dos PLDs é que estes possuem um *overhead* devido aos elementos que controlam a programação resultando em circuitos maiores e mais lentos. Apesar disso, a frequência de funcionamento de circuitos implementados em PLDs é adequada para muitas aplicações.

Tabela 2.1. Comparação de várias metodologias de projecto.

Metodologia / **Característica**	**programável**	***gate array***	***standard cell***	***full custom***
Tempo para preparação de produção	imediato	médio	longo	longo
Tempo de desenvolvimento	muito curto	curto	curto-médio	longo
Custo de desenvolvimento	baixo	médio	médio-alto	muito alto
Desempenho	médio-alto	médio-alto	alto	muito alto
Densidade	média	grande	grande	muito grande
Dependência dos custos do volume de produção	pequena	média-grande	grande	grande
Flexibilidade arquitectural	média	média-grande	grande	muito grande
Incorporação de modificações	fácil	difícil	difícil	muito difícil
Eficiência de solução	média	média	grande	muito grande

Recentemente, começou a ser explorada a possibilidade de actualização das características de um produto depois deste ser vendido, o que representa uma vantagem fundamental de tecnologias

[1] Neste contexto, o termo *não programável* refere-se ao facto de impossibilidade de realização de modificações estruturais. Contudo o dispositivo pode incluir componentes que são microprogramáveis, tais como núcleos de processadores.

reprogramáveis [Kean00]. Neste caso, em vez de programar o PLD uma só vez durante a fabricação do sistema, pode-se carregar configurações aperfeiçoadas várias vezes durante a "vida" do dispositivo estendendo assim a sua duração. Embora tradicionalmente, os PLDs tenham sido considerados inadequados para volumes de produção consideráveis, as diferenças entre os custos de produção de PLDs e dos dispositivos desenvolvidos recorrendo a outras metodologias em volumes grandes estão a diminuir continuamente [Tredennick03].

2.2 A evolução dos dispositivos lógicos programáveis

Os dispositivos lógicos programáveis são normalmente divididos em duas classes: os PLDs elementares (SPLD – *Simple PLD*) e os de capacidade elevada (HCPLD – *High-Capacity PLD*) que por sua vez se dividem em subclasses (ver fig. 2.1). Descrevemos as características mais importantes destes dispositivos.

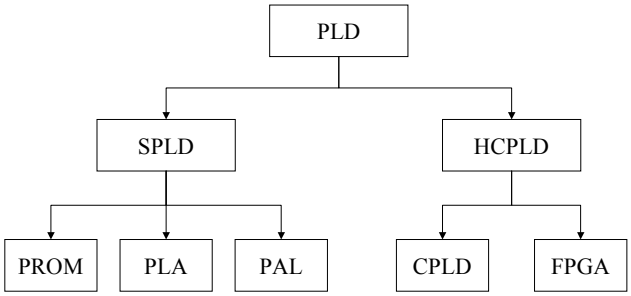


Fig. 2.1. Classificação de dispositivos lógicos programáveis.

2.2.1 PLDs elementares

Os PLDs elementares chegaram a substituir circuitos lógicos integrados de pequena escala (SSI) e média (MSI) a fim de assegurar a maior densidade de elementos [Sharma98]. PLDs elementares são compostos por dois *arrays* de portas lógicas: um *array AND* que é seguido por um *array OR*. Portanto, os SPLDs servem muito bem para a implementação de funções lógicas apresentadas em DNF (*Disjunctive Normal Form*) i.e. na soma de produtos. Um ou ambos os *arrays* são programáveis, dividindo deste modo os PLDs elementares em três categorias: PROMs, PLAs (*Programmable Logic Array*) e PALs (*Programmable Array Logic*), que descrevemos a seguir.

2.2.1.1 PROM

O primeiro tipo de PLDs utilizados para implementar circuitos lógicos, foi PROM. PROM é um *array* de células de memória que podem ser programadas de acordo com os padrões de zeros e uns definidos pelo utilizador [Sharma98]. PROMs são capazes de implementar circuitos lógicos para os quais as linhas de endereço especificam as entradas do circuito, e as linhas de dados coincidem com as saídas do mesmo. Contudo, as funções lógicas que se pretende implementar raramente contêm mais do que poucos produtos (relembramos que se trata das funções em DNF), enquanto PROM inclui o descodificador completo para as suas linhas de endereço (como está ilustrado na fig. 2.2a, uma PROM pode ser vista como um *array AND* fixo que alimenta um *array OR*

programável). Consequentemente, PROMs possuem uma arquitectura ineficiente para a realização de circuitos lógicos, e são raramente utilizadas para este fim.

2.2.1.2 PLA

PLA é um sucessor de PROM, desenvolvido especialmente para a implementação de circuitos lógicos em meados dos anos 70. Um PLA é composto por dois níveis de portas lógicas: um *array AND* programável seguido de um *array OR* programável (fig. 2.2b). Num PLA quaisquer das suas entradas (ou as suas negações) podem ser combinadas (através da função *AND*) no *array AND*. Assim, cada saída do *array AND* corresponde a um produto de entradas. O *array OR* produz as somas lógicas de quaisquer saídas do *array AND*. Os PLAs tiveram sucesso limitado dado que os *arrays* programáveis foram difíceis de fabricar e utilizar. Para além disso, os atrasos de propagação são significativos [Brown96, Sharma98].

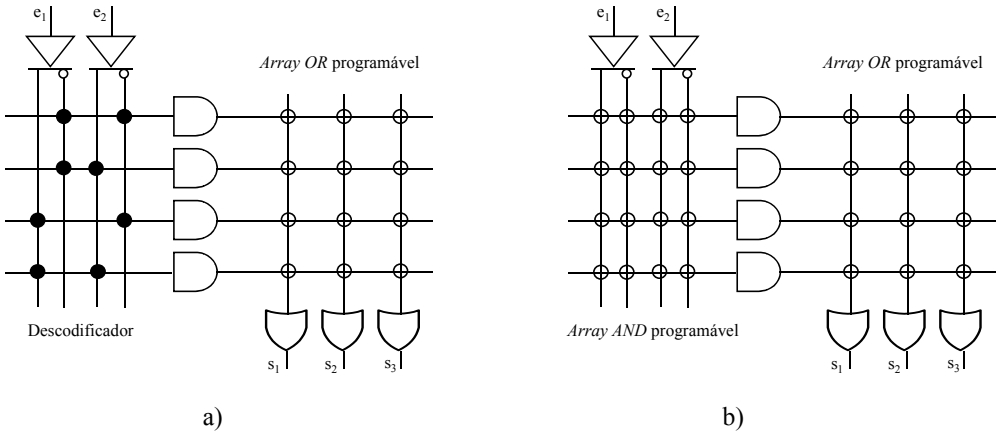


Fig. 2.2. Estrutura de PROM (a) e PLA (b). Os círculos transparentes denotam as ligações programáveis, enquanto os pretos especificam as ligações fixas.

2.2.1.3 PAL

A seguir, nos finais dos anos 70, apareceram os PALs [Sharma98]. A arquitectura PAL combina um *array AND* programável com um *array OR* fixo de tal maneira que cada saída é a soma de um conjunto específico de produtos de uma função em DNF (fig. 2.3).

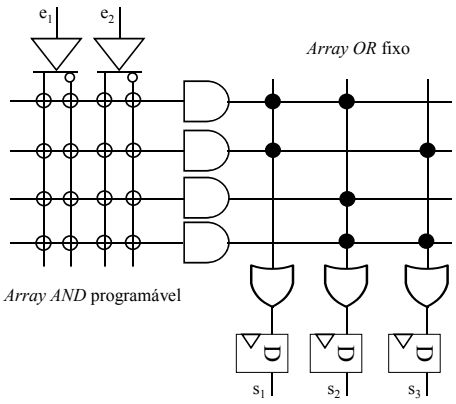


Fig. 2.3. Estrutura de PAL. Os círculos transparentes denotam as ligações programáveis, enquanto os pretos especificam as ligações fixas.

Os PALs asseguram o desempenho elevado para o caso dos circuitos lógicos simples. Para compensar a falta da generalidade (causada pelo facto do *array OR* ser fixo), são produzidas várias versões de PALs com números diferentes de entradas, saídas e dimensões variadas de portas lógicas *OR*. Normalmente, os PALs contêm *flip-flops* ligados às saídas de portas lógicas *OR* o que permite realizar circuitos sequenciais.

2.2.2 PLDs de densidade elevada

Com os avanços na tecnologia tornou-se possível produzir dispositivos com capacidade mais elevada do que a dos SPLDs. Uma das dificuldades com o aumento da capacidade da arquitectura SPLD foi causada pelo facto da estrutura de *arrays* lógicos programáveis crescer muito rapidamente com o aumento do número de entradas [Brown96]. Por essa razão, foram necessárias abordagens diferentes para aumentar a densidade lógica. Dispositivos lógicos programáveis de capacidade elevada, tais como CPLDs (*Complex Programmable Logic Devices*) e FPGAs, procuram resolver este problema com a ajuda de blocos lógicos e recursos de encaminhamento flexíveis.

2.2.2.1 CPLD

Uma maneira eficiente de construir dispositivos de capacidade maior, baseados em arquitectura SPLD, é integrar múltiplos SPLDs numa única pastilha de silício e assegurar ligações programáveis entre estes blocos. Os dispositivos programáveis que possuem esta estrutura básica são chamados CPLDs.

Um CPLD é normalmente composto por um conjunto de blocos lógicos, recursos de interligação programáveis e células de entrada/saída (ver fig. 2.4). Cada bloco lógico é semelhante a um SPLD e contém um *array AND* programável, um bloco de distribuição de produtos e um conjunto de macrocélulas. Os blocos lógicos executam a função lógica pretendida (expressa em DNF) e guardam os resultados em registos que se encontram em macrocélulas. Vários blocos lógicos comunicam entre si através dos recursos de encaminhamento longos e fixos que são interligados por uma matriz de interruptores.

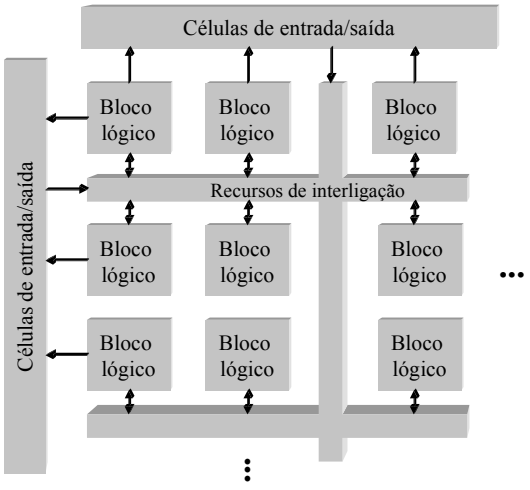


Fig. 2.4. Arquitectura típica de CPLD.

Os CPLDs podem realizar projectos bastante complexos tais como controladores gráficos, controladores LAN (*Local Area Network*), etc. Como regra geral, os circuitos que exploram portas lógicas de muitas entradas e não precisam de muitos *flip-flops*, são candidatos ideais para serem implementados em CPLDs [Brown96]. As vantagens principais dos CPLDs são a reprogramabilidade, a previsibilidade dos atrasos e o desempenho do circuito implementado.

2.2.2.2 FPGA

CPLDs asseguram um bom desempenho e densidade lógica que satisfaz uma ampla gama de aplicações. Contudo, para construir PLDs de elevada capacidade, é preciso seguir uma abordagem diferente. Os dispositivos lógicos programáveis de maior capacidade disponíveis hoje em dia, são as FPGAs. O número de portas de sistema[2] em FPGAs disponíveis comercialmente tem crescido bastante rapidamente desde a sua introdução nos meados dos anos 80, e atinge actualmente 10M em dispositivos da Xilinx [Xilinx]. As FPGAs combinam a estrutura estendível de interligações de MPGAs com a programabilidade de CPLDs, possibilitando deste modo a implementação de lógica multi-nível [Chang99]. A arquitectura típica de uma FPGA está representada na fig. 2.5.

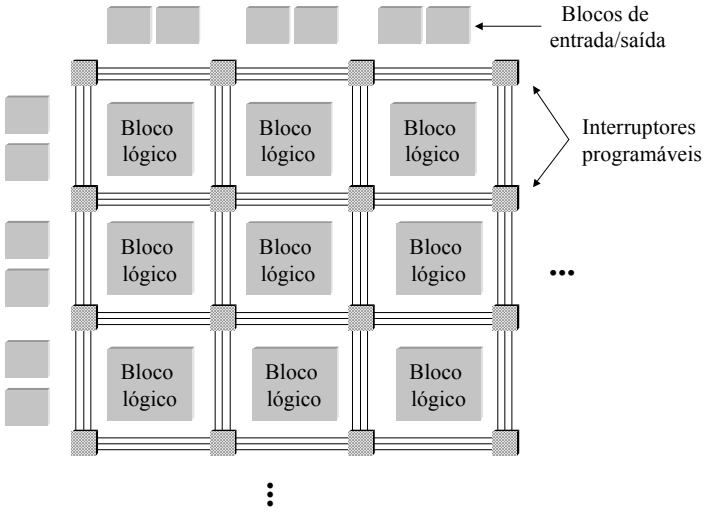


Fig. 2.5. Arquitectura típica de FPGA.

Uma FPGA inclui um *array* de blocos lógicos interligados por recursos de encaminhamento e cercado por um conjunto de blocos de entrada/saída, sendo todos estes componentes programáveis pelo utilizador. Os blocos lógicos contêm elementos combinatórios e sequenciais possibilitando a implementação de funções lógicas bem como de circuitos sequenciais. Os recursos de encaminhamento incluem segmentos de pistas de ligação pré-fabricadas e interruptores programáveis. Um circuito lógico é implementado em FPGA ao distribuir a lógica entre os blocos individuais e interligá-los posteriormente com os interruptores programáveis. É de notar que os atrasos resultantes são fortemente influenciados pela distribuição actual da lógica e pela estrutura de encaminhamento. Portanto, o desempenho de circuitos mapeados em FPGA depende bastante da eficácia das ferramentas CAD (*Computer-Aided Design*) utilizadas para a implementação.

[2] Esta medida não é muito adequada para expressar exactamente a capacidade lógica, mas é utilizada amplamente por vários vendedores de FPGAs.

Uma das maiores vantagens das FPGAs é a sua arquitectura flexível que serve muito bem para uma ampla gama de aplicações. Estas aplicações incluem implementação de controladores de dispositivos, circuitos de codificação, lógica arbitrária, prototipagem e emulação de sistemas, etc. As FPGAs recentes passaram a incorporar várias estruturas heterogéneas tais como blocos de memória, o que possibilita a implementação de sistemas completos num único encapsulamento. As FPGAs deram também origem a um novo paradigma de computação, a computação reconfigurável, à qual dedicamos o capítulo 3.

2.3 Tecnologias de ligações programáveis pelo utilizador

Consideremos as principais tecnologias utilizadas para implementar ligações e elementos programáveis em PLDs.

O primeiro tipo de ligações desenvolvido foi o *fusível*. No estado original do dispositivo todos os fusíveis estão intactos. Uma configuração particular obtém-se ao queimar os fusíveis ao longo dos caminhos que devem ser removidos. Para tal é necessário aplicar uma tensão relativamente alta fazendo com que a corrente corte a ligação apropriada.

A tecnologia de programação baseada em antifusíveis é oposta aos fusíveis. Um *antifusível* consiste em dois condutores separados por um material com resistência alta. Antes de programado o antifusível representa um caminho aberto. Ao aplicar nos condutores uma tensão adequada, o material funde e passa a ter resistência baixa. Como resultado, o caminho respectivo fica ligado. As maiores vantagens dos antifusíveis são o seu tamanho pequeno e a resistência e capacitância relativamente baixas [Chang99]. Estas características permitem obter uma maior densidade de interruptores o que pode aliviar as restrições de encaminhamento. Contudo, os antifusíveis possuem processo de fabricação relativamente complexo e não são reprogramáveis.

As duas tecnologias de programação descritas acima são permanentes, i.e. os dispositivos não podem ser reprogramados dado que as alterações físicas são irreversíveis. Consequentemente, não há nenhuma possibilidade de recuperar o dispositivo se se verificar que a programação é incorrecta ou se for necessário introduzir alguma modificação no circuito.

A tecnologia SRAM recorre ao uso de células de memória estática para controlar elementos de ligação e configurar os blocos lógicos. Por exemplo, o bit de uma célula SRAM pode controlar a porta de transmissão dum transístor. Dado que o conteúdo da SRAM é modificável, o dispositivo baseado em SRAM é facilmente reprogramável. É de salientar que a memória estática é volátil, por isso a configuração perde-se ao cortar a alimentação. Para além de controlar transístores, a tecnologia SRAM utiliza-se amplamente para implementar LUTs (*Look-Up Tables*). As maiores vantagens da tecnologia SRAM são a sequência de fabricação simples e a reprogramação rápida [Chang99]. Contudo, as maiores desvantagens desta tecnologia são as dimensões físicas grandes e a volatilidade. A última característica implica o uso de uma memória externa persistente para guardar a configuração respectiva.

Para gerir a comutação de transístores recorre-se frequentemente a tecnologias baseadas em guarda de carga eléctrica em portas flutuantes. A porta flutuante está electricamente isolada do resto do transístor, portanto são necessárias técnicas especiais que permitem mover electrões para esta.

Dado que se pode remover a carga guardada, torna-se possível reprogramar os estados de transístores. As três tecnologias que se utilizam para o efeito são EPROM (*Erasable PROM*), EEPROM (*Electrically Erasable PROM*) e *Flash*. As suas maiores vantagens são a reprogramabilidade, testabilidade e não-volatilidade. Contudo, estas tecnologias caracterizam-se por processo de fabricação bastante complexo.

Para reprogramar EPROM é necessário apagar todos os dados escritos previamente (isto faz-se com a ajuda de uma fonte de luz ultravioleta). Para carregar a porta flutuante na tecnologia EPROM recorre-se à técnica conhecida sob o nome de *Channel Hot Electron injection*.

EEPROM é mais flexível dado que permite a reprogramação de células individuais. Esta flexibilidade deve-se à célula de memória mais complexa, o que aumenta o seu custo. Para carregar e descarregar a porta flutuante do transístor utiliza-se o efeito de *tunneling*.

Semelhante a EEPROM, a *Flash* também pode ser apagada e reprogramada electricamente. Contudo, o tamanho de uma célula *Flash* é mais pequeno que o da EEPROM permitindo deste modo uma maior densidade.

A tabela 2.2 sumaria as características mais importantes das tecnologias de programação descritas acima. Em CPLDs são normalmente utilizadas as tecnologias EPROM, EEPROM e *Flash*, enquanto as FPGAs recorrem a antifusíveis e SRAM.

Tabela 2.2. Características principais das tecnologias de programação utilizadas em PLDs.

nome	**volátil**	**reprogramável**
Fusível	não	não
Antifusível	não	não
SRAM	sim	sim
EPROM	não	sim (com equipamento especial)
EEPROM	não	sim
Flash	não	sim

É de salientar que de acordo com o tipo de ligações programáveis utilizadas, todos os PLDs podem ser divididos nas três categorias seguintes:

- *Dispositivos de programação única* depois de configurados não podem ser reprogramados. Todos os PLDs que se baseiam em fusíveis e antifusíveis são programáveis uma só vez. Os PLDs que utilizam a tecnologia EPROM, quando fornecidos num encapsulamento plástico (sem a "janela" especial que permite apagar os dados com a ajuda de luz ultravioleta), também pertencem a esta categoria.
- *Dispositivos reprogramáveis em sistema* podem ser reprogramados depois de serem instalados numa placa junto com quaisquer outros componentes. Esta característica reduz o tempo de preparação para a fabricação e permite depurar o sistema e potencialmente aperfeiçoá-lo no futuro. PLDs deste tipo baseiam-se em tecnologias SRAM, EEPROM e *Flash*. É de notar que todos os PLDs da tecnologia SRAM obviamente pertencem a esta categoria dado que precisam de ser configurados depois de cada corte de alimentação. A

característica de reprogramação em sistema requer a presença de circuitos especiais que possibilitam a reconfiguração. A comunicação com estes circuitos efectua-se através de interface *standard* JTAG (*Joint Test Action Group*) ou via qualquer outra interface específica do vendedor do PLD.

- *Dispositivos reprogramáveis num programador* não podem ser reprogramados enquanto instalados numa placa. Isto explica-se pelo facto de estes carecerem de circuitos especiais que possibilitam a reconfiguração. A esta categoria pertencem alguns PLDs que se baseiam em tecnologias EPROM, EEPROM e *Flash*.

2.4 Projecto assistido por computador

Para desenvolver circuitos e implementá-los em HCPLDs, é essencial recorrer ao uso de programas de desenvolvimento assistido por computador (CAD). Um sistema CAD típico para CPLDs inclui ferramentas de software para executar a sequência de tarefas representada na fig. 2.6.

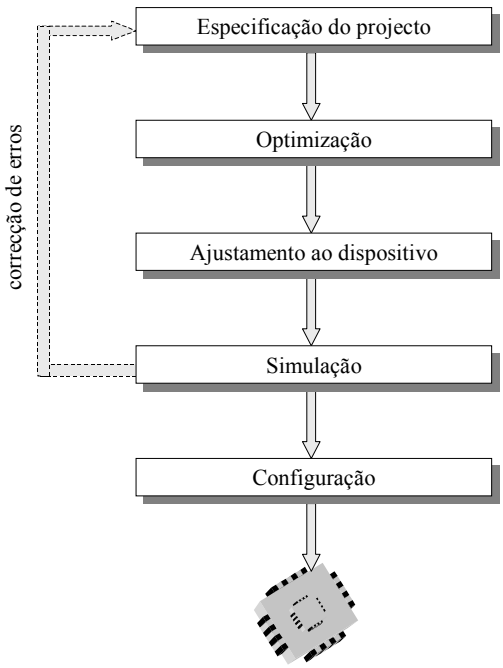


Fig. 2.6. Sequência de passos necessários para projectar e implementar um circuito em CPLD.

A especificação do projecto envolve a descrição das funções que se pretende implementar em hardware. Depois, os algoritmos especializados que permitem compilar e optimizar os circuitos, e a seguir, ajustá-los às características dum dado CPLD, são invocados. A simulação serve para detectar os erros possíveis e, se for preciso, voltar ao início para os corrigir. Caso não haja erros, é gerado um ficheiro de configuração. Normalmente, todos os passos de optimização, ajustamento e geração de configuração são executados de uma maneira automática. O método de configuração depende da tecnologia de programação utilizada.

2.4.1 Sequência de projecto para FPGA

No caso de FPGAs, o desenvolvimento de um circuito inclui três fases principais: especificação, implementação e verificação (ver fig. 2.7). Descrevemos cada uma destas fases em mais detalhe.

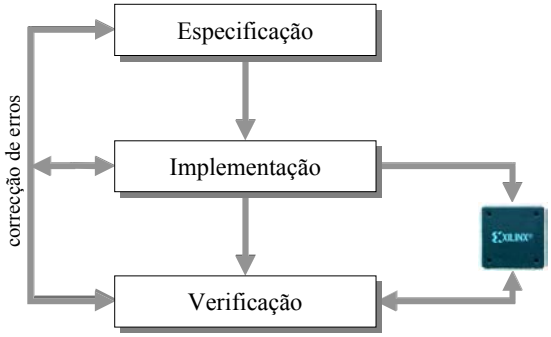


Fig. 2.7. Sequência de passos necessários para projectar e implementar um circuito em FPGA.

2.4.1.1 Especificação

A especificação do circuito a ser implementado numa FPGA, pode ser feita com a ajuda de linguagens de descrição de hardware (VHDL - *VHSIC Hardware Description Language*, Verilog, etc.) ou através de criação de um diagrama esquemático com uma ferramenta gráfica de projecto assistido por computador. Os blocos básicos utilizados no diagrama esquemático são descritos numa HDL (*Hardware Description Language*), extraídos das bibliotecas específicas para cada família de FPGA ou criados pelos geradores de componentes parametrizáveis (tais como *LogiBLOX* e *CORE Generator* da Xilinx). A utilização de componentes IP (*Intellectual Property*) aumenta significativamente a produtividade pois possibilita trabalhar a níveis de abstracção mais altos. Existem também ferramentas gráficas que permitem descrever o funcionamento em forma de diagramas de transições de estados.

Recentemente, tornou-se possível especificar a funcionalidade de circuitos numa linguagem de alto nível, tal como Handel-C [Handel-C] ou SystemC [SystemC]. Por exemplo, SystemC é uma biblioteca de classes que possibilita a modelação de sistemas digitais recorrendo à tecnologia orientada por objectos. O compilador *CoCentric* da Synopsys sintetiza a descrição em SystemC na descrição RTL (*Register Transfer Level*) em VHDL ou Verilog [CoCentric]. O ambiente DK1 da Celoxica permite a compilação directa da descrição em Handel-C para VHDL estrutural ou para EDIF (*Electronic Design Interchange Format*). Note-se que os circuitos descritos a alto nível e criados com a ajuda de ferramentas automáticas ocupam normalmente uma maior área e são mais lentos que os projectados à mão. Mas as ferramentas automáticas facilitam e aceleram significativamente o desenvolvimento de sistemas digitais. Para além disso, as linguagens de alto nível geram descrições mais portáveis possibilitando a implementação mais simples em várias arquitecturas de FPGAs.

2.4.1.2 Implementação

No caso das FPGAs o processo de implementação incorpora três etapas. Primeiro, efectua-se o *mapeamento* das estruturas do projecto nos blocos lógicos existentes na FPGA. A seguir, os blocos

resultantes devem ser *colocados* no hardware reconfigurável. Cada um destes blocos fica atribuído a um local específico da FPGA, procurando os algoritmos de colocação que este fique junto daqueles blocos lógicos com os quais comunica.

Uma das técnicas que é utilizada para aliviar os custos de colocação (com o aumento de capacidade de FPGAs esta operação passa a consumir muito tempo) é o *floorplanning* [Compton02]. O algoritmo de *floorplanning* primeiro organiza em grupos aqueles blocos lógicos que requerem uma comunicação muito intensiva. Os grupos são colocados em FPGA e, a seguir, efectua-se a colocação detalhada de blocos lógicos individuais dentro dos limites atribuídos ao grupo. Assim, a colocação completa reduz-se a um conjunto de problemas locais de dimensões menores facilitando deste modo a resolução da tarefa.

Finalmente, realiza-se o *encaminhamento* do circuito que interliga os vários componentes reconfiguráveis. Os recursos de encaminhamento disponíveis são sempre limitados, portanto o objectivo é minimizar o número de segmentos de pistas de ligação utilizados. Uma colocação eficiente é essencial para que esta fase seja bem sucedida porque caso os componentes interligados fiquem colocados longe uns dos outros, as ligações respectivas vão precisar de muitos recursos de encaminhamento e ser mais lentas.

Se um sistema utilizar mais que uma FPGA, a sequência de implementação torna-se ainda mais difícil. Neste caso, o projecto deve ser distribuído entre várias FPGAs. Isto é conseguido ao colocar porções menos interligadas do circuito em FPGAs separadas. Depois, o encaminhamento global determina as ligações entre as FPGAs que é seguido pelo encaminhamento detalhado que atribui segmentos de pistas de ligação a cada sinal. A comunicação entre FPGAs distintas deve ser minimizada para reduzir o número de ligações com atrasos grandes. Para ultrapassar restrições no número de pinos utilizam-se frequentemente as técnicas de multiplexagem no tempo [Babb93].

2.4.1.3 Verificação

Existem três tipos básicos de verificação do projecto para FPGAs. Primeiro, realiza-se a simulação funcional que ocorre durante a fase de especificação. O objectivo da simulação funcional é verificar a funcionalidade lógica do circuito.

Depois das fases de mapeamento, colocação e encaminhamento, efectua-se a simulação temporal que já pode ter em conta todos os atrasos lógicos e os de encaminhamento. Para que isto seja possível, é preciso efectuar a re-anotação da informação física ao projecto lógico.

Finalmente, depois de configurar a FPGA, é possível verificar a implementação física do circuito. Para tal recorre-se normalmente ao uso de um analisador lógico, encaminhando os sinais que se pretende verificar para os pinos externos da FPGA.

Uma metodologia diferente que surgiu recentemente, consiste em incorporar no projecto um núcleo do analisador lógico interno e utilizar os blocos de memória embutidos para guardar valores dos sinais necessários que são posteriormente transferidos para um computador para os analisar. Este método possui algumas vantagens relativamente ao anterior porque a comunicação com o computador efectua-se via interface JTAG não ocupando deste modo os pinos adicionais da FPGA. Um exemplo desta metodologia são as ferramentas *ChipScope Pro* da Xilinx [Przybus02, Hernandez02].

2.5 Classificação de FPGAs

Várias arquitecturas de FPGAs são classificadas de acordo com a granulosidade (i.e. tamanho e complexidade) dos seus blocos lógicos e com a estrutura de ligações programáveis [Sharma98].

2.5.1 Blocos lógicos

Os blocos lógicos da FPGA devem ser capazes de implementar várias funções lógicas e podem ter complexidade diversa. Normalmente, as arquitecturas de FPGAs são baseadas nos elementos seguintes: pares de transístores, portas lógicas simples, multiplexadores e LUTs. Os blocos lógicos, compostos por estes elementos, diferem bastante em termos do tamanho e das capacidades de implementação de funções lógicas. Estas diferenças são normalmente classificadas por *granulosidade*.

A granulosidade de um bloco lógico pode ser definida de maneiras diversas, tais como número de funções booleanas que este pode implementar, número de portas lógicas *NAND* equivalentes, número total de transístores, ou número de entradas/saídas [Sharma98]. De acordo com estas características, as arquitecturas de blocos lógicos são divididas em duas categorias: as de *granulosidade fina* e as de *granulosidade grossa*.

Os blocos lógicos de *granulosidade fina* podem implementar funções elementares servindo deste modo muito bem às manipulações ao nível de bits individuais e atingindo um grande nível de utilização de recursos lógicos disponíveis. Uma desvantagem de blocos lógicos finos é que estes requerem muitos segmentos de pistas de ligação e interruptores programáveis. É de notar também que as FPGAs de granulosidade fina possuem muitos pontos de configuração e precisam por essa razão de mais bits para serem reconfiguradas [Compton02].

Os blocos lógicos de *granulosidade grossa* têm normalmente muitas entradas e servem bem à implementação de funções complexas. Dado que tais blocos lógicos são optimizados para funções lógicas mais complexas, estas são tipicamente executadas mais rapidamente e consomem menos área do que um conjunto de blocos lógicos elementares interligados de maneira adequada. Contudo, se for necessário realizar operações lógicas elementares, a arquitectura de granulosidade grossa vai sofrer de uma baixa utilização dos recursos disponíveis. É de notar também que os blocos lógicos, quando são suficientemente complexos, requerem bastantes recursos locais de encaminhamento que tendem a explodir com o aumento da granulosidade [Betz98].

É de salientar que a maioria de FPGAs disponíveis no mercado utilizam blocos lógicos de granulosidade *média* a fim de minimizar as desvantagens inerentes aos dois casos extremos descritos acima.

2.5.2 Recursos de encaminhamento

Os recursos de encaminhamento ocupam uma área da FPGA que é muito maior que a usada pelos recursos lógicos [Compton02]. A arquitectura de encaminhamento de FPGAs define como os interruptores programáveis e os segmentos de pistas de ligação de comprimentos variáveis são utilizados a fim de interligar os blocos lógicos. O número e a distribuição de segmentos de pistas

incorporados afecta a densidade e o desempenho atingíveis na FPGA. Caso este número seja inadequado só uma fracção dos blocos lógicos poderá ser aproveitada. Contudo, se o número de segmentos for excessivo, isto pode aumentar o tamanho da FPGA e resultar em eficiência reduzida de utilização do silício.

A estrutura de encaminhamento da FPGA deve ser capaz de acomodar todas as ligações de uma aplicação típica e assegurar a velocidade de funcionamento adequada. Um factor importante no desempenho de FPGAs é o atraso de propagação através dos recursos de encaminhamento atribuídos, dado que algumas ligações requerem caminhos mais compridos que as outras. Para além disso, qualquer interruptor programável (antifusível, SRAM ou transístor EPROM) possui também uma resistência e capacitância que provocam atrasos adicionais.

Existem dois métodos principais de atribuir os recursos de encaminhamento: *segmentado* e *hierárquico* [Compton02]. No encaminhamento segmentado as pistas de ligação curtas servem para as comunicações locais. Estas podem ser interligadas com a ajuda de interruptores a fim de estabelecer ligações longas. Para além disso, existem normalmente pistas de ligação mais compridas que servem para encaminhar sinais a distâncias longas sem ter de passar por interruptores.

Na estrutura de encaminhamento hierárquico os blocos lógicos são organizados em grupos. Primeiro, efectua-se a interligação de blocos pertencentes a um grupo com a ajuda de pistas curtas. A seguir, utilizam-se pistas compridas para interligar vários grupos de blocos lógicos. Esta estratégia pode ser repetida várias vezes através da organização de uma série de grupos em conjuntos maiores e da sua interligação posterior.

2.5.3 Estruturas heterogéneas

A fim de assegurar melhor desempenho e flexibilidade na computação, as FPGAs recentes passaram a incluir estruturas heterogéneas, i.e. para além dos componentes típicos que fazem parte de qualquer FPGA, utilizam-se blocos específicos.

Por exemplo, é bastante difícil implementar em FPGA a operação de multiplicação de uma maneira eficiente. Portanto, foram criadas unidades de multiplicação optimizadas e embutidas na estrutura de algumas FPGAs.

A maioria das arquitecturas actuais incluem blocos de memória dedicados que servem para guardar dados que são utilizados frequentemente. Para além disso, com a ajuda dos blocos de memória embutidos pode-se emular expressões lógicas de um grande número de entradas. Uma outra aplicação de blocos de memória é para depuração dos circuitos desenvolvidos em *run-time* conforme descrito na secção 2.4.1.3.

As FPGAs recentes incluem também transmissores/receptores de alto débito (múltiplos Gbits) e núcleos de processadores o que aumenta a gama de aplicações possíveis nas áreas de telecomunicações e de processamento de sinal e imagem.

Dado que as FPGAs são uma tecnologia relativamente recente, as suas arquitecturas estão sujeitas a uma evolução constante [Actel, Altera, Atmel, Lattice, QuickLogic, Xilinx]. Algumas das características das FPGAs disponíveis comercialmente estão sumariadas na tabela 2.3.

Tabela 2.3. Características principais de FPGAs disponíveis comercialmente.

Companhia	**Família**	**Tecnologia de programação**	**Granulosidade**	**Densidade (portas de sistema)**	**Estruturas heterogéneas**
Actel	Axcelerator	antifusível	fina	até 2M	SRAM embutida (até 330 Kbits)
	ProASIC^{PLUS}	*Flash*	fina	até 1M	SRAM embutida (até 198 Kbits)
Altera	Stratix	SRAM	média	até 114K de elementos lógicos[3]	SRAM (até 10 Mbits), blocos DSP
Atmel	AT40K	SRAM	média	até 50K	--
Lattice	ORCA	SRAM	grossa	até 900K	RAM embutida (até 148 Kbits)
QuickLogic	pASIC3	antifusível	média	até 75K	--
	Eclipse-II	antifusível	média	até 370K	SRAM (até 55Kbits) e unidades computacionais embutidas
Xilinx	XC4000XL	SRAM	média	até 180K	--
	Virtex-EM	SRAM	média	até 3M	SRAM embutida (até 1120 Kbits)
	Virtex-II	SRAM	média	até 8M	SRAM (até 3 Mbits) e multiplicadores embutidos
	Virtex-II Pro	SRAM	média	--- [4]	SRAM (até 10 Mbits), multiplicadores, processadores PowerPC, transmissores/receptores de múltiplos Gbits embutidos

2.6 FPGAs da Xilinx

A Xilinx produz várias séries de FPGAs baseadas em SRAM [Xilinx]. Dispositivos das séries XC4000 e Virtex (nomeadamente, as FPGAs XC4010XL e XCV812E) foram utilizados no âmbito deste trabalho, portanto descrevemos as famílias XC4000XL e Virtex-EM em mais detalhe.

[3] Dados sobre as portas de sistema estão indisponíveis
[4] Por causa de existência de várias estruturas heterogéneas não faz sentido expressar a densidade em termos de portas de sistemas

2.6.1 Família XC4000XL

A família XC4000XL inclui 11 dispositivos (ver tabela 2.4). A arquitectura XC4000XL consiste num *array* de CLBs (*Configurable Logic Blocks*), interligados por uma hierarquia de recursos de encaminhamento e cercado na periferia pelos blocos de entrada/saída programáveis (ver fig. 2.8). As FPGAs são personalizadas através do carregamento dos dados de configuração em células de memória internas. A FPGA pode ler activamente os seus dados de configuração de uma PROM externa, ou os dados de configuração podem ser carregados na FPGA por um dispositivo externo.

A arquitectura XC4010XL contém quatro descodificadores programáveis localizados em cada lado do dispositivo, cujas saídas podem ser encaminhadas quer para os pinos da FPGA quer para as entradas de CLBs (possibilitando deste modo a formação de funções do tipo PAL). A FPGA inclui também um oscilador interno de 8 MHz que pode ser utilizado quer como fonte de relógio nos projectos do utilizador quer para operações tais como inicialização da memória de configuração, fonte de relógio para a configuração, etc.

Tabela 2.4. Características principais de FPGAs da família XC4000XL. O dispositivo utilizado no trabalho está destacado com os caracteres em negrito.

FPGA	Portas de sistema	*Array* de CLBs	Entradas/saídas de utilizador	RAM distribuída (bits)
XC4002XL	até 3K	8 × 8	64	2048
XC4005XL	até 9K	14 × 14	112	6272
XC4010XL	**até 20K**	**20 × 20**	**160**	**12800**
XC4013XL	até 30K	24 × 24	192	18432
XC4020XL	até 40K	28 × 28	224	25088
XC4028XL	até 50K	32 × 32	256	32768
XC4036XL	até 65K	36 × 36	288	41472
XC4044XL	até 80K	40 × 40	320	51200
XC4052XL	até 100K	44 × 44	352	61952
XC4062XL	até 130K	48 × 48	384	73728
XC4085XL	até 180K	56 × 56	448	100352

2.6.1.1 Blocos principais da família XC4000XL

Um bloco lógico (CLB) da família XC4000XL inclui duas LUTs de quatro entradas, referenciadas como geradores de funções *F* e *G*, e uma LUT de três entradas (gerador de função *H*) que pode receber dados quer dos blocos *F* e *G* quer de fora do CLB (ver o canto inferior direito da fig. 2.8.). Sendo assim, um CLB é capaz de realizar qualquer função lógica de cinco variáveis, ou algumas funções lógicas com nove argumentos no máximo. Para além disso, cada CLB contém dois elementos de memória configuráveis em *flip-flops* ou *latches*, que podem ser utilizados quer para guardar as saídas das LUTs, quer de uma maneira independente. Cada CLB possui 13 entradas e 4 saídas.

Os geradores de funções *F* e *G* podem ser configurados como um *array* de células de memória possibilitando deste modo implementar num CLB RAM/ROM com as seguintes configurações possíveis: 16×2 ou 32×1 *single-port*, ou 16×1 *dual-port* (ver o canto superior direito da fig. 2.8.). Como resultado, as FPGAs da família XC4000XL contêm até 98 Kbits de memória distribuída, i.e. construída com base em CLBs (ver tabela 2.4).

Cada gerador de função *F* e *G* contém lógica dedicada para a geração rápida de sinais de *carry*. Estes sinais são transmitidos verticalmente para os geradores de função no CLB adjacente pela cadeia de *carry* que é independente dos recursos de encaminhamento gerais. As cadeias de *carry* aumentam a eficiência e o desempenho na implementação de somadores, subtractores, acumuladores, comparadores e contadores [Xilinx00].

Os blocos de entrada/saída (IOB - *Input/Output Blocks*) implementam a interface entre os pinos externos e a lógica interna (ver o canto inferior esquerdo da fig. 2.8). Cada IOB controla um pino e pode ser configurado para sinais de entrada, de saída ou bidireccionais.

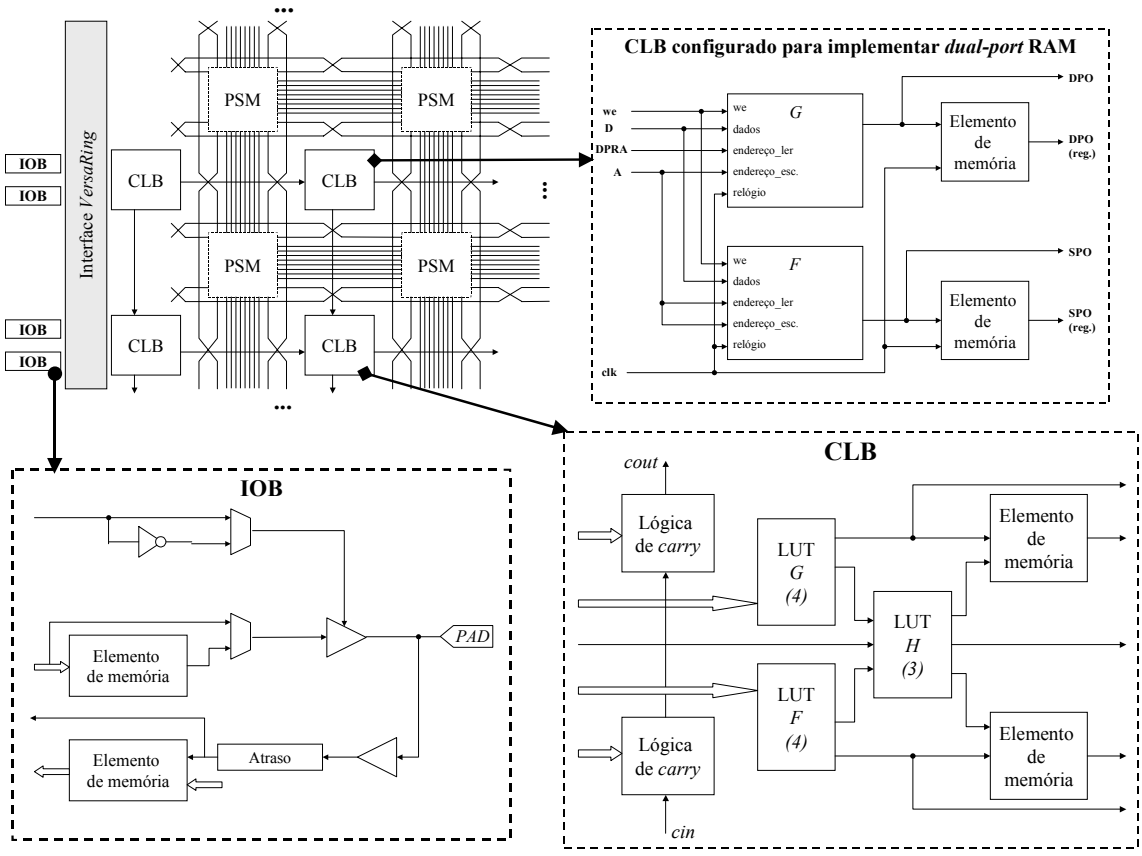


Fig. 2.8. Arquitectura das FPGAs da família XC4000XL.

2.6.1.2 Recursos de encaminhamento da família XC4000XL

A estrutura de encaminhamento consiste de segmentos de pistas de metal com pontos de interrupção programáveis e matrizes de interruptores [Xilinx00]. Na arquitectura XC4000XL há vários tipos de interligações [Xilinx00]:

- *encaminhamento de CLBs* está associado com cada linha e coluna no *array* de CLBs;

- *encaminhamento de IOBs* forma o anel *VersaRing* à volta do *array* de CLBs que liga as entradas/saídas com os blocos lógicos internos;
- *encaminhamento global* consiste em redes dedicadas que distribuem sinais de relógio e outros sinais de grande *fanout* por toda a FPGA.

As linhas de interligação são classificadas de acordo com o comprimento relativo dos seus segmentos nos cinco tipos seguintes: linhas unitárias, duplas, quadruplas, de comprimento oito e longas. Para além disso, há ligações directas que permitem a propagação rápida de dados entre os CLBs adjacentes e entre os CLBs e IOBs.

As linhas horizontais e verticais unitárias e duplas cruzam-se em blocos PSM (*Programmable Switch Matrix*). Cada PSM consiste em transístores de transmissão programáveis utilizados para estabelecer ligações entre as linhas. As linhas unitárias possibilitam o encaminhamento eficiente de sinais entre blocos adjacentes. A cada CLB estão associadas oito linhas unitárias horizontais e oito linhas verticais (ver o canto superior esquerdo da fig. 2.8). Estas linhas são ligadas às PSMs localizadas em cada linha e coluna de CLBs. As linhas duplas atravessam dois CLBs antes de entrar numa PSM. A cada CLB estão associadas quatro linhas duplas horizontais e quatro linhas verticais (ver o canto superior esquerdo da fig. 2.8). Estas linhas servem para encaminhar sinais a distâncias intermédias. Há doze linhas quadruplas horizontais e doze linhas verticais por cada linha e coluna de CLBs. Estas linhas são interligadas via PSMs com *buffers* distribuídas a distância de quatro CLBs. Devido ao uso de PSM com *buffers*, as linhas quadruplas são muito rápidas e servem para encaminhar sinais a distâncias longas. As 8 linhas de comprimento oito estão localizadas entre o *array* de CLBs e o anel de IOBs. Ao passar por cada oito CLBs estas linhas entram num *buffer* programável. As linhas longas atravessam toda a FPGA horizontal e verticalmente e servem para encaminhar sinais com grande *fanout*.

As redes globais distribuem sinais de relógio e outros sinais de controlo de grande *fanout* por toda a FPGA. Estas incluem 8 linhas longas verticais em cada coluna de CLBs que complementam as linhas longas utilizadas para o encaminhamento *standard*. As linhas das redes globais são ligadas aos *buffers* globais especiais.

2.6.2 Família Virtex-EM

A família Virtex-EM (Virtex-E *Extended Memory*) é uma extensão da arquitectura Virtex-E e inclui dois dispositivos (ver tabela 2.5). A arquitectura consiste num array de CLBs e de blocos de memória embutidos, cercado por blocos de entrada/saída programáveis e DLLs (*Delay-Locked Loops*), sendo todos estes interligados por uma hierarquia de recursos de encaminhamento (ver fig. 2.9). As FPGAs da família Virtex-EM contêm até 4704 CLBs, 294 Kbits de RAM distribuída (i.e. construída com base em CLBs) e até 1120 Kbits em blocos de memória dedicados.

2.6.2.1 Blocos principais da família Virtex-EM

O bloco lógico básico da arquitectura Virtex-EM é a célula lógica (LC – *Logic Cell*). Cada LC inclui uma LUT de quatro entradas, a lógica de *carry* e um elemento de memória. A saída da LUT pode controlar a saída do CLB bem como a entrada do elemento de memória. Cada CLB contém

quatro LCs organizadas em duas *slices* (ver o canto inferior direito da fig. 2.9). Um CLB é capaz de implementar qualquer função com seis argumentos no máximo, um multiplexador 8:1 ou algumas funções com até 19 entradas. Cada LUT permite construir um registo de deslocamento (até 16 bits). Para a execução de operações aritméticas, bem como para a realização de funções lógicas complexas, em cada CLB existem duas cadeias de *carry*. Cada *slice* possibilita a implementação de um somador completo de 2 bits. Para além de realização de funções lógicas, as LUTs podem ser configuradas em RAM. É possível combinar duas LUTs numa *slice* a fim de construir RAM síncrona com as seguintes configurações possíveis: 16×2 ou 32×1 *single-port*, ou 16×2 *dual-port*. Os elementos de memória incluídos em LCs podem ser configurados para funcionarem quer como *flip-flops* do tipo *D* activos na transição ascendente (*edge-triggered*) quer como *latches* activas ao nível (*level-sensitive*). Os CLBs incluem também dois *drivers* do tipo *tri-state* para ligações com os barramentos da FPGA.

Tabela 2.5. Características principais de FPGAs da família Virtex-EM. O dispositivo utilizado no trabalho está destacado com os caracteres em negrito.

FPGA	Portas de sistema	*Array* de CLBs	Entradas/saídas de utilizador	RAM distribuída (Kbits)	RAM embutida (Kbits)
XCV405E	até 1.307K	40 × 60	404	150	560
XCV812E	**até 3.062K**	**56 × 84**	**556**	**294**	**1120**

Um bloco de entrada/saída (ver o canto inferior esquerdo da fig. 2.9) contém três elementos de memória configuráveis quer em *flip-flops* do tipo *D* activos na transição ascendente (*edge-triggered*) quer em *latches* activas ao nível (*level-sensitive*). Os IOBs suportam uma ampla gama de normas de I/O (*Input/Output*) existentes [Xilinx_025]. Os sinais de entrada são encaminhados directamente para a lógica interna ou passam primeiro através do *flip-flop*. Na entrada do *flip-flop* fica um elemento de atraso opcional. Os sinais de saída passam através de um *flip-flop* ou provêm directamente da lógica interna e são encaminhados por um *buffer* do tipo *tri-state*.

Os blocos de memória *SelectRAM* que complementam a memória distribuída (implementada em CLBs), são organizados em colunas e inseridos em cada quarta coluna de CLBs. Cada bloco de memória (ver o topo da fig. 2.9) representa uma RAM *dual-port* de 4096 bits com os sinais de controlo independentes para ambas as portas. Cada bloco pode implementar uma RAM com as seguintes configurações: 4096×1, 2048×2, 1024×4, 512×8, 256×16, sendo a largura do barramento de dados seleccionada independentemente para cada porta.

Existem 8 DLLs localizados em extremidades opostas da FPGA. Os DLLs (ver o canto superior direito da fig. 2.9) observam os sinais de relógio (os de entrada e distribuídos) e ajustam automaticamente o elemento de atraso do relógio. Um atraso adicional é introduzido de tal maneira que as transições do relógio atingem os *flip-flops* internos exactamente um período depois destas aparecerem nas entradas. Isto permite eliminar o atraso de distribuição do relógio ao assegurar que as transições do relógio atingem os *flip-flops* internos de uma maneira síncrona com as transições do relógio nas entradas externas. Para além disso, os DLLs implementam um número de funções úteis tais como o deslocamento das fases do sinal de relógio de entrada, a duplicação ou a divisão do sinal de relógio, etc.

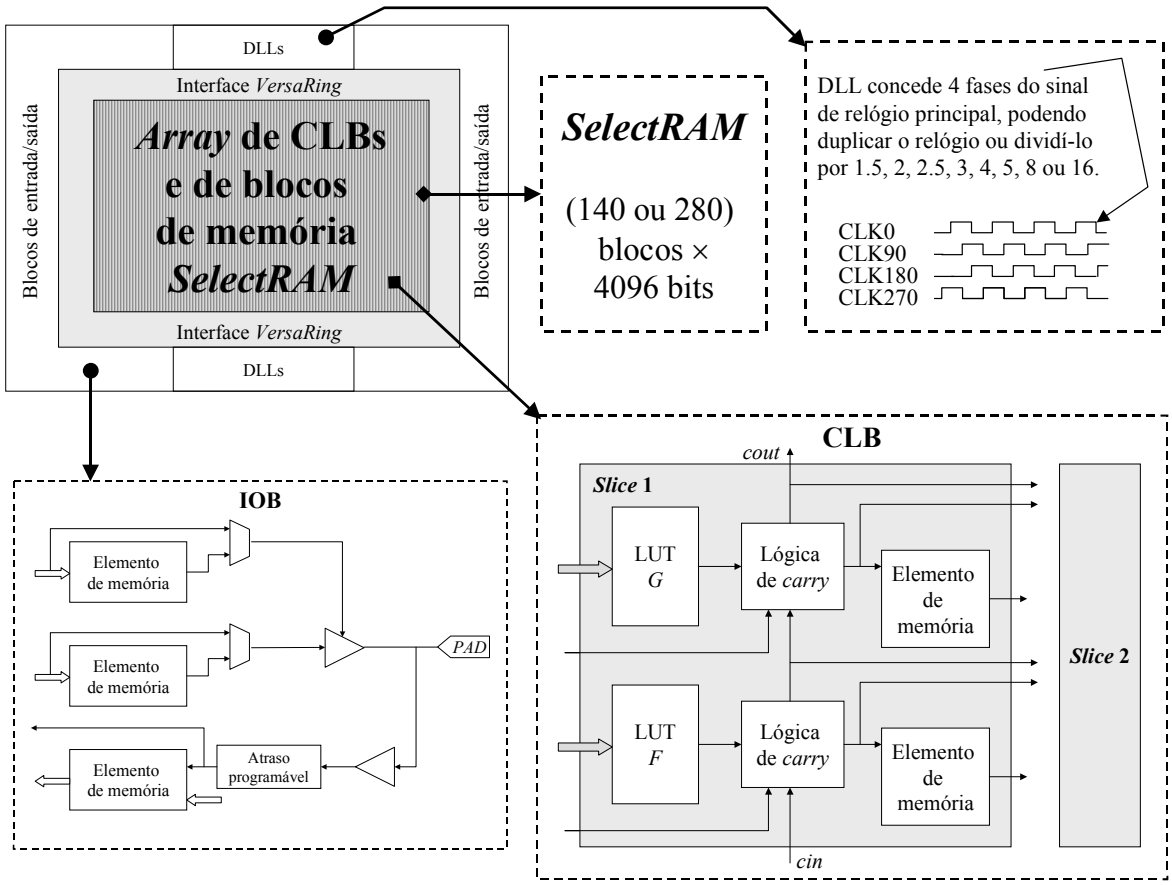


Fig. 2.9. Arquitectura das FPGAs da família Virtex-EM.

2.6.2.2 Recursos de encaminhamento da família Virtex-EM

Na arquitectura Virtex-EM há recursos de encaminhamento locais, gerais, de entrada/saída, dedicados e de relógio. Os recursos locais asseguram as ligações entre LUTs, *flip-flops* e a matriz de encaminhamento geral (GRM – *General Routing Matrix*), as ligações de realimentação dentro dum CLB, e também as ligações directas entre os CLBs adjacentes horizontalmente, ultrapassando deste modo o atraso inerente à GRM. A GRM é uma matriz de interruptores adjacente a cada CLB através da qual são interligados os recursos de encaminhamento horizontais e verticais.

Os recursos de encaminhamento gerais estão localizados em pistas horizontais e verticais associadas a cada linha e coluna de CLBs. Estes recursos incluem:

- 24 linhas unitárias que interligam GRMs adjacentes em cada quatro direcções;
- 72 linhas *Hex buffered* que ligam uma GRM às GRMs localizadas à distância de seis blocos em cada uma das quatro direcções;
- 12 linhas longas bidireccionais que distribuem sinais por toda a FPGA, atravessando toda a altura e largura do dispositivo.

Os recursos de entrada/saída (referenciados por *VersaRing*) formam a interface entre o *array* de CLBs e os blocos de I/O. Os recursos dedicados servem para encaminhar algumas classes de sinais

específicos (propagação de sinais de *carry* verticalmente entre os CLBs adjacentes, implementação de barramentos do tipo *tri-state*, etc.) a fim de maximizar o desempenho.

Os recursos de encaminhamento de relógio distribuem os sinais de relógio bem como outros sinais com *fanout* elevado por toda a FPGA. Estes recursos são classificados em globais e locais. Os recursos de relógio globais consistem em quatro redes dedicadas que podem controlar as entradas de relógio de todos os CLBs, blocos de entrada/saída e blocos de memória. Os recursos de relógio locais consistem de 24 linhas, 12 das quais atravessam o topo da FPGA e as outras 12 atravessam a parte inferior. Destas linhas, pode-se distribuir até 12 sinais por coluna através de linhas longas. Diferentemente dos recursos globais, os locais não estão limitados ao encaminhamento só de sinais de relógio.

Uma característica importante da arquitectura Virtex é o suporte para reconfiguração parcial em *run-time*. Para o efeito foi desenvolvida a ferramenta de software *JBits* que permite efectuar pequenas modificações directamente no circuito sem interromper o seu funcionamento [McMillan00].

2.6.3 Ferramentas CAD para as FPGAs da Xilinx

O desenvolvimento de circuitos para as FPGAs das famílias XC4000XL e Virtex-EM faz-se com a ajuda de sistemas *Foundation Series*, *Alliance Series* e ISE - *Integrated Software Environment* (esta última ferramenta não suporta a família XC4000XL). No âmbito deste trabalho foram utilizados os sistemas *Foundation Series* 3.1i e 4.1i.

A especificação do projecto pode ser feita com a ajuda dum editor de esquemáticos, dum editor de máquinas de estados ou numa HDL (VHDL, Verilog ou ABEL - *Advanced Boolean Expression Language*). O editor esquemático da Xilinx permite criar blocos funcionais utilizando a biblioteca de elementos da Xilinx e as ferramentas *CORE Generator* e *LogiBLOX*. A biblioteca da Xilinx inclui primitivas (i.e. elementos básicos tais como as portas lógicas) e macros (que são elementos compostos de múltiplas primitivas e outras macros). Para criar os módulos parametrizáveis de alto nível são utilizadas as ferramentas gráficas *CORE Generator* e *LogiBLOX* (a última não suporta a família Virtex-EM). Para a síntese utilizam-se normalmente as ferramentas *third-party* tais como *FPGA Express* da Synopsys [Synopsys].

A implementação inclui o mapeamento na FPGA específica, a colocação, o encaminhamento e a geração do *bitstream*. Na primeira fase efectua-se o mapeamento da lógica nas componentes da FPGA, tais como células lógicas, blocos de memória, etc. A seguir, realiza-se a colocação e o encaminhamento. É possível restringir o projecto com os parâmetros específicos de mapeamento, colocação e temporais. Estas restrições podem ser especificadas à mão ou com a ajuda de ferramentas assim como *Constraints Editor*, *Floorplanner* e *FPGA Editor*. Finalmente, gera-se o *bitstream* que contém toda a informação de configuração, definindo a lógica interna e as interligações. Para a verificação do projecto utilizam-se as ferramentas *Logic Simulator* e *Timing Analyzer* da Xilinx ou as ferramentas *third-party*.

O mais recente sistema de desenvolvimento ISE 5.x da Xilinx é bastante mais poderoso do que o *Foundation Series* pois possui a sua própria ferramenta de síntese e incorpora metodologias que possibilitam aumentar o desempenho dos circuitos, reutilizar módulos com a informação de

colocação relativa incorporada (tecnologia *Macro Builder*), reduzir o tempo de recompilação do projecto, etc. [Hansen02].

2.6.4 Famílias recentes de FPGAs da Xilinx

Actualmente a Xilinx produz FPGAs das séries Spartan, Virtex e Virtex-II, nomeadamente as famílias Spartan, Spartan-XL, Spartan-II, Spartan-IIE, Spartan-3, Virtex, Virtex-E, Virtex-EM, Virtex-II e Virtex-II Pro, sendo todas baseadas em SRAM [Xilinx].

A série Spartan tem como objectivo principal fornecer desempenho elevado, memória embutida e suporte robusto para IP a preço reduzido. As séries Virtex e Virtex-II abrangem dispositivos mais complexos que possibilitam a implementação de sistemas completos num único encapsulamento (ver fig. 2.10). A densidade elevada combinada com os blocos de memória embutidos, as tecnologias avançadas de gestão de relógio e suporte para uma ampla gama de normas de I/O, permitem aos utilizadores atingir níveis de desempenho que anteriormente só eram viáveis com os circuitos personalizados a uma aplicação específica (ASICs - *Application Specific Integrated Circuits*).

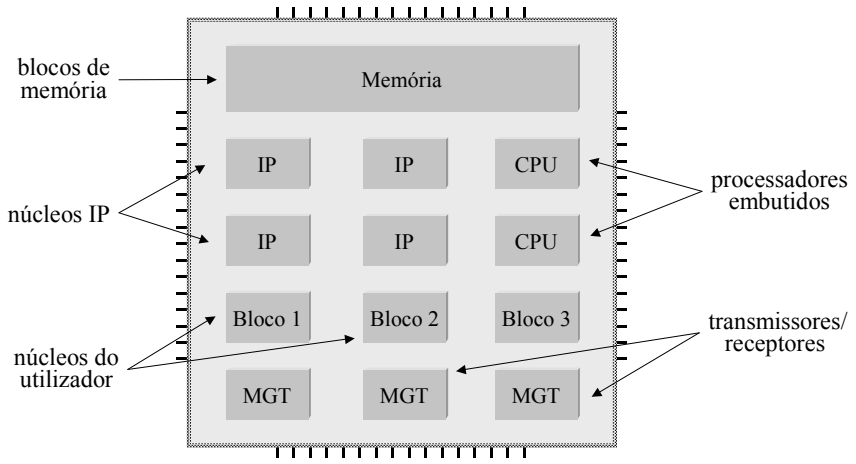


Fig. 2.10. Implementação de um sistema num único encapsulamento com a ajuda de FPGAs da família Virtex-II Pro.

Dado que a família Virtex-II Pro inclui os dispositivos mais poderosos descrevemo-la em mais detalhe. Os componentes básicos da arquitectura Virtex-II Pro estão ilustrados na fig. 2.11. As FPGAs desta família possuem as seguintes características principais:

- Até 14416 CLBs organizados num *array* de 136×106, cada um dos quais é composto por 4 *slices* e dois *buffers* do tipo *tri-state*. A estrutura de *slices* é semelhante à considerada na secção 2.6.2.1. A densidade das FPGAs da família Virtex-II Pro atinge 10M de portas de sistema. É de notar que o número de portas de sistema não reflecte as capacidades das estruturas heterogéneas existentes tais como processadores e transmissores/receptores embutidos
- Até 10008 Kbits de memória embutida e até 1738 Kbits de memória distribuída. A memória embutida consiste em 556 (no máximo) blocos *SelectRAM* de 18 Kbits cada. Os blocos são programáveis em várias configurações entre 16K×1 bit e 512×36 bits.

- Suporte para mais que 25 normas de I/O (tecnologia *Select I/O-Ultra*); até 1200 de entradas/saídas.
- Entre 4 e 12 blocos de gestão de relógio - DCMs (*Digital Clock Managers*). Os DCMs implementam toda a funcionalidade de DLLs descritos na secção 2.6.2.1 e para além disso asseguram a síntese flexível da frequência, a diferença mais precisa de fases, etc.
- Até 556 multiplicadores dedicados (18 bits × 18 bits). Os multiplicadores estão associados com cada bloco de memória embutido e são optimizados para operações baseadas no seu conteúdo.
- Até 24 transmissores/receptores MGT (*Multi-Gigabit Transceivers*) que suportam velocidades de transferência de dados entre 622 Mbits/s e 3.125 Gbits/s.
- Até 4 núcleos de processadores IBM PowerPC 405 embutidos que podem funcionar a 300 MHz. Os processadores possibilitam a implementação de sistemas embutidos complexos e permitem partilhar o projecto de uma maneira óptima atribuindo à lógica da FPGA porções computacionalmente intensivas da aplicação ficando os processadores com as porções orientadas ao controlo. Deste modo, software e hardware podem ser desenvolvidos em paralelo, o que não tem acontecido tradicionalmente, quando era impossível começar o desenvolvimento de software antes de ter um protótipo de hardware.
- Suporte para reconfiguração parcial. Todos os dados de configuração bem como os conteúdos de todos os *flip-flops*/*latches*, LUTs e blocos de memória embutidos podem ser lidos da FPGA possibilitando deste modo a depuração em *run-time*.
- Suporte completo pelo sistema ISE 5.x da Xilinx incluindo a ferramenta *CORE Generator* com uma série de núcleos de IP.

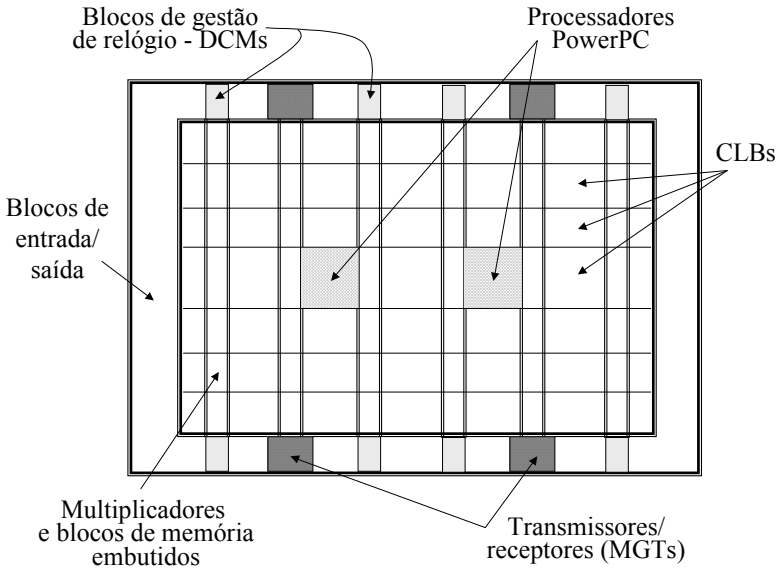


Fig. 2.11. Componentes básicos da arquitectura Virtex-II Pro.

2.7 Conclusões

Os dispositivos lógicos programáveis podem ser reconfigurados pelo utilizador a fim de implementarem circuitos diferentes. Dependendo da tecnologia aplicada, os PLD são de programação múltipla ou única. De acordo com a capacidade lógica todos os PLDs são classificados em PLDs elementares e PLDs de capacidade elevada que incluem CPLDs e FPGAs. Para implementar circuitos com base em CPLDs e FPGAs é necessário recorrer ao uso de ferramentas de desenvolvimento assistido por computador. Os actuais PLDs de capacidade elevada são considerados como uma boa alternativa aos ASICs pois são capazes de oferecer uma grande flexibilidade de utilização mantendo os custos e o tempo de desenvolvimento reduzidos.

Existe uma ampla gama de FPGAs produzidas hoje em dia. Distinguem-se pela granulosidade de blocos lógicos, arquitecturas de encaminhamento e estruturas heterogéneas embutidas. O mercado de lógica programável continua a desenvolver-se muito rapidamente. As FPGAs contemporâneas atingem a densidade de 10 milhões de portas de sistema e incluem núcleos de processadores e transmissores/receptores de múltiplos Gbits embutidos.

O estudo, análise e classificação de dispositivos lógicos programáveis feitos neste capítulo fornece a base necessária para o capítulo seguinte pois as FPGAs são o componente principal de sistemas reconfiguráveis considerados no capítulo 3.

3 Computação reconfigurável

Sumário

Os sistemas reconfiguráveis possibilitam aumentar significativamente o desempenho de computações que não são bem suportadas por processadores de uso geral correntes (com arquitecturas convencionais). Tais sistemas baseiam-se em unidades de processamento personalizadas às necessidades de uma dada aplicação recorrendo para tal aos dispositivos lógicos programáveis de alta capacidade, normalmente às FPGAs. Contudo, as FPGAs tradicionais de pouco servem para a implementação completa dum sistema. A maioria das aplicações inclui grandes porções de código que são executadas raramente. O mapeamento de todas as porções deste tipo na lógica reconfigurável seria pouco eficiente. Para além disso, algumas computações, tais como operações sobre valores reais, são melhor suportadas pelas unidades funcionais do processador que são optimizadas para realizar estas operações. Sendo assim, um sistema reconfigurável é normalmente composto por dois componentes interligados que são a lógica reconfigurável (FPGA) e um processador convencional, possibilitando deste modo aproveitar as vantagens principais de ambas as partes.

Neste capítulo introduzimos a noção de computação reconfigurável e descrevemos os seus conceitos básicos. Para tal apresentamos uma revisão de vários tipos de organização de sistemas reconfiguráveis, nomeadamente consideramos modos de reconfiguração, mecanismos de interacção entre os componentes principais, modelos de programação de todo o sistema, etc. A seguir, analisamos técnicas utilizadas na computação reconfigurável para atingir desempenho elevado. Apresentamos também alguns exemplos de sistemas reconfiguráveis mais conhecidos e as áreas típicas de aplicação da computação reconfigurável. Finalmente, serão descritas duas placas de desenvolvimento baseadas em FPGAs que foram utilizadas no âmbito deste trabalho.

3.1 Introdução

Os computadores convencionais podem ser utilizados para uma ampla gama de aplicações. Um computador de uso geral possui uma arquitectura e um conjunto de instruções fixos e aplicações diferentes são programadas dentro de restrições predefinidas. A programação destes computadores é bastante simples dado que existem muitas ferramentas disponíveis para o efeito. Tudo isto resulta num bom desempenho atingido para muitas aplicações a preço razoável. A vantagem principal desta abordagem é um grande nível de flexibilidade. Qualquer alteração no algoritmo é facilmente incorporável no código (especialmente, se o código foi desenvolvido com base em tecnologia orientada por objectos). Contudo, se considerarmos uma aplicação específica, os computadores de uso geral não são capazes de assegurar o melhor desempenho.

Quando os requisitos de uma dada aplicação excedem as capacidades dos computadores de uso geral, recorre-se a abordagens diferentes destinadas a criar sistemas computacionais de desempenho elevado. Uma das técnicas utilizadas é a computação paralela. Ao decompor um problema em tarefas menores e processar estas tarefas em paralelo, pode-se atingir resultados bastante bons. Contudo, a aplicação deve possuir estrutura compatível com tal modelo de computação. Para muitas aplicações a computação paralela oferece pouca aceleração por cada unidade de processamento adicional, sendo a razão disso parcialmente explicada pela lei de Amdhal[5] [Hartenstein01a]. Sistemas deste tipo sofrem também de grande sobrecarga (*overhead*) de comunicação entre vários processadores.

Uma abordagem diferente consiste em construir circuitos orientados à aplicação que são capazes de fornecer um bom desempenho para essa aplicação específica. Por exemplo, é possível projectar e fabricar um circuito integrado específico (i.e. um ASIC) com o controlo fixo e as unidades funcionais personalizadas e optimizadas para uma dada aplicação. Com esta técnica pode-se atingir resultados muito bons com menos recursos de hardware. Contudo os custos de projecto e de implementação de tal sistema são demasiado elevados e só podem ser justificados no caso de produções de alto volume. Quando um circuito integrado é produzido em grandes quantidades, o custo inicial é amortizado dado que cada pastilha de silício fica responsável só por uma pequena parte do custo inicial. Por esta razão os computadores de uso geral são vendidos a um preço relativamente baixo. Uma outra dificuldade da abordagem orientada à aplicação é o tempo de desenvolvimento longo enquanto o desempenho de computadores de uso geral aumenta bastante rapidamente. Por este motivo o hardware de uso especial pode ficar obsoleto passado pouco tempo. É de salientar também que as soluções baseadas em ASIC são completamente inflexíveis dado que a sua funcionalidade não pode ser modificada após o fabrico.

Levando em consideração o custo e a curta duração da vida útil dos computadores dedicados, estes não representam uma abordagem suficientemente atractiva a não ser que a necessidade deste hardware seja muito forte e justificada. Contudo, se for possível diminuir os custos e o tempo de desenvolvimento, a técnica orientada à aplicação torna-se bastante viável.

A computação reconfigurável é uma abordagem que combina as duas técnicas descritas acima possibilitando deste modo eliminar as desvantagens associadas com software "puro"

[5] O aumento de desempenho que é possível atingir com a ajuda de qualquer modo de execução mais rápido é limitado pela fracção de tempo durante a qual este modo pode ser utilizado.

(implementado num computador de uso geral) e hardware "puro" (ASIC) (ver fig. 3.1). A computação reconfigurável baseia-se em dispositivos lógicos reprogramáveis que podem atingir um desempenho elevado e, ao mesmo tempo, fornecer a flexibilidade da programação a nível de portas lógicas. Um dispositivo de hardware típico utilizado em computação reconfigurável são as FPGAs consideradas no capítulo 2. A velocidade e os recursos disponíveis em FPGAs recentes são comparáveis com os dos ASICs, enquanto gozam de muita da flexibilidade inerente às implementações em software.

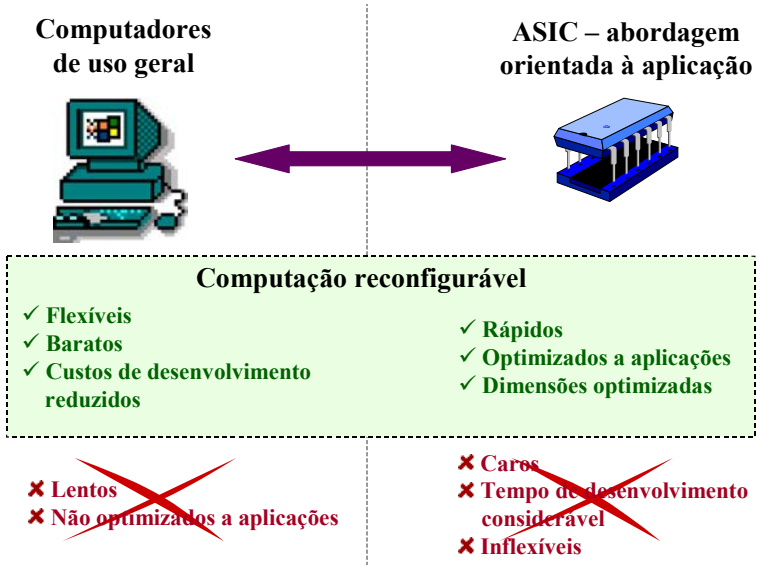


Fig. 3.1. Computação reconfigurável combina vantagens principais de computadores de uso geral e de circuitos integrados orientados à aplicação e permite eliminar as suas desvantagens principais.

3.2 Sistemas reconfiguráveis

Embora o conceito de computação reconfigurável exista há já bastante tempo [Estrin60, Estrin02], apenas recentemente surgiram tecnologias que possibilitaram a sua implementação e aplicação na prática. O interesse começou no início dos anos 90 quando a densidade das FPGAs ultrapassou as 10K portas lógicas [Guccione01]. Desde aí, a computação reconfigurável, devido à possibilidade de acelerar várias aplicações, tornou-se objecto de investigação intensiva. A sua característica mais importante é a capacidade de realizar computações em hardware com o objectivo de incrementar o desempenho, ficando ao mesmo tempo com a flexibilidade do software [Compton02].

A fim de atingir desempenho elevado e suportar uma ampla gama de aplicações, os sistemas reconfiguráveis são normalmente compostos pela lógica reconfigurável e um processador de uso geral. Para executar a aplicação de uma maneira mais eficiente, aquelas partes que não são facilmente mapeadas na lógica reconfigurável, são executadas num processador hospedeiro, enquanto as partes que requerem computações muito intensivas e podem beneficiar da sua implementação em hardware, são realizadas em FPGAs. Assim, um sistema reconfigurável típico consiste em lógica reconfigurável (denominada por *componente variável*) e num processador hospedeiro (referenciado como *componente fixo*) conforme representado na fig. 3.2.

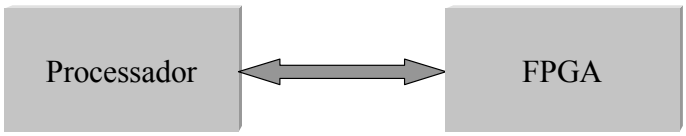


Fig. 3.2. Estrutura de um sistema reconfigurável típico.

Descrevemos seguidamente as propriedades mais importantes que caracterizam os sistemas reconfiguráveis.

3.2.1 Modos de reconfiguração

O tempo de uma operação realizada num sistema reconfigurável é a soma do tempo de configuração e do tempo de execução. A configuração é feita normalmente pelo processador hospedeiro que envia os dados de configuração que definem o funcionamento actual do hardware. A configuração pode ser carregada só no início de execução ou periodicamente a fim de variar o funcionamento do sistema. O modelo de execução também varia de sistema para sistema [Compton02]: em alguns sistemas o processador hospedeiro fica suspenso enquanto o hardware reconfigurável está activo; em outros é permitida a execução simultânea.

No domínio da computação reconfigurável podemos distinguir entre dois modos de configuração: *estático* e *dinâmico* [Sanchez99]. A reconfiguração *estática* pressupõe o funcionamento permanente da FPGA depois desta ser configurada (ver fig. 3.3). De facto, o modo estático não concede grande flexibilidade mas permite atingir um bom desempenho via utilização do hardware optimizado para uma dada aplicação [Skliarova01b].

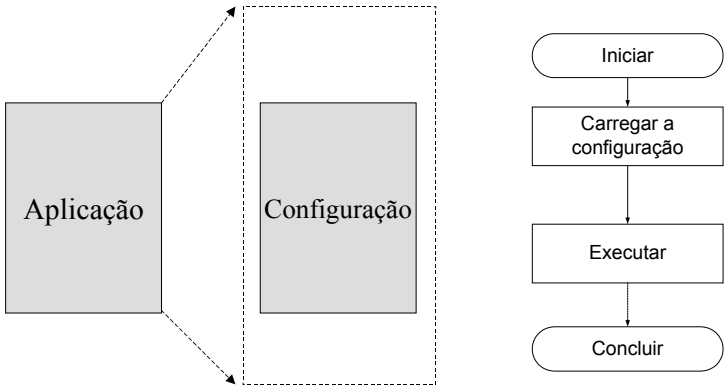


Fig. 3.3. Reconfiguração estática.

Às vezes é preciso activar configurações diferentes de acordo com a necessidade corrente de uma aplicação. Isto possibilita mais secções da aplicação a serem mapeadas em hardware do que o que pode "caber" numa FPGA. Deste modo, a maior parte da aplicação pode potencialmente ser acelerada num sistema reconfigurável resultando num desempenho mais elevado. Este conceito é frequentemente referido por reconfiguração dinâmica a que está associada a noção de *hardware virtual*. Neste caso, os recursos disponíveis na FPGA são bastante menores que o conjunto de

recursos necessários para todas as configurações. Mas em vez de reduzir o número de configurações mapeadas, estas são transferidas para a FPGA de acordo com a necessidade actual [Compton02]. Uma das vantagens dos sistemas reconfiguráveis dinamicamente comparando-os com os sistemas programáveis só no início da aplicação, é a potencialidade de efectuar optimizações do hardware com base na informação determinada durante a execução.

A reconfiguração dinâmica pode, por sua vez, ser dividida em reconfiguração global e reconfiguração parcial. A reconfiguração *global* reserva todos os recursos de hardware para cada fase de execução. Depois de uma fase ser concluída, todos os recursos da FPGA são reconfigurados para a fase seguinte (ver fig. 3.4a). A reconfiguração *parcial* recorre à modificação selectiva dos recursos de hardware ao longo de execução de uma aplicação (ver fig. 3.4b). Esta flexibilidade permite que o hardware seja mais personalizado às necessidades correntes da aplicação. A reconfiguração parcial exige que só os recursos seleccionados sejam reprogramados o que resulta num *overhead* de configuração menor do que no caso anterior. No âmbito deste trabalho utilizamos a reconfiguração dinâmica parcial.

É de notar que existem outros termos que são utilizados para designar reconfiguração estática e dinâmica, por exemplo "ligação durante o projecto" (*design-time binding*) e "ligação durante a execução" (*implementation-time binding*) [Schaumont01]; "configuração em *compile-time*" e "configuração em *run-time*" [Hutching95].

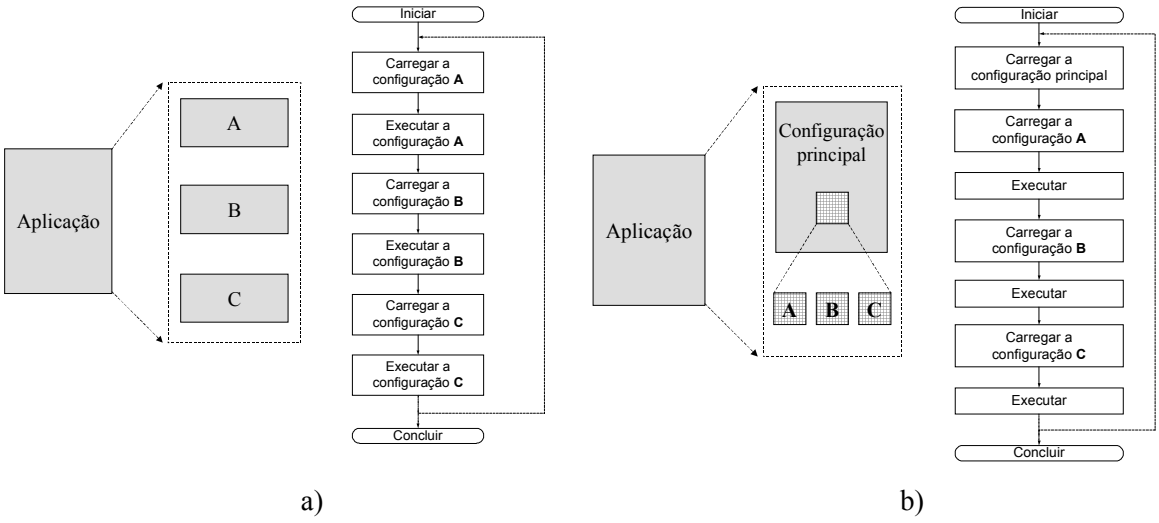


Fig. 3.4. Reconfiguração dinâmica global (a) e parcial (b).

A fim de possibilitar a reconfiguração dinâmica existem três tipos de dispositivos disponíveis [Compton02]:

1) *Dispositivos de contexto único* requerem uma reconfiguração completa para alterar qualquer um dos bits de programação (ver fig. 3.5a). A maioria das FPGAs disponíveis no mercado pertencem a esta classe. É de notar que existe uma técnica especial que permite a reprogramação parcial deste grupo de FPGAs. Esta baseia-se em modelos (*hardware templates*) e será considerada em detalhe no capítulo 5.

2) *Dispositivos de contexto múltiplo* possuem várias camadas de bits de programação estando em cada instante activa apenas uma camada (ver fig. 3.5b). A vantagem principal destes dispositivos é a possibilidade de mudar de contexto de uma maneira extremamente rápida. Para além disso, é permitida a configuração nos bastidores (*background*) possibilitando reprogramar um contexto enquanto um outro contexto está activo.

3) *Dispositivos parcialmente reconfiguráveis* permitem que áreas pequenas da FPGA sejam modificadas sem necessidade de reprogramar todo o dispositivo (ver fig. 3.5c). As reconfigurações parciais requerem bastante menos tempo do que a reconfiguração total. Frequentemente, as zonas da FPGA que não são perturbadas podem continuar a execução permitindo que a reconfiguração seja realizada em simultâneo com a execução [Compton02].

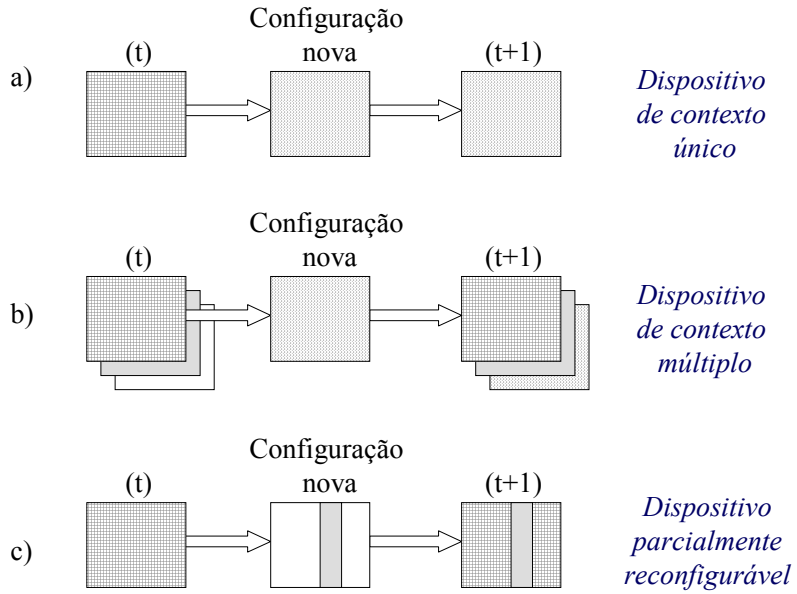


Fig. 3.5. Vários tipos de dispositivos reconfiguráveis.

Uma característica importante a ter em conta em sistemas reconfiguráveis dinamicamente, é que o tempo gasto em reconfiguração não deve absorver a aceleração ganha. Por isso são frequentemente aplicadas várias técnicas destinadas a reduzir o *overhead* da reconfiguração [Compton02, Kennedy03]. Estas incluem utilização de métodos de compressão de dados de configuração; carregamento dos ficheiros de configuração em *background*, enquanto o processador principal está a executar uma outra parte da aplicação; etc.

3.2.2 Mecanismo de interacção entre o hardware reconfigurável e o processador hospedeiro

O mecanismo de interacção define o nível de "proximidade" dos componentes fixo e variável dum sistema reconfigurável [Compton02, Jacob98, Radunovic98]. No caso mais simples, o hardware reconfigurável pode representar uma unidade de processamento externa que comunica com o processador hospedeiro raramente ou até funciona de uma maneira independente (ver fig. 3.6a). Se a unidade reconfigurável for utilizada como um acelerador, então o processador hospedeiro

inicializa o hardware e envia-lhe todos os dados necessários [Vuillemin96, Buell96, Athanas93]. O acelerador efectua todas as computações independentemente do processador principal e no final retorna os resultados, podendo ambos os componentes executar em simultâneo.

Existem arquitecturas em que ambos os componentes principais (i.e. a lógica reconfigurável e o processador) residem na mesma pastilha de silício e são interligados muito fortemente (ver fig. 3.6b) [Razdan94, Gustin99, Hauck97]. O hardware reconfigurável é usado apenas para implementar unidades funcionais reconfiguráveis dentro do processador hospedeiro. Neste caso as unidades reconfiguráveis actuam como unidades funcionais no *datapath* do processador sendo os operandos guardados em registos. Este modelo permite estender a programação tradicional com a capacidade de definir e adicionar instruções personalizadas. Arquitecturas deste tipo beneficiam do nível próximo de interacção resultando num baixo *overhead* de comunicação. Contudo, tais sistemas são de projecto e implementação difíceis e sofrem frequentemente de limitações no acesso à memória.

A fim de explorar todo o potencial da lógica reconfigurável é necessário assegurar a largura de banda adequada na comunicação com a memória (ver fig. 3.6c). Sistemas nesta categoria permitem que a lógica reconfigurável tenha acesso directo à memória em vez de transferir todos os dados necessários via registos do processador hospedeiro [Jacob98, Callahan00, Waingold97, Singh00]. A fim de possibilitar isso normalmente cria-se uma memória local (semelhante a *cache*) que tem ligação directa eficiente com a memória principal. A integração directa da memória e da lógica reconfigurável aumenta a quantidade de dados disponíveis para os recursos reconfiguráveis possibilitando deste modo explorar melhor o paralelismo e aumentar o desempenho. Como foi descrito no capítulo 2, todas as FPGAs recentes incorporam grandes blocos de memória embutidos que podem ser utilizados para estes fins. As arquitecturas propostas neste trabalho aproveitam esta possibilidade.

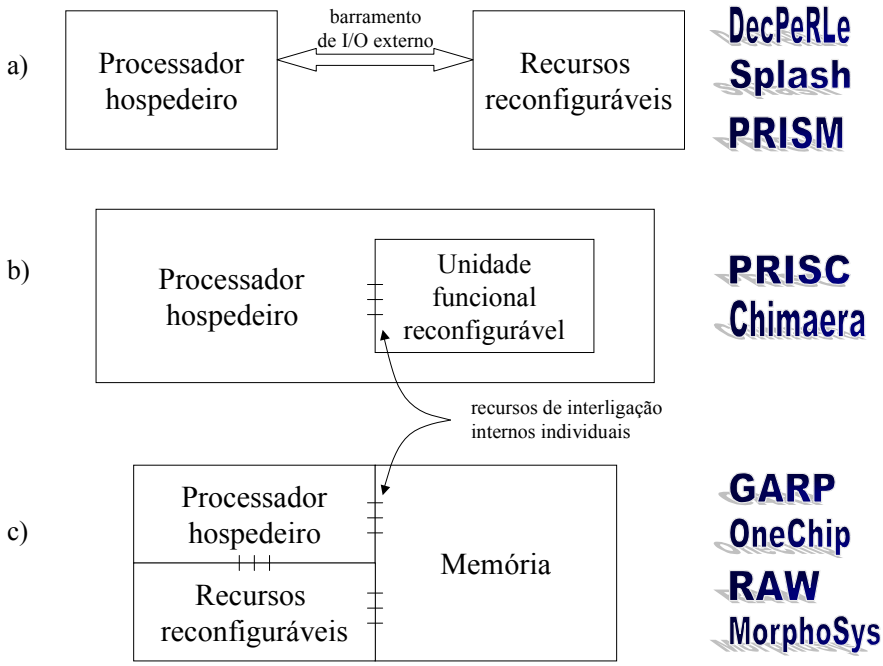


Fig. 3.6. Diferentes tipos de mecanismos de interacção entre o hardware reconfigurável e o processador hospedeiro.

O nível de "proximidade" dos componentes principais de um sistema reconfigurável influencia a largura de banda suportada na comunicação e afecta o modelo de programação do sistema. Assim, sistemas reconfiguráveis fortemente interligados servem bem para a aceleração em geral (execução de ciclos internos, etc.). Sistemas com fraca interligação são adequados para processamento idêntico de grandes volumes de dados, sendo muito importantes para algumas classes de aplicações [Radunovic98].

3.2.3 Capacidade lógica

O número de FPGAs envolvidas num sistema reconfigurável pode variar de 1 a algumas centenas. A capacidade lógica determina a complexidade das aplicações que podem ser mapeadas em hardware. Por exemplo, o detector *DZero* (utilizado na área de física subatómica) inclui um *array* de 582 FPGAs das famílias Spartan, Virtex e Virtex-E da Xilinx, capaz de processar 1.5 *terabytes* de dados por segundo [Havener02].

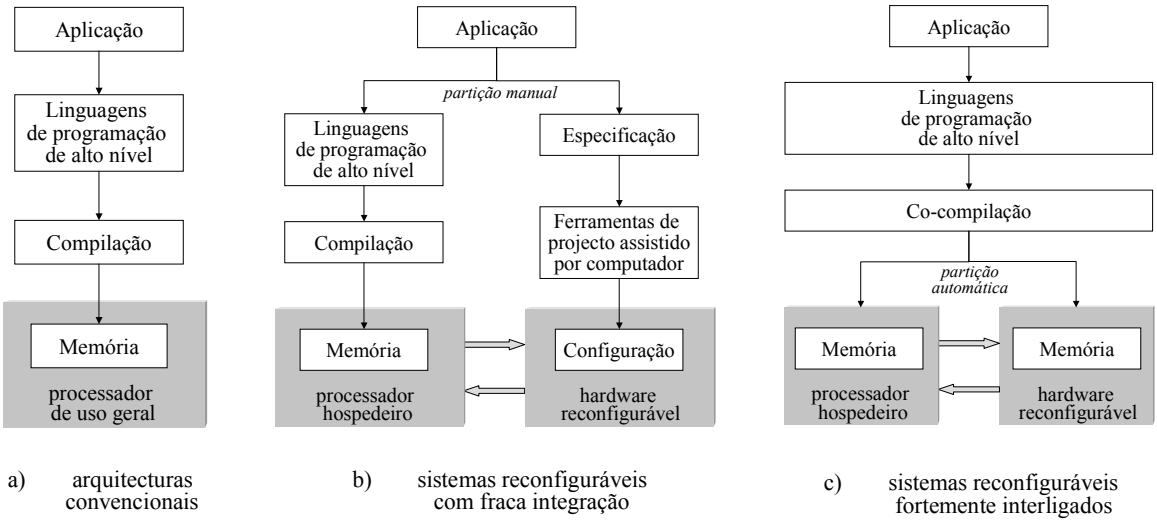
A utilização de sistemas compostos por múltiplas FPGAs requer a definição de uma boa topologia de encaminhamento e a existência de ferramentas de projecto assistido por computador capazes de dividir circuitos em porções interligadas de uma maneira eficiente, que permitam atingir um nível adequado de utilização dos recursos lógicos.

3.2.4 Modelo de programação

Existem várias maneiras de mapear uma aplicação num sistema reconfigurável. No caso ideal, um compilador devia compilar automaticamente um programa escrito numa linguagem de programação de alto nível (assim como C++) numa configuração a ser carregada na FPGA. É de notar contudo que linguagens de programação convencionais não podem ser utilizadas directamente para estes fins por não suportarem noções como processamento paralelo e ser bastante difícil identificar o paralelismo automaticamente. Contudo, há sistemas reconfiguráveis que são programados deste modo [Callahan00]. A vantagem principal desta técnica é que o utilizador não precisa de ter conhecimentos sobre a arquitectura do sistema bem como sobre o hardware em geral, e a utilização da lógica reconfigurável fica transparente.

Uma abordagem diferente consiste no desenvolvimento de hardware reconfigurável de acordo com as necessidades de uma dada aplicação. Para tal utiliza-se o fluxo de projecto típico descrito no capítulo 2. Esta técnica requer que o utilizador tenha experiência em projecto de circuitos de hardware. Embora o tempo total de implementação de uma aplicação seja neste caso maior, os resultados podem ser melhores dado que o projectista tem mais flexibilidade e mais opções a explorar.

É de salientar que cada uma das técnicas descritas acima possui as suas vantagens e desvantagens. Normalmente a compilação automática utiliza-se em arquitecturas orientadas para aceleração de aplicações gerais, enquanto para aplicações específicas podem ser atingidos melhores resultados com a segunda abordagem. A fig. 3.7 ilustra as três maneiras possíveis de mapear uma aplicação em arquitecturas convencionais e sistemas reconfiguráveis.



a) arquitecturas convencionais

b) sistemas reconfiguráveis com fraca integração

c) sistemas reconfiguráveis fortemente interligados

Fig. 3.7. Modelos de programação e de execução de uma aplicação em várias plataformas computacionais.

3.2.5 Modelo de execução

Uma aplicação pode ser inteiramente implementada em hardware reconfigurável (cabendo ao processador hospedeiro apenas tarefas de inicialização da FPGA) ou pode ser partilhada entre o hardware e o software. Existem vários métodos de partilhar a aplicação. Uns implementam em FPGA instruções específicas estendendo deste modo o conjunto básico de instruções do processador hospedeiro [Razdan94, Hauck97]. Outros métodos efectuam a partição de acordo com a complexidade da função a executar. Por exemplo, partes computacionalmente intensivas do programa são atribuídas à FPGA enquanto as partes que exibem pouco paralelismo são processadas pelo processador de uso geral [Sousa01]. Sistemas deste tipo baseiam-se na regra 90/10, que diz que 90% do tempo de execução dum programa é despendido por 10% do seu código. A fim de atingir resultados significativos tenta-se acelerar esta pequena porção do código com a ajuda de FPGA. Os 90% restantes do código possuem pouco paralelismo e são melhor executados num processador de uso geral. Sendo assim, o processador é responsável pelas tarefas sequenciais enquanto à FPGA são atribuídas tarefas computacionalmente intensivas [DeHon00]. Num terceiro tipo de métodos a aplicação é partilhada de acordo com a capacidade lógica do componente variável do sistema [Skliarova01b]. Neste caso, se uma aplicação (ou alguma parte desta) não "cabe" na FPGA disponível, deve ser primeiro processada pelo processador hospedeiro e depois transferida para a FPGA. Neste trabalho recorremos ao uso dos dois últimos métodos. A partição pode ser executada de uma maneira manual (ver fig. 3.7b) ou automática (ver fig. 3.7c).

3.3 Técnicas básicas utilizadas na computação reconfigurável a fim de atingir um desempenho elevado

Note-se que a frequência de relógio suportada por FPGAs recentes é aproximadamente uma ordem de grandeza menor do que a frequência utilizada em computadores de uso geral, que actualmente se situa em alguns GHz. Isto explica-se parcialmente pela necessidade de acomodar interligações

programáveis. Portanto, o mapeamento directo de uma aplicação do software para uma FPGA não irá assegurar um desempenho mais elevado que o atingido num computador de uso geral. A fim de conseguir um bom desempenho, são aplicadas as três técnicas seguintes [Abramson98a]:

1) *Largura de banda elevada no acesso à memória*. Tradicionalmente, os computadores de uso geral organizam a memória em forma de um conjunto de palavras de tamanho fixo. Os dados de um problema podem não caber exactamente numa só palavra, consequentemente são precisos vários acessos à memória para os processar. Contudo, em FPGA é possível organizar a memória de acordo com a dimensão actual dos dados, permitindo que estes sejam processados numa única operação.

2) *Uso de unidades funcionais específicas e optimizadas*. O conjunto de instruções básicas utilizado em computadores de uso geral foi desenvolvido para servir uma ampla gama de aplicações. Por isso, algumas operações específicas podem precisar de muitas instruções para serem expressas, e levar muitos ciclos de relógio para serem concluídas. Frequentemente, as aplicações envolvem operações bastante simples mas estas não são bem suportadas por ALUs (*Arithmetic and Logic Units*) convencionais. Com FPGAs torna-se possível criar unidades funcionais optimizadas para certas operações e tamanhos de dados (necessários para uma aplicação particular). Como resultado, a unidade é capaz de executar num ciclo de relógio algumas operações que requerem vários ciclos num computador de uso geral. Bons exemplos de operações deste tipo são as que funcionam a nível de bits individuais e envolvem expressões lógicas relativamente complexas.

3) *Exploração de paralelismo e de pipelining*. As unidades funcionais descritas acima são normalmente simples e ocupam poucos recursos de hardware, sendo possível reproduzi-las para explorar o paralelismo. Para além disso, as FPGAs típicas incorporam um grande número de *flip-flops* e LUTs distribuídos por todo o dispositivo que podem ser utilizados para permitir uma gestão flexível de registos construídos com base nestes elementos de memória. Isto possibilita a implementação de técnicas de escalonamento (*pipelining*) eficientes que resultam em melhor desempenho, aumentam o nível de utilização dos recursos de hardware e reduzem acessos à memória externa.

3.4 Exemplos de sistemas reconfiguráveis

Durante os últimos 10-15 anos foram propostas muitas arquitecturas de sistemas reconfiguráveis. Estas diferem bastante em termos de mecanismos de interacção dos componentes principais e dos modelos de programação e de execução. Descrevemos alguns exemplos de sistemas mais conhecidos agrupando-os de acordo com o mecanismo de interacção adoptado.

3.4.1 Sistemas reconfiguráveis com fraca interligação

3.4.1.1 DECPeRLe

Um dos primeiros trabalhos que se enquadram no paradigma de computação reconfigurável é o DECPeRLe-1 também conhecido sob o nome mais geral de PAM [Vuillemin96]. DECPeRLe-1 é

um coprocessador ligado ao barramento de I/O de um processador hospedeiro. O processador hospedeiro deve carregar um ficheiro de configuração no PAM e a seguir, pode ler e escrever no PAM como num módulo de memória, enquanto o PAM processa os dados entre as instruções de leitura e escrita.

DECPeRLe-1 foi implementado numa placa que contém 16 FPGAs XC3090 da Xilinx organizadas num *array* de 4×4 (ver fig. 3.8). Cada FPGA tem ligações directas às 4 FPGAs-vizinhas. Para além disso existem barramentos comuns. DECPeRLe-1 inclui também 4 bancos de memória (4MB no total). As aplicações implementadas que utilizaram a memória incorporada funcionaram a uma frequência de ~25 MHz, enquanto para as aplicações que não necessitavam aceder à memória, foram atingidas frequências maiores. DECPeRLe-1 possui 4 conectores, três dos quais servem para estabelecer ligações com os dispositivos externos e o último destina-se a estabelecer a interface com o processador hospedeiro. Para descrever circuitos foi escolhida a linguagem C++ complementada com uma biblioteca proprietária.

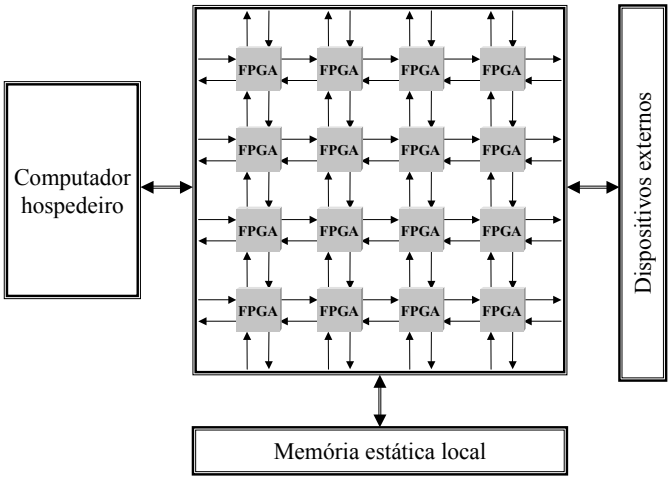


Fig. 3.8. Arquitectura DECPeRLe-1.

3.4.1.2 Splash

A série de sistemas reconfiguráveis Splash também é muito conhecida [Buell96]. Splash-2 foi implementado numa placa com 17 FPGAs XC4010 da Xilinx interligadas num *array* linear. Para além das ligações entre as FPGAs adjacentes existem interruptores para comunicações gerais limitadas entre as FPGAs que não são vizinhas. Cada FPGA é ligada a um banco de RAM com a configuração 256K×16 bits. Uma ou mais placas com as FPGAs podem ser ligadas a um computador hospedeiro via um barramento especial. Aplicações executadas em Splash-2 incluem processamento de sinal, pesquisa no texto, processamento de padrões de ADN (*Ácido Desoxirribonucleico*), etc.

3.4.1.3 PRISM

Uma das primeiras arquitecturas na qual FPGAs interferiram directamente com um processador foi o PRISM (*Processor Reconfiguration through Instruction Set Metamorphosis*) [Athanas93]. PRISM liga um microprocessador 68010 da Motorola que funciona a 10 MHz a um *array* de 4 FPGAs XC3090 da Xilinx. O processador é complementado com um conjunto de instruções de

hardware que representam um subconjunto de funções de C do programa executado no sistema. A aceleração é limitada pela interface processador-coprocessador embora a comunicação fosse restringida à passagem de argumentos e dos resultados respectivos.

3.4.2 Sistemas em que a lógica reconfigurável implementa unidades funcionais no *datapath* do processador

3.4.2.1 PRISC

A arquitectura PRISC (*Programmable Instruction Set Computers*) [Razdan94] baseia-se num processador RISC (*Reduced Instruction Set Computer*) complementando a funcionalidade do *datapath* com a lógica reconfigurável sendo esta integrada no processador através de adição de uma unidade funcional programável – PFU (*Programmable Functional Unit*) às unidades funcionais fixas (ver fig. 3.9). A PFU é implementada com base em LUTs. A complexidade das funções realizadas na PFU é de molde a que a sua latência não excede o tempo de um ciclo do processador. Várias imagens de PFU são pré-compiladas e activadas através de comutação de contextos. Deste modo o conjunto básico de instruções do processador é estendido com as instruções de hardware implementadas em PFU. A arquitectura requer que uma ferramenta de software extraia automaticamente a função de hardware e gere imagens de hardware adequadas.

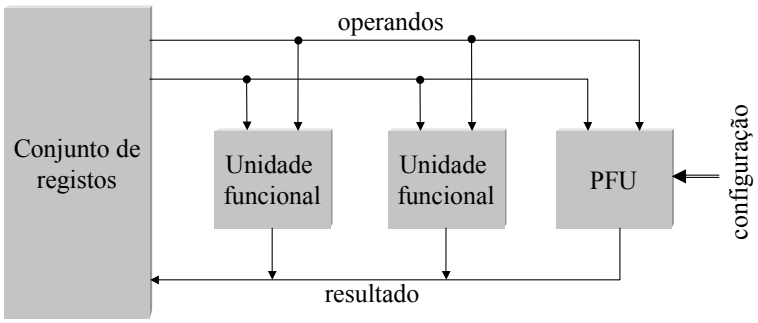


Fig. 3.9. Arquitectura PRISC.

3.4.2.2 Chimaera

Chimaera [Hauck97] é um sistema parcialmente reconfigurável em *run-time*. A interface entre o processador e a lógica reconfigurável é semelhante à implementada no PRISC. Contudo, Chimaera permite que o processador e a lógica executem em simultâneo. Componente principal do Chimaera é o *array* reconfigurável, onde todas as computações são realizadas. O *array* é composto por linhas de células lógicas entre as quais passam canais de encaminhamento horizontais. O *array* obtém as suas entradas directamente dos registos do processador hospedeiro. A cada linha do *array* reconfigurável corresponde uma posição da CAM (*Content Addressable Memory*) que indica quais as instruções que já concluíram a sua execução. A CAM analisa todas as instruções a serem executadas recolhendo apenas aquelas que devem ser processadas pelo *array* reconfigurável. Se a instrução lida já está presente no *array*, o seu resultado é enviado para um registo. Caso contrário, o processador é parado enquanto a lógica de controlo carrega a instrução adequada da memória para o *array* reconfigurável.

3.4.3 Sistemas reconfiguráveis fortemente interligados

3.4.3.1 MorphoSys

A arquitectura MorphoSys [Singh00] consiste num componente reconfigurável (*array* de células reconfiguráveis) combinado com um processador *TinyRISC* e na interface à memória de largura de banda elevada, todos integrados numa mesma pastilha de silício (ver fig. 3.10). O *array* é organizado em 8×8 células cada uma das quais inclui uma ALU de 28 bits, um multiplicador de 16×12 bits, uma unidade de deslocamento, dois multiplexadores e registos. As células são configuradas por palavras de contexto guardadas numa memória de contexto. A memória de contexto é capaz de armazenar múltiplos conjuntos de dados de configuração (até 32) diminuindo deste modo o tempo despendido em reconfiguração. Uma palavra define a configuração para toda a linha ou toda a coluna do *array* seguindo deste modo o modelo de computação SIMD (*Single Instruction stream, Multiple Data streams*). O funcionamento do *array* reconfigurável é controlado pelo processador através de instruções especiais adicionadas ao conjunto de instruções básico. Para assegurar a interface adequada com a memória MorphoSys inclui um *buffer* de armazenamento interno e um controlador de DMA (*Direct Memory Access*). O sistema é orientado às aplicações computacionalmente intensivas que operam sobre dados de tamanho de uma palavra tais como compressão de vídeo e codificação de dados.

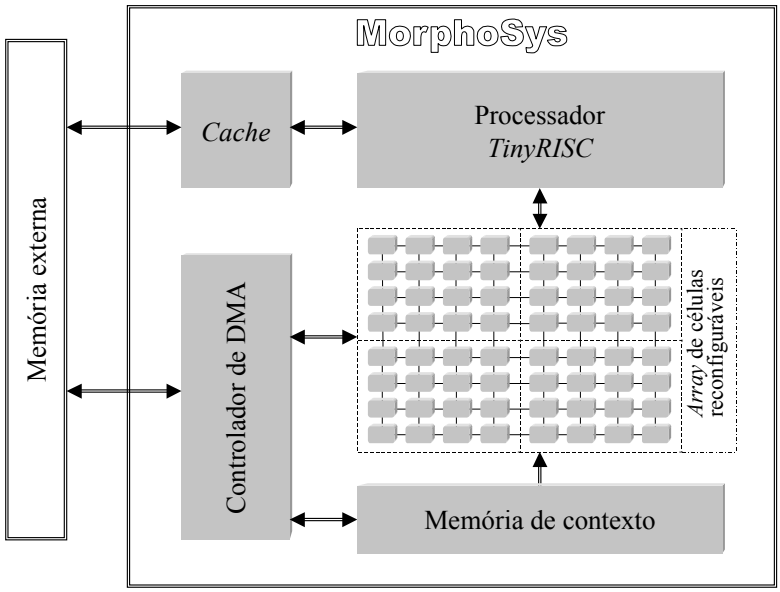


Fig. 3.10. Componentes principais da arquitectura MorphoSys.

3.4.3.2 Garp

A arquitectura Garp [Callahan00] pretende integrar numa mesma pastilha de silício um processador MIPS (*Million Instructions Per Second*) e um coprocessador programável optimizado para acelerar ciclos em aplicações de software de uso geral (i.e. recorre à regra 90/10). O problema de largura de banda no acesso à memória (dado que os ciclos operam frequentemente sobre estruturas de dados residentes na memória) é resolvido ao dar à lógica reconfigurável acesso à hierarquia de memória

do processador hospedeiro. Para reduzir o tempo de reconfiguração, o coprocessador inclui uma *cache* que mantém configurações recentemente removidas. Depois de configurado, o Garp pode executar de uma maneira independente podendo ser interrompido pelo processador a qualquer altura. O compilador desenvolvido para Garp adapta muitas técnicas propostas previamente para a exploração do paralelismo ao nível de instruções em processadores VLIW (*Very Long Instruction Word*). O fluxo de compilação consiste na identificação de porções do código C que podem ser executadas pelo coprocessador reconfigurável, síntese do código de interface entre hardware/software e das configurações da lógica reconfigurável.

3.4.3.3 RAW

O sistema RAW (*Reconfigurable Architecture Workstation*) [Waingold97] consiste em múltiplos blocos, sendo cada bloco composto por memória de instruções, memória de dados e um *datapath* com unidade reconfigurável e unidade de processamento de números reais. Todos os blocos estão interligados possibilitando comunicações entre as unidades de processamento vizinhas. A abordagem adoptada por RAW para atingir uma grande largura de banda no acesso à memória é diferente da utilizada em Garp. Cada processador tem a ligação directa dedicada com a sua própria memória local resultando numa arquitectura do tipo MIMD (*Multiple Instruction streams, Multiple Data streams*). O sistema precisa de efectuar uma distribuição das instruções entre todos os blocos e tratar da comunicação entre estes.

3.4.3.4 OneChip

O sistema OneChip [Jacob98] é semelhante a PRISC pois utiliza PFU junto com as unidades funcionais fixas. Contudo, OneChip recorre a múltiplas PFUs cada uma das quais não precisa de ser combinatória pura podendo implementar circuitos sequenciais. Portanto, o atraso lógico em OneChip não é limitado a um ciclo de relógio. Para além disso, na arquitectura OneChip o processador fixo, a lógica reconfigurável e a memória são integrados numa única pastilha de silício. O processador e a lógica reconfigurável podem executar em paralelo. Para tal foi proposto um mecanismo que impede o processador de aceder a alguns blocos da memória que são utilizados pelo componente variável, preservando deste modo a consistência da memória. A comunicação com a memória é efectuada através de um controlador que arbitra acessos entre os dois componentes principais. Foi criado um protótipo do OneChip utilizando o sistema Transmogrifier-2. Para avaliar a arquitectura OneChip foram escolhidas duas aplicações: a transformada bidimensional discreta do coseno e um filtro FIR (*Finite Impulse Response*).

Outros exemplos de sistemas reconfiguráveis [RC_Bibliography] incluem Nano Processor [Wirthlin94], RENCO (*REconfigurable Network COmputer*) [Sanchez99], SPYDER (*REconfigurable Processor Development SYstem*) [Sanchez99], HARP [Page96], MATRIX [Mirsky96], NAPA (*National Adaptive Processing Architecture*) [Rupp98, Gokhale98], KressArray [Hartenstein01b], ADRES (*Architecture for Dynamically Reconfigurable Embedded System*) [Mei03], etc.

A área de aceleração de computações combinatórias também tem sido alvo de uma investigação intensiva que resultou no desenvolvimento de muitas arquitecturas orientadas a estas aplicações. Sistemas reconfiguráveis propostas nesta área serão descritos e analisados em mais detalhe, destacando as suas vantagens e desvantagens principais, nos capítulos subsequentes.

3.5 Aplicações da computação reconfigurável

Para algumas classes de tarefas, sistemas reconfiguráveis foram capazes de atingir um desempenho muito bom em comparação com os computadores de uso geral. Outras tarefas foram mapeadas para hardware reconfigurável pois este abre oportunidades inovadoras a explorar. Todas as aplicações potenciais podem ser divididas em três categorias de acordo com o objectivo principal que se pretende atingir. A seguir, descrevemos brevemente estas categorias ilustrando cada área de aplicação com exemplos.

3.5.1. Aceleração de tarefas computacionalmente intensivas

Todas as implementações pertencentes a esta categoria têm como objectivo principal acelerar tarefas computacionalmente intensivas. Uma característica comum é que estas aplicações servem bem para implementações paralelas aproveitando as capacidades básicas da computação reconfigurável.

Processamento de sinal e imagem é um domínio de aplicações bastante popular na área de computação reconfigurável. Este domínio caracteriza-se pelo elevado grau de paralelismo inerente, grande quantidade dos dados a processar e é adequado para *pipelining* [Mangione97]. Estes factores permitem atingir um bom desempenho num sistema reconfigurável comparando com implementações num computador de uso geral. Por exemplo, PAM foi utilizado para implementar um algoritmo de visão estéreo conseguindo-se uma aceleração de 30 vezes em comparação com uma realização em hardware dedicado baseado em 4 DSPs (*Digital Signal Processors*) [Vuillemin96]. Splash-2 também foi utilizado para processamento de imagens. Os dados de vídeo entravam numa extremidade do *array* linear e eram processados ao passar por cada elemento de processamento. Descrições de outros sistemas reconfiguráveis destinados a processar imagens e sinais encontram-se na literatura [Bouridane99, Dick99, Haynes00, Callahan00, Tessier01, Singh00].

Criptografia e aritmética de inteiros longos também são áreas bastante populares na computação reconfigurável. Isto explica-se pelo facto de as operações sobre inteiros longos não serem directamente suportadas pelos computadores convencionais. Assim, PAM foi utilizado para calcular o resultado da operação $A \times B + C$, onde A pode ter até 2 Kbits, e B e C são parâmetros de tamanhos arbitrários [Vuillemin96]. O sistema produziu resultados à taxa de 66 Mb/s. A criptografia de RSA (*Rivest-Shamir-Adelman*), por sua vez, envolve operações que podem ser decompostas numa sequência de multiplicações de inteiros longos. Por exemplo, PAM para as chaves de 512 bits atingiu a velocidade de descodificação de 300 Kb/s [Vuillemin96].

Foram propostas várias arquitecturas de sistemas reconfiguráveis na área de comunicações. Estes sistemas são optimizados para tarefas tais como implementação de protocolos de acesso, armazenamento e envio de dados, controlo de tráfego e suporte para a qualidade de serviço [Iliopoulos00, Tang00, Rabaey00].

Recentemente, foram efectuadas muitas tentativas de acelerar aplicações que envolvem algoritmos de controlo bastante complexos. Uma atenção especial neste contexto foi dada aos problemas na área de optimização combinatória. Dado que esta tese também é dedicada à investigação de

arquitecturas reconfiguráveis para problemas de optimização combinatória, uma revisão mais completa do trabalho relacionado será efectuada nos capítulos seguintes.

3.5.2 Emulação de hardware

É bastante difícil saber se algum circuito projectado é correcto e estimar o seu desempenho antes da sua implementação física. Tradicionalmente utilizam-se duas abordagens para a verificação de projectos complexos: prototipagem e simulação em software. A construção de um protótipo é uma tarefa dispendiosa e morosa. As técnicas de simulação são muito flexíveis, mas muito lentas.

É de salientar que as FPGAs servem muito bem para emular dispositivos de hardware, pois são plataformas extremamente flexíveis que em geral não exigem conhecimento prévio sobre as características das aplicações que podem ser mapeadas [Dubois98, Gschwind01]. A emulação é muito mais rápida que a simulação possibilitando deste modo a verificação de um projecto com grandes quantidades de dados reais. Para além disso, a emulação reflecte as características da implementação final mais fielmente do que a simulação, e portanto resulta em avaliação do desempenho e verificação do projecto mais correctas.

FPGAs recentes tais como Virtex-II da Xilinx e Stratix da Altera incorporam grandes quantidades de lógica, unidades aritméticas, blocos de memória embutidos, o que as torna uma plataforma muito adequada para a prototipagem [Souza03]. Uma revisão de plataformas de emulação e de prototipagem baseadas em FPGAs disponíveis comercialmente e desenvolvidas no âmbito de investigação académica, é feita em [Krupnova00].

3.5.3 Hardware evolutivo

Uma área de investigação bastante recente é o *hardware evolutivo*. Este termo descreve abordagens diferentes utilizadas para desenvolver circuitos electrónicos com a ajuda de técnicas evolutivas [Miller98]. Normalmente estas abordagens dividem-se em dois grupos de acordo com a área de aplicação. O primeiro grupo abrange a evolução directa em componentes existentes (geralmente, em FPGAs), enquanto o segundo grupo inclui as tentativas de simular a funcionalidade desejada a fim de a implementar posteriormente em componentes reais.

As potencialidades das técnicas de hardware evolutivo são bastante extensas pois teoricamente possibilitam a construção autónoma (i.e. quase sem intervenção dos peritos humanos) dos circuitos com as características necessárias cuja estrutura é previamente desconhecida. Isto explica-se pelo facto de os circuitos evolutivos possuírem capacidades de auto-optimização. Por exemplo, Miller *et al*. evoluíram um multiplicador de 3 bits 20% mais eficiente (em termos do número de portas lógicas utilizadas) do que qualquer projecto convencional conhecido [Miller00].

Uma outra aplicação possível destas técnicas é a auto-reparação de circuitos autónomos cuja reparação por peritos humanos é por algum motivo impossível (por exemplo, nos sistemas utilizados no espaço) [Vigander01]. Tudo isto torna o hardware evolutivo uma área de investigação bastante promissora [Sanchez96, Sipper98, Koza99]. É de notar todavia que ainda não foi construído nenhum sistema suficientemente complexo, capaz de efectuar auto-optimização rápida e eficiente. Algumas razões que explicam este facto podem ser encontradas em [Torresen00].

3.6 Placas de desenvolvimento

No âmbito deste trabalho foram utilizadas duas placas de desenvolvimento: a placa XStend v1.2 da XESS [XESS] e a ADM-XRC da Alpha Data [Alpha]. A primeira é um componente bastante simples que comunica com o computador hospedeiro através da porta paralela. A segunda placa é mais complexa e liga-se ao processador hospedeiro via barramento PCI (*Peripheral Component Interconnect*) abrindo deste modo mais possibilidades para a reconfiguração dinâmica parcial. Descrevemos a seguir as características principais destas placas.

3.6.1 Placa XStend

Na fig. 3.11 estão representados os componentes principais da placa XStend da XESS. Esta inclui uma porta PS/2 que permite comunicar com dispositivos tais como teclado e rato, uma porta VGA (*Video Graphics Array*) que possibilita a comunicação com um monitor VGA, 2 *displays* de 7 segmentos, 10 LEDs individuais, 8 interruptores, conectores para ligar dois bancos de memória de 32KB cada, botões, área de prototipagem e interface para ligar uma placa das séries XS40 ou XS95 da XESS.

No âmbito do trabalho foi utilizada a placa XS40-010XL ligada à XStend. A arquitectura básica da XS40-010XL também está representada na fig. 3.11. O componente principal da placa é a FPGA XC4010XL da Xilinx. Para além disso, a XS40 inclui um banco de memória estática de 32 KB, um *display* de 7 segmentos, um oscilador de 12 MHz, um microcontrolador 8031, uma porta VGA e uma porta paralela. Existe também um conector que permite ligar uma EEPROM série externa para o armazenamento persistente dos dados de configuração da FPGA.

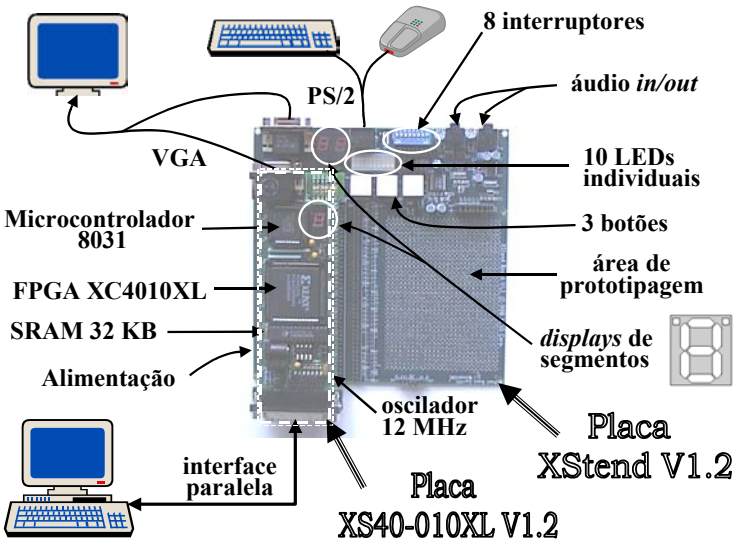


Fig. 3.11. Componentes principais da placa XStend V1.2.

O carregamento dos *bitstreams* e a comunicação com o computador efectuam-se através da porta paralela. O teste e a programação da placa realizam-se com a ajuda de ferramentas XSTOOLS da XESS [XESS]. O programa GXSLOAD configura a FPGA com o ficheiro de *bitstream*. O

programa GXSPORT serve para testar o circuito implementado enviando os sinais de teste para a placa via porta paralela (para tal há 8 bits de dados disponíveis). A comunicação da placa para o computador também se efectua através da porta paralela podendo enviar 4 bits de dados. O programa GXSTEST permite testar a placa.

3.6.2 Placa ADM-XRC

A placa ADM-XRC é mais recente e poderosa que a XStend considerada acima. ADM-XRC é um dispositivo PMC (*PCI Mezzanine Card*) baseado em FPGAs das famílias Virtex e Virtex-EM da Xilinx. No âmbito do trabalho foi utilizada a FPGA XCV812E da família Virtex-EM. O diagrama simplificado da placa ADM-XRC está representado na fig. 3.12.

A placa ADM-XRC vem por sua vez instalada numa placa ADC-PMC que pode levar até dois dispositivos ADM-XRC. ADC-PMC é um adaptador para PCI que se baseia numa *bridge* PCI-PCI da Intel e suporta dois barramentos PCI (primário de 32 bits e secundário de 32 bits). Ambos os barramentos podem funcionar à frequência de 33 MHz.

A interface PCI na ADM-XRC está implementada com a ajuda do PCI 9080 da PLX Technology [PLX]. PCI 9080 suporta vários modos de configuração do barramento local e possibilita a realização de transferências DMA através de dois canais independentes.

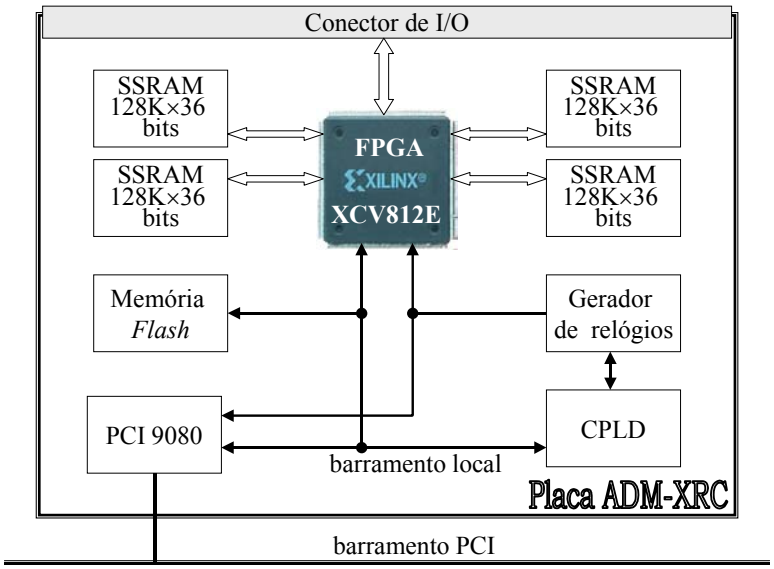


Fig. 3.12. Arquitectura da placa ADM-XRC.

A FPGA pode ser configurada de três maneiras diferentes: com a ajuda do barramento PCI e através da porta *SelectMAP* da FPGA [Xilinx_025]; a partir da memória *Flash*; através do conector JTAG externo. Há 4 bancos de memória SRAM de 256K×36 bits. A placa inclui também um conector de entrada/saída externo de 34 pinos (formato SCSI-2). Existem duas fontes de relógio. Um relógio (local) é utilizado para fornecer sinais de sincronização para a interface PCI e a FPGA. O segundo relógio serve para aplicações do utilizador. Ambos os relógios são programáveis (o local – de 400 KHz a 40 MHz; o de utilizador – de 400 KHz a 100 MHz) e podem ser utilizados por DLLs da FPGA.

O componente PCI 9080 permite fazer o mapeamento de dois espaços de endereçamento do barramento local (*S0* e *S1*). O espaço de endereçamento *S0* de 4M×32 bits permite ao utilizador aceder à FPGA e aos registos de controlo do barramento local. O espaço *S1* de 4M×8 bits possibilita acesso à memória *Flash* e à porta *SelectMap* da FPGA. Os registos de controlo do barramento local são implementados num CPLD. Estes registos controlam o acesso ao relógio, permitem gerar interrupções, configurar a FPGA, etc.

A interacção com a FPGA é feita com a ajuda da biblioteca de interface com a placa ADM-XRC que suporta a inicialização e configuração da FPGA, transferência de dados, processamento de interrupções e erros, gestão de relógio, etc.

3.7 Conclusões

A computação reconfigurável tornou-se uma área de investigação importante. Ao atribuir porções computacionalmente intensivas de uma aplicação ao hardware reconfigurável, pode-se acelerar significativamente a execução desta aplicação. Isto deve-se ao facto de sistemas reconfiguráveis combinarem vantagens das implementações baseadas em ASIC e das baseadas em software. Os circuitos resultantes são flexíveis podendo ser modificados em qualquer momento mesmo durante a execução da aplicação. Para além disso, estes circuitos são capazes de atingir melhor desempenho que o de software ao eliminar o *overhead* inerente a qualquer processador de uso geral.

Para atingir melhores resultados, os sistemas reconfiguráveis são normalmente compostos pela lógica reconfigurável interligada com um processador de uso geral. O processador executa operações que não podem ser realizadas eficientemente em lógica reconfigurável, enquanto as partes computacionalmente intensivas da aplicação são mapeadas em hardware. Em sistemas reconfiguráveis o nível de interligação dos componentes principais pode variar desde unidades funcionais configuráveis integradas no processador até FPGAs autónomas.

Configurações diferentes podem ser utilizadas em várias fases de execução de uma aplicação personalizando hardware a cada uma das fases. A reconfiguração em *run-time* permite implementar configurações maiores que os recursos de hardware disponíveis, dividindo os circuitos respectivos em circuitos mais pequenos que são utilizados sucessivamente. Devido aos atrasos associados à configuração, a reconfiguração deve ser realizada de uma maneira muito eficiente.

A implementação de uma aplicação num sistema reconfigurável não é uma tarefa simples. Esta envolve a programação a nível de software e o projecto da parte de hardware. A fim de atingir bons resultados no mapeamento de uma aplicação na FPGA é necessário explorar o paralelismo e operações específicas e optimizadas. Para além disso, a compilação de configurações de hardware deve ser efectuada rapidamente.

É de notar que embora já tenham sido propostas muitas arquitecturas eficientes que revelaram vantagens significativas da computação reconfigurável, ainda há muitas questões por responder. Em particular, os sistemas reconfiguráveis introduziram uma nova dimensão a nível de programação. Portanto, são necessárias tecnologias e ferramentas inovadoras que sejam capazes de cobrir todos os aspectos da programação e do uso eficiente de sistemas reconfiguráveis.

4 Problemas de optimização combinatória

Sumário

Neste capítulo é dada uma definição de problemas combinatórios. São apresentados alguns exemplos de problemas combinatórios, classificados de acordo com os modelos matemáticos empregues normalmente para a sua especificação. Destes modelos os mais usados são conjuntos, grafos, matrizes e funções lógicas. É importante notar que todos os modelos matemáticos considerados são mutuamente conversíveis uns nos outros.

Os problemas combinatórios abordados são os de partição de conjuntos, coloração e corte de grafos, minimização de funções booleanas, encontro de menores com as características específicas em matrizes, etc. Todos estes possuem inúmeras aplicações práticas. Assim a secção 4.3 é dedicada a exemplos de aplicações possíveis destacando as que surgem na área de desenvolvimento de sistemas digitais.

A secção 4.4 descreve os algoritmos que podem ser utilizados para resolver problemas de optimização combinatória. Nesta secção, primeiro é introduzida a terminologia necessária para a descrição de algoritmos. Todos os algoritmos foram divididos em duas classes de acordo com a qualidade das soluções que são capazes de assegurar. De seguida, é feita uma revisão de métodos pertencentes a cada uma das classes incluindo os algoritmos clássicos, tais como pesquisa local, programação dinâmica, etc., e os métodos mais recentes, tais como *simulated annealing* e os algoritmos evolutivos.

A secção 4.5 introduz brevemente o conceito de complexidade de algoritmos e a divisão de problemas combinatórios em várias classes, tais como *P*, *NP*, *NP-complete* e *NP-hard*. Finalmente, a secção 4.6 contém as conclusões.

4.1 Introdução

Os problemas combinatórios surgem em várias áreas tais como desenvolvimento de circuitos digitais [Micheli94], diagnóstico técnico [Zakrevski81], reconhecimento de padrões [Huang93], planeamento de circulação de transporte [Phillips81], etc. Uma das características distintivas destes problemas é que a sua resolução está ligada ao exame de elementos de um conjunto finito especificado pelos dados iniciais.

Um problema combinatório Π consiste num conjunto de instâncias D_Π e para cada instância $I \in D_\Pi$ existe um conjunto de soluções $S_{\Pi(I)}$. Os problemas combinatórios possuem 4 variedades principais que diferem pelo tipo de solução que se pretende encontrar:

1) Os *problemas de decisão* requerem a verificação se $S_{\Pi(I)}$ é o conjunto vazio e portanto possuem apenas duas respostas possíveis que são "sim" ou "não".

2) Os *problemas de pesquisa* são semelhantes com os problemas correspondentes de decisão: quando a resposta é "não" o algoritmo deve devolver esta resposta, mas quando a resposta é "sim" é preciso encontrar uma solução $x \in S_{\Pi(I)}$ que faz com que a resposta seja "sim".

3) Nos *problemas de enumeração* pede-se para encontrar o número total de soluções para as quais a resposta é "sim", i.e. descobrir o número de elementos no conjunto $S_{\Pi(I)}$.

4) Nos *problemas de optimização* é definida uma função de custo $aval_\Pi$ que atribui a cada instância $I \in D_\Pi$ e cada solução $x \in S_{\Pi(I)}$, um número racional $aval_\Pi(I, x)$ que reflecte a adequabilidade da solução x. Na resolução dum problema de optimização é preciso encontrar uma solução x^* para a qual a função de custo toma o valor mínimo (ou máximo), i.e. encontrar $x^* \in S_{\Pi(I)}$: $\forall x \in S_{\Pi(I)}, x \neq x^*$, $aval_\Pi(I, x) \geq aval_\Pi(I, x^*)$ (ou $aval_\Pi(I, x) \leq aval_\Pi(I, x^*)$).

Ilustremos os quatro tipos de problemas combinatórios com a ajuda do problema do caixeiro viajante:

Instância. É dado um conjunto finito de cidades: $C=\{c_1,c_2,...,c_n\}$, as distâncias para cada par de cidades: $(\forall\ c_i \in C)\ (\forall c_j \in C)\ (\exists\ d\ (c_i,c_j) \in Z^+)$, $i,j=1,...,n$, e um limite $B \in Z^+$.

Problema de decisão. Existe algum caminho que interliga todas as cidades de C (passando por cada uma delas apenas uma vez) cujo comprimento total é igual ou menor que B? Por outras palavras, existe uma cadeia de cidades $[c_{\sigma(1)}, c_{\sigma(2)},..., c_{\sigma(n)}]$ tal que a função seguinte é válida?

$$\left(\left(\sum_{i=1}^{n-1} d\ (c_{\sigma(i)}, c_{\sigma(i+1)})\right) + d\ (c_{\sigma(n)}, c_{\sigma(1)})\right) \leq B \qquad (1)$$

Problema de pesquisa. Encontrar uma cadeia de cidades $[c_{\sigma(1)}, c_{\sigma(2)},..., c_{\sigma(n)}]$ que satisfaz a função (1) ou responder que tal cadeia não existe.

Problema de enumeração. Calcular o número de cadeias (diferentes) de cidades que satisfazem a função (1).

Problema de optimização. Encontrar uma cadeia de cidades $[c_{\sigma(1)}, c_{\sigma(2)},..., c_{\sigma(n)}]$ que representa um caminho com o comprimento mínimo.

Os algoritmos desenvolvidos e utilizados para a resolução de problemas combinatórios podem ser avaliados com os dois critérios seguintes: a complexidade e a qualidade de soluções produzidas. De acordo com o critério da complexidade os algoritmos são divididos normalmente em duas classes: polinomiais e exponenciais. Segundo o critério de qualidade de soluções obtidas, os algoritmos dividem-se em *exactos* e *aproximados*. Os algoritmos exactos, que garantem a obtenção da solução exacta do problema, têm frequentemente complexidade muito elevada, o que impede a sua aplicação na prática. Por outro lado os algoritmos aproximados nem sempre levam à solução exacta mas asseguram uma aproximação bastante boa a esta.

4.2 Modelos matemáticos

Os problemas de optimização combinatória são normalmente formulados sobre *modelos matemáticos* tais como conjuntos, grafos, matrizes e funções lógicas. Analisemos alguns exemplos de problemas combinatórios típicos que surgem frequentemente em projecto de dispositivos digitais. Classificamos os problemas de acordo com os modelos matemáticos usados normalmente para a sua especificação [Skliarova01a].

4.2.1 Conjuntos

4.2.1.1 Obtenção de partição mínima

O problema é formulado da maneira seguinte. Num conjunto C é necessário encontrar uma partição π que é composta por um número mínimo de blocos n. Cada bloco C^{i}, i=1,...,n, desta partição deve corresponder a determinados requisitos e

$$\pi = \left\{C^{1},...,C^{\mathrm{n}}\right\},\quad C^{\mathrm{i}} \subseteq C,\quad C^{\mathrm{i}} \neq \varnothing,\quad C^{\mathrm{i}} \cap C^{\mathrm{j}} = \varnothing,\quad \bigcup_{i=1}^{n} C^{\mathrm{i}} = C,\quad \forall i, j = 1,...,n,\ i \neq j$$

4.2.2 Grafos

4.2.2.1 Coloração de grafos

Este problema é um dos mais conhecidos problemas combinatórios. Normalmente exige-se colorir um grafo usando um número mínimo de cores atribuindo a todos os vértices adjacentes cores diferentes [Zakrevski00, Diestel00]. A fig. 4.1 ilustra a coloração mínima dum grafo com 3 cores.

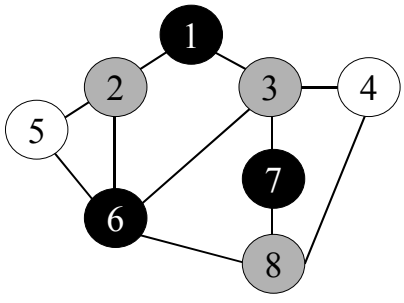


Fig. 4.1. Coloração mínima do grafo.

4.2.2.2 Determinação do caminho mais curto (longo)

O problema de encontro de caminhos num grafo também é muito conhecido [Diestel00, Aho83]. Este é formulado de modo seguinte. Há um grafo orientado e interpretado $G(V,E,W)$, onde V é um conjunto de vértices, E é um conjunto de arcos e W é um conjunto de pesos atribuídos a cada arco. É preciso encontrar o caminho mais curto ou mais comprido (em termos dos pesos de arcos) que liga um vértice inicial v_b com um vértice final v_e, b, e=1,...,$|V|$. Na fig. 4.2a está representado um grafo interpretado, se $v_b = v_2$ e $v_e = v_5$ então o caminho mais curto é o v_2-v_3-v_5, e o mais comprido é o v_2-v_5.

4.2.2.3 Partição em cliques

Uma clique dum grafo é um subgrafo completo. Um grafo diz-se decomposto em cliques quando o conjunto dos seus vértices fica dividido em subconjuntos que não se intersectam, cada um dos quais representa uma clique [Golumbic80]. O problema mais frequente é a procura da partição mínima. O grafo $G(V, E)$ na fig. 4.2b pode ser dividido em duas cliques $\{v_1, v_2, v_3\}$ e $\{v_4\}$.

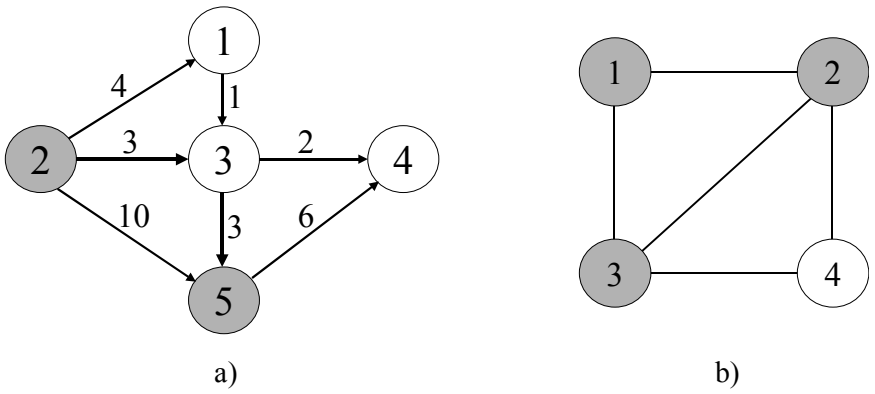


Fig. 4.2. O caminho mais curto de v_2 a v_5 é v_2-v_3-v_5 (a); Partição em cliques (b).

4.2.2.4 Corte

Este problema consiste em cortar um grafo em vários subgrafos de tal maneira que o número de vértices em cada um dos subgrafos não exceda k e o peso total de arestas eliminadas fique minimizado [Sklyarov84a]. Na fig. 4.3 está representado um exemplo de corte de um grafo com k=2.

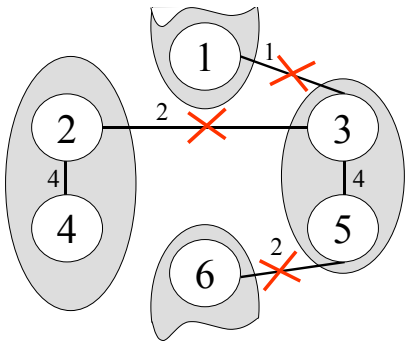


Fig. 4.3. Corte do grafo com k = 2.

4.2.3 Matrizes

Aqui consideramos apenas matrizes booleanas e ternárias, utilizadas para a especificação de relações binárias e ternárias arbitrárias.

4.2.3.1 Pesquisa de menores

Muitos problemas estão relacionados com o encontro de um menor (subconjunto de linhas ou colunas) com determinadas características numa matriz. Um dos mais conhecidos é o problema da obtenção da cobertura mínima, onde se procura descobrir um menor mínimo composto pelas colunas (linhas) que contenham em cada linha (coluna) pelo menos um elemento igual a *1* [Zakrevski71]. O problema de determinar o teste diagnóstico mínimo é um outro representante desta classe. Aqui pretende-se descobrir um menor mínimo, cujas linhas sejam todas diferentes (supõe-se que na matriz inicial estas também são diferentes) [Zakrevski98]. Consideremos a matriz booleana **B** representada na fig. 4.4. Neste caso o vector booleano [00110] representa a cobertura mínima (composta pela terceira e quarta colunas da matriz **B**) e o vector [11010] representa o teste diagnóstico mínimo.

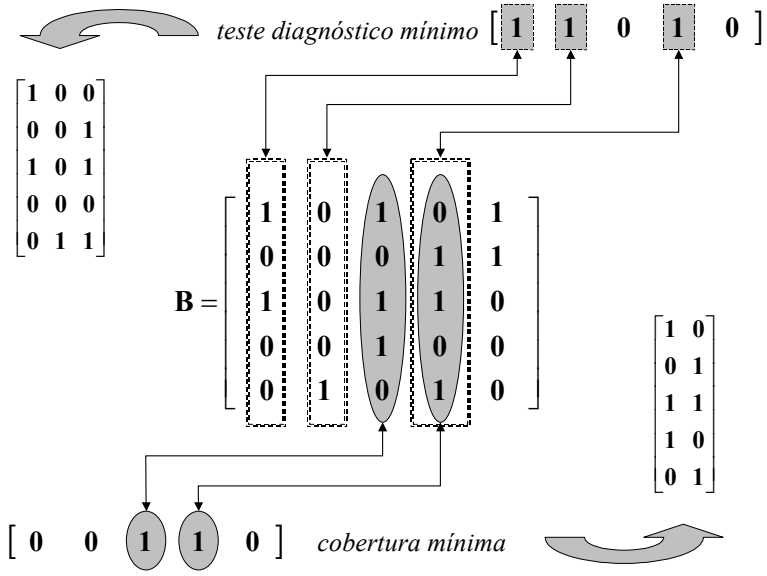


Fig. 4.4. Pesquisa de menores em matrizes.

4.2.3.2 Pesquisa de vectores

Os problemas deste tipo exigem encontro de um vector que esteja numa determinada relação com as linhas ou as colunas da matriz. Um dos exemplos é o seguinte: é necessário encontrar um vector que seja ortogonal a todas as linhas de uma dada matriz ou concluir que este não existe [Zakrevski81]. Neste caso, dois vectores consideram-se ortogonais se pelo menos numa das suas componentes um vector tiver o valor *0* e outro o valor *1*.

4.2.3.3 Construção de uma nova matriz

Muitos problemas exigem a construção de uma nova matriz **C** que deve estar numa determinada relação com uma dada matriz **B**. Um exemplo desta classe de problemas é o da determinação da

base disjuntiva mínima, onde é preciso construir a matriz **C** com o número mínimo de linhas de tal maneira que cada linha da matriz **B** seja igual à disjunção de várias linhas de **C** [Zakrevski90, Zakrevski81]. Ilustramos este problema com as duas matrizes **B** e **C** representadas na fig. 4.5.

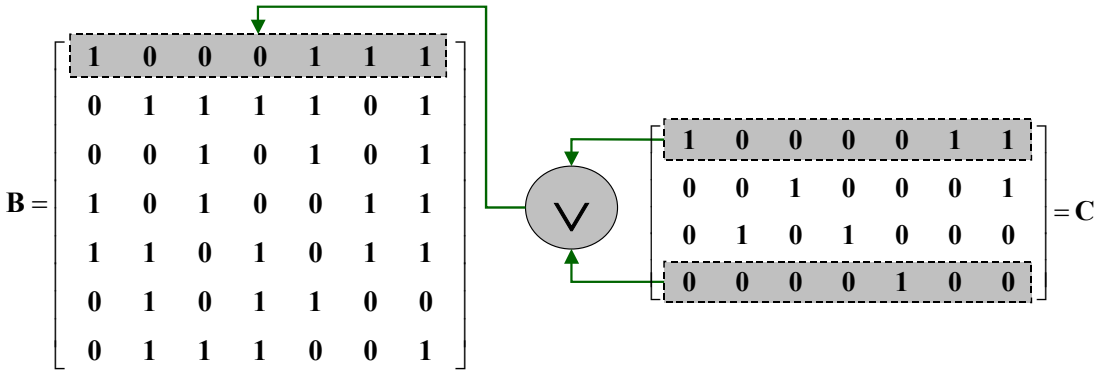


Fig. 4.5. A matriz **C** representa a base disjuntiva mínima da matriz **B**.

4.2.3.4 Minimização

Quando uma matriz representa a DNF de uma função (a transformação respectiva será demonstrada na secção 4.2.5) surgem frequentemente problemas de minimização da mesma. Por exemplo, para uma dada matriz ternária é preciso encontrar uma outra matriz equivalente com o número de linhas minimizado. Assim, a seguinte matriz **T** representa uma DNF inicial e a matriz **U** representa a DNF mais curta equivalente à inicial [Zakrevski81].

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & - \\ - & 0 & 1 & 1 \\ - & 0 & 1 & 0 \end{bmatrix} \qquad \mathbf{U} = \begin{bmatrix} 1 & 0 & - & - \\ 0 & 0 & 1 & - \end{bmatrix}$$

4.2.4 Funções booleanas

4.2.4.1 Minimização

É conhecido que qualquer função booleana pode ser representada em DNF [Handbook00], i.e. em forma de disjunção de um conjunto finito de conjunções elementares diferentes. O problema de obtenção de DNFs mais simples para uma dada função booleana é bastante complicado. Normalmente por critério de simplicidade toma-se o número total de argumentos (quanto à procura da DNF mínima) ou o número de conjunções elementares (quanto à procura da DNF mais curta) [Zakrevski81, Micheli94]. Por exemplo, para a função $f = x_1 x_2 \vee \bar{x}_1 x_2 \vee \bar{x}_1 \bar{x}_2$ obtemos a seguinte DNF mais curta $f = \bar{x}_1 \vee x_2$, que neste caso é igual à DNF mínima. Note-se que o problema de encontro da DNF mais curta coincide com o da determinação da matriz ternária mínima (com o número de linhas mínimo) equivalente à dada.

4.2.4.2 Decomposição

Este problema requer a representação de uma função booleana em forma de composição de várias funções booleanas cada uma com um menor número de argumentos [Sasao97]. Por exemplo, na bidecomposição disjunta é dada uma função $y=f(\mathbf{x})$, onde $\mathbf{x}=[x_1,...,x_n]$. É preciso encontrar dois subconjuntos **u** e **v** tais que $\mathbf{u}\neq\varnothing$, $\mathbf{v}\neq\varnothing$, $\mathbf{u}\subset\mathbf{x}$, $\mathbf{v}\subset\mathbf{x}$, $\mathbf{u}\cup\mathbf{v}=\mathbf{x}$, $\mathbf{u}\cap\mathbf{v}=\varnothing$, e três funções μ, λ e φ que satisfazem a equação seguinte: $f(\mathbf{x}) = \varphi(\mu(\mathbf{u}), \lambda(\mathbf{v}))$ conforme representado na fig. 4.6.

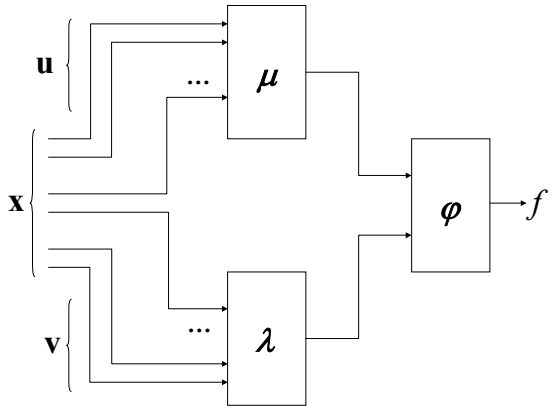


Fig. 4.6. Bidecomposição disjunta da função *f*.

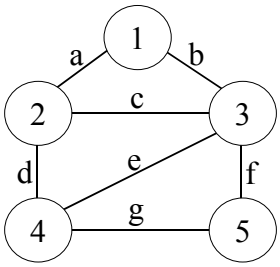
4.2.4.3 Satisfação booleana

Este problema consiste na verificação de existência de valores de argumentos tais que tornem uma função booleana igual a *1* [Micheli94]. Tautologia é um problema complementar que verifica se a função é equivalente a *1*, i.e. adquire o valor *1* independentemente dos valores dos seus argumentos. Notemos que este problema coincide com o da determinação de um vector que é ortogonal a cada linha da matriz que representa a DNF da função (caso seja impossível encontrá-lo, pode-se concluir que a função é uma tautologia).

4.2.5 Transformações mútuas de modelos

Com a ajuda de matrizes lógicas pode-se formular muitos problemas combinatórios e construir algoritmos para a sua resolução. As matrizes possuem uma estrutura que as torna bastante convenientes para o armazenamento e processamento em sistemas digitais, pelo que mostraremos como especificar grafos e funções booleanas através das matrizes respectivas [Skliarova01a].

Um grafo $G(V, E)$ pode ser representado em forma de uma matriz de adjacência ou de incidência. Uma matriz de adjacência **A** é uma matriz quadrada de dimensão $|V|\times|V|$, onde cada elemento a_{ij}, $i,j=1,...,|V|$, é igual a *1* caso o vértice *i* seja adjacente ao vértice *j*, e é igual a *0* caso contrário. Uma matriz de incidência **N** é uma matriz com as dimensões $|V|\times|E|$. Para um grafo não orientado cada elemento n_{ij}, $i=1,...,|V|$, $j=1,...,|E|$, da matriz é igual a *1* caso a aresta *j* seja incidente ao vértice *i*, e é igual a *0* caso contrário. Para um grafo orientado um elemento n_{ij} é igual a *1* se o vértice *i* é a origem do arco *j*, é igual a *-1* se o vértice *i* é o destino do arco *j*, e é igual a *0* caso contrário. Na fig. 4.7 está representado um grafo não orientado e as suas matrizes de incidência e de adjacência.



$$\mathbf{A}=\begin{array}{c}\begin{matrix}1&2&3&4&5\end{matrix}\\ \begin{bmatrix}0&1&1&0&0\\1&0&1&1&0\\1&1&0&1&1\\0&1&1&0&1\\0&0&1&1&0\end{bmatrix}\end{array}\begin{matrix}1\\2\\3\\4\\5\end{matrix}$$

$$\mathbf{N}=\begin{array}{c}\begin{matrix}a&b&c&d&e&f&g\end{matrix}\\ \begin{bmatrix}1&1&0&0&0&0&0\\1&0&1&1&0&0&0\\0&1&1&0&1&1&0\\0&0&0&1&1&0&1\\0&0&0&0&0&1&1\end{bmatrix}\end{array}\begin{matrix}1\\2\\3\\4\\5\end{matrix}$$

Fig. 4.7. Grafo e as matrizes de adjacência e de incidência respectivas.

As funções booleanas podem também ser representadas em forma de matrizes booleanas e ternárias. Quando uma função booleana está em DNF então as colunas da matriz correspondem aos argumentos da função e as linhas determinam as conjunções elementares. Por exemplo, para a função representada na fig. 4.8. é bastante fácil construir as respectivas matrizes booleana **B** e ternária **T**. Se a função tiver n variáveis e m conjunções elementares, então cada elemento da matriz ternária t_{ij}, i=1,…,m, j=1,…,n, é igual a:

- *1* – se a variável j entra na conjunção i;
- *0* – se a variável j entra na conjunção i com a negação;
- *'-'* (*don't care*) – se a variável j não entra na conjunção i.

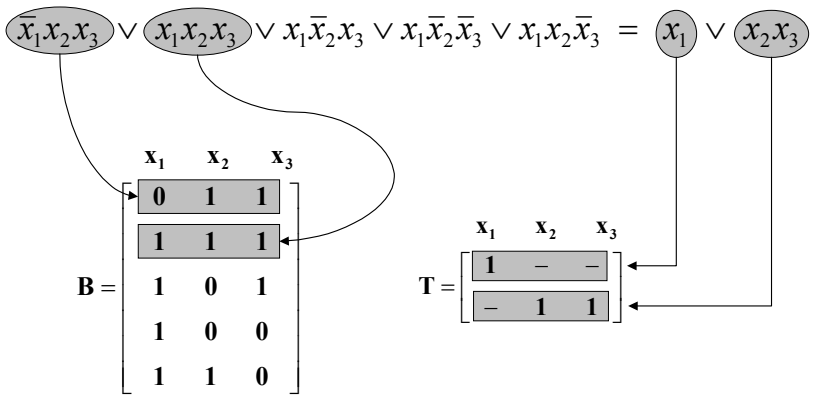


Fig. 4.8. Representação de uma função booleana em forma de matrizes.

Notemos que a matriz booleana **B** pode ser obtida a partir da matriz ternária **T** através da substituição das componentes que possuem valor *'-'* com todas as combinações possíveis de *0* e *1*, e da eliminação posterior de linhas iguais.

Um sistema de funções booleanas **y=f(x)** pode ser especificado com a ajuda de um par de matrizes **X** e **Y**. Se as funções booleanas estão em DNF então as colunas da matriz **X** determinam um

sistema de intervalos juntamente com as conjunções elementares respectivas, e as linhas da matriz **Y** correspondem às DNFs, i.e. cada elemento y_{ij} é igual a *1* quando a conjunção elementar *j* entra na DNF *i*. A fig. 4.9 ilustra a representação de um sistema de DNFs com as matrizes **X** e **Y**.

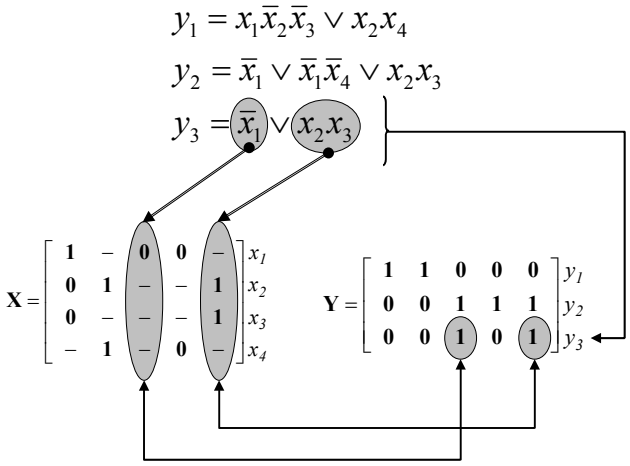


Fig. 4.9. Representação de um sistema de funções booleanas com a ajuda de matrizes.

4.3 Aplicações práticas de problemas combinatórios

Muitos problemas práticos reduzem-se à resolução de vários problemas combinatórios. Consideremos alguns exemplos destes problemas práticos destacando os que surgem na área do desenvolvimento de circuitos digitais [Skliarova01a].

É conhecido que uma das fases da síntese e optimização arquitectural consiste na distribuição de operações no tempo, i.e. na determinação do início de execução de cada operação com base na análise da sequência e interacção de todas as operações [Micheli94]. A possibilidade de resolução deste problema com restrições temporais dadas pode ser verificada aplicando o método de determinação do caminho mais comprido num grafo. Para tal constrói-se um grafo orientado e interpretado, cujos vértices representam as operações e cujos arcos correspondem às dependências entre as operações. O peso de cada arco é igual ao atraso da operação colocada na sua origem. Os arcos complementares especificam as restrições temporais, e as restrições máximas u_{ij}^{max} definem o intervalo máximo de tempo que pode decorrer entre duas operações *i* e *j*. Sendo assim, para que o problema tenha solução, o caminho mais comprido (o caminho com o peso maior) neste grafo entre cada par de vértices v_i e v_j deve ser $\leq u_{ij}^{max}$.

Na fase seguinte da síntese arquitectural efectua-se a atribuição de operações aos recursos existentes. Um dos objectivos a atingir nesta fase consiste na diminuição da área do circuito permitindo que várias operações distribuídas no tempo utilizem o mesmo hardware [Micheli94]. Neste caso duas operações consideram-se compatíveis caso não executem simultaneamente e possam ser implementadas em recursos do mesmo tipo. Constrói-se o grafo de compatibilidade, cujos vértices correspondem às operações e cujas arestas ligam os pares de operações compatíveis. Sendo assim o problema da distribuição óptima de recursos reduz-se à partição do grafo num número mínimo de cliques.

Para a minimização do número de condições lógicas numa máquina de estados finitos (FSM – *Finite State Machine*) usa-se a informação complementar sobre as suas alterações possíveis no processo de execução. Designemos por $A(x_i)$ o conjunto de estados cujas transições dependem da condição lógica x_i, i=1,..,L. Coloquemos em correspondência a cada conjunto $A(x_i)$ um vector $\mathbf{g}_i=(g_{i1},...,g_{iL})$, e representemos todos os vectores por uma matriz **G**, onde cada elemento g_{ij}, i,j=1,...,L, é igual a:

- *1* - se em todos os estados de $A(x_i)$ $x_j = 1$;
- *0* - se em todos os estados de $A(x_i)$ $x_j = 0$;
- *'-'* - se nos estados de $A(x_i)$ a condição x_j pode tomar tanto o valor *1* como o valor *0*.

Com base na matriz **G** constrói-se um grafo com os vértices correspondentes às condições lógicas. Dois vértices x_i e x_j são ligados por uma aresta caso $g_{ij}\neq g_{ji}$. Depois de colorir o grafo com o número mínimo de cores podemos unir (de um modo especial [Baranov81]) aquelas condições lógicas que correspondem aos vértices da mesma cor.

Na fase da síntese lógica de circuitos efectua-se a minimização de funções booleanas. O objectivo é construir um sistema de funções que sendo equivalente ao inicial corresponde aos requisitos especificados (por exemplo, a complexidade mínima ou o atraso de propagação mínimo, etc.).

O método de determinação das partições mínimas de um conjunto aplica-se na fase de mapeamento de elementos lógicos em células de uma dada biblioteca. Por exemplo, existe um sistema de funções booleanas em DNF com L argumentos, N funções e B conjunções diferentes. Designemos com L_i , i=1,...,B, o número de argumentos na conjunção i, então $L_{max} = \max_i L_i$. Suponhamos que é preciso implementar este sistema de funções booleanas em PLAs (s, t, q) (PLA com s entradas, t saídas e q linhas intermédias) e $L > s$, $N > t$, $B > q$, $L_{max} \le s$. Consideremos o conjunto de todas as conjunções elementares diferentes $E=\{e_1,...,e_B\}$. Então o problema reduz-se ao encontro de partição mínima no conjunto E que para cada bloco E^u faz cumprir as seguintes restrições: o número de argumentos em $E^u \le s$, o número de funções que dependem de $E^u \le t$ e $|E^u| \le q$ [Baranov86].

Frequentemente surge o problema de decomposição de uma FSM em várias sub-FSMs mais simples. Um dos objectivos que se coloca consiste em minimizar a quantidade de ligações entre sub-FSMs, o que é equivalente à minimização do número de transições entre os estados pertencentes a sub-FSMs diferentes [Sklyarov84a, Hachtel96]. Para resolver este problema é preciso cortar o diagrama de transições de estados da FSM inicial em sub-diagramas minimizando o número de sub-FSMs e o número de arcos cortados.

Suponhamos que temos um conjunto de vectores booleanos **Y** e precisamos de construir um *array OR* (o mais simples possível) que implemente estes vectores. Pede-se também arranjar os vectores de entrada **X** respectivos que serão transformados em dados vectores de saída. Notemos que a estrutura de um *array OR* pode ser especificada com a ajuda de uma matriz **B**, e a sua funcionalidade descreve-se com a equação $\mathbf{Y}=\mathbf{B}\times\mathbf{X}$ [Zakrevski81]. Sendo assim, temos de encontrar as matrizes booleanas **B** (com o número mínimo de colunas k) e **X** (com o mesmo número de linhas k) que satisfaçam a equação $\mathbf{Y}=\mathbf{B}\times\mathbf{X}$. Neste caso cada coluna da matriz **Y** é igual à disjunção de várias colunas da matriz **B** (a matriz **X** indica quais). Portanto o problema de encontro da matriz **B** reduz-se à determinação da base disjuntiva mínima da matriz **Y**.

Às vezes surge o problema de determinação de causas a partir dos efeitos que estas provocam (por exemplo, em diagnóstico). As causas abrangem os fenómenos cuja observação é difícil ou mesmo impossível. Por outro lado, os efeitos são directamente detectáveis. Contudo a detecção dos efeitos tem um custo associado, por isso o seu número deve ser restringido. Sendo assim, surge o problema de determinar o subconjunto mínimo de efeitos cuja observação permite descobrir a causa respectiva. A relação entre os conjuntos de causas e efeitos pode ser representada em forma de uma matriz booleana **X** onde cada elemento x_{ij} é igual a *1* se a causa *j* provoca o efeito *i*, e é igual a *0* caso contrário. Então o problema reduz-se a determinar o teste diagnóstico mínimo para a matriz **X** [Zakrevski81].

4.4 Algoritmos

Analisemos algoritmos que são utilizados para a solução de problemas combinatórios. Os três conceitos básicos inerentes a qualquer abordagem algorítmica são *a representação*, *o objectivo* e *a função de avaliação* [Michalewicz00].

A *representação* codifica as soluções-candidatas de uma determinada maneira. Por exemplo, para o problema do caixeiro viajante de *n* cidades, uma representação evidente de uma solução seria a permutação de números naturais 1,...,*n*. É de notar que para cada problema, a representação de soluções bem como a interpretação correspondente implicam o espaço de pesquisa e o sua dimensão [Michalewicz00]. Portanto a selecção do espaço de pesquisa adequado é uma tarefa de grande importância. Note-se também que para muitos problemas, as soluções de interesse só constituem um subconjunto *F* do espaço de pesquisa *S* (ver fig. 4.10). Estas soluções satisfazem todas as restrições do problema e são referenciadas por *soluções possíveis*.

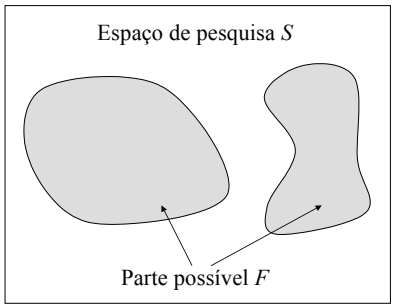


Fig. 4.10. Espaço de pesquisa *S* e a sua parte possível *F*.

O *objectivo* é uma expressão matemática que declara o que se pretende atingir. Considerando o mesmo problema do caixeiro viajante, o objectivo típico é minimizar a distância total percorrida pelo caixeiro.

A *função de avaliação* (*aval*) é normalmente um mapeamento do espaço de soluções possíveis num conjunto de números em que a cada solução potencial é atribuído um valor numérico que indica a sua qualidade. O objectivo de função de avaliação é possibilitar a comparação de soluções alternativas. Voltando novamente ao problema do caixeiro viajante, podemos definir uma função de avaliação que mapeie cada caminho na sua distância correspondente. Sendo assim, torna-se possível comparar os caminhos diferentes.

4.4.1 Terminologia

Num problema de optimização é dado um espaço de pesquisa S e a sua parte possível $F \subseteq S$, e pretende-se encontrar uma solução $x \in F$ tal que $aval(x) \geq aval(y)$ para cada $y \in F$. O ponto x que satisfaz esta condição, chama-se a solução óptima *global*.

De facto às vezes é bastante difícil encontrar a solução óptima global para um problema. Contudo, se nos concentrarmos num subconjunto relativamente pequeno do espaço de pesquisa, torna-se mais fácil encontrar a melhor solução (local) porque o número de alternativas a analisar é menor. Este subconjunto que cerca qualquer ponto x no espaço de pesquisa é denotado por $N(x)$ e é referenciado por *vizinhança* deste ponto. A vizinhança $N(x)$ do ponto x inclui todos os pontos do S que são próximos (segundo alguma medida) a x (ver fig. 4.11).

A vizinhança pode ser definida de várias maneiras. As duas possibilidades mais utilizadas são a distância entre x e todos os pontos em $N(x)$ que deve ser menor que um valor pré-estabelecido, ou um mapeamento que define a vizinhança para cada ponto $x \in S$.

A noção de vizinhança permite definir o conceito do *óptimo local*. Uma solução possível $x \in F$ é o *óptimo local* na vizinhança $N(x)$ se $aval(x) \geq aval(y)$ para cada $y \in N(x)$. Muitas estratégias de pesquisa baseiam-se na estatística da vizinhança de um determinado ponto e consequentemente estas são denominadas por *métodos de pesquisa local*.

As técnicas de pesquisa eficientes devem assegurar um balanço apropriado entre a exploração da vizinhança das melhores soluções encontradas e a exploração do espaço de pesquisa [Michalewicz00]. Por exemplo, os algoritmos de pesquisa local incidem sobre a exploração das melhores soluções encontradas a fim de atingir algum aperfeiçoamento, e acabam por não analisar grandes regiões do espaço de pesquisa S. Ao contrário disso, a pesquisa aleatória explora o espaço de pesquisa a fundo, mas não faz a análise cuidadosa das regiões promissoras do S.

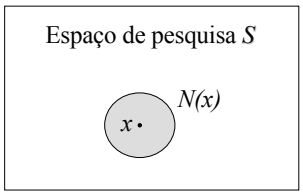


Fig. 4.11. Espaço de pesquisa S, solução possível x e a sua vizinhança $N(x)$.

A seguir descrevemos os tipos principais de algoritmos utilizados para a solução de problemas combinatórios. Dividimos os algoritmos em duas classes de acordo com a qualidade de soluções que estes são capazes de encontrar.

4.4.2 Algoritmos exactos

4.4.2.1 Pesquisa exaustiva

A pesquisa exaustiva examina todas as soluções possíveis no espaço de pesquisa até que seja encontrada a solução óptima global. Caso o valor do óptimo global seja desconhecido, o algoritmo implica a revisão de todas as soluções potenciais. Devido ao facto do tamanho do espaço de

pesquisa para problemas práticos ser enorme, os algoritmos exaustivos dificilmente podem ser aplicados. Contudo, estes são muito simples, pois o único requisito é poder gerar todas as soluções possíveis de um modo sistemático. Para além disso existem maneiras de reduzir o número de revisões necessárias.

A fim de gerar soluções possíveis de uma maneira sistemática recorre-se frequentemente à construção de uma *árvore de pesquisa* representada na fig. 4.12. A raiz da árvore de pesquisa corresponde à situação inicial e reflecte os dados iniciais do problema. Os vértices restantes representam várias situações que podem ser atingidas durante a pesquisa da solução. Os arcos da árvore especificam os passos do algoritmo que foram executados. Inicialmente, a árvore é desconhecida e só se constrói durante o processo de pesquisa. Uma característica distintiva desta abordagem é que em cada vértice da árvore de pesquisa resolve-se o mesmo problema, sendo apenas os dados de entrada modificados de vértice em vértice. Deste modo, o problema completo reduz-se à execução dum grande número de operações repetidas sobre um conjunto de dados periodicamente modificável.

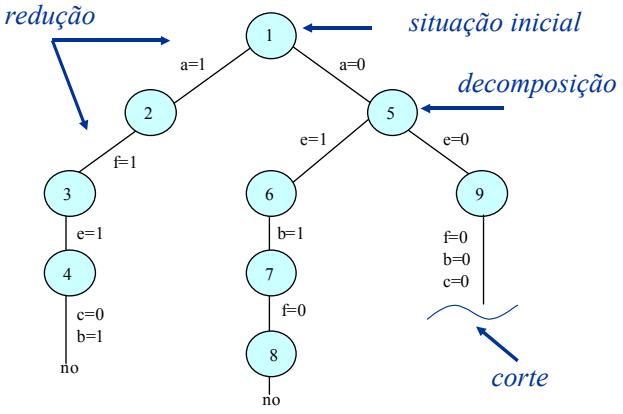


Fig. 4.12. Árvore de pesquisa.

Obviamente, o método de pesquisa exaustiva não pode ser utilizado para a solução de problemas complexos pois requer tempo de processamento muito longo. Consequentemente, é necessário aplicar algumas técnicas de optimização capazes de reduzir o número de situações a examinar. Uma das técnicas mais conhecidas é a de corte (*tree pruning*). Ao atingir uma dada situação devemos determinar todas as variantes possíveis da continuação da pesquisa, i.e. todos os arcos de saída. É evidente que é necessário reduzir de algum modo o seu número para que a solução seja encontrada o mais depressa possível. Normalmente esta redução é baseada no corte dos ramos que não alteram a situação actual, na rejeição das variantes já analisadas anteriormente e em parar de procurar aquelas soluções cujo custo no ponto actual já se tornou maior do que o custo de alguma das soluções já encontradas. Às vezes é possível aplicar métodos orientados para o problema que exploram a informação específica à instância obtida durante a pesquisa, e permitem cortar regiões grandes da árvore de pesquisa. Um exemplo desta técnica é a análise de conflitos e o retrocesso não cronológico, utilizados amplamente em algoritmos de solução do problema de satisfação booleana [Moskewicz01, Silva99].

Uma outra técnica que aumenta significativamente a eficácia da pesquisa é a *redução*. Nesta técnica uma situação actual substitui-se por uma outra situação mais simples, o que permite reduzir o número de operações necessárias para analisar o conjunto de variantes resultantes desta situação

actual sem sacrificar a solução óptima global. Os modos concretos de redução são extremamente diversos. Por exemplo, é preciso encontrar a cobertura mínima de uma matriz booleana. Neste caso a matriz inicial pode ser diminuída através da eliminação de linhas dominantes e das colunas dominadas (se procuramos a cobertura composta por colunas) [Zakrevski81], i.e. a simplificação da situação consiste na diminuição das dimensões da matriz. Um exemplo clássico desta técnica é o da determinação das raízes de uma equação pelo método da bissecção do intervalo, em que a simplificação da situação consiste na diminuição do intervalo no qual se procura a raiz da equação.

Para além da técnica de redução é necessário considerar o processo da decomposição de situações. Neste caso uma das situações torna-se crítica e por esta razão deve ser substituída por um conjunto de situações mais simples, cada uma das quais deve ser examinada. É de notar que convém decompor no menor número possível de situações mais simples. O processo de decomposição é representado por ramificação da árvore de pesquisa. Por exemplo, temos uma função booleana $f(x_1,...,x_n)$ e pede-se para encontrar um vector booleano **x** que faz com que a função fique igual a *1*. De acordo com a expansão de Shannon [Murgai95] podemos substituir uma situação corrente por duas, escolhendo uma variável x_i, i=1,...,n, e supondo que numa situação esta variável fica igual a *1* e noutra fica igual a *0*:

$$f(x_1, ...,x_n) = x_i f(x_1, ...,1, . .., x_n) \vee \overline{x}_i f(x_1, ...,0, . .., x_n)$$

A seguir analisamos a primeira situação. Se esta nos levar à solução com a qualidade satisfatória, o processo da pesquisa para; caso contrário teremos de examinar a segunda situação.

Às vezes consegue-se ordenar as situações nas quais se decompõe a situação corrente, i.e. pode-se analisar algumas características do problema e com base nesta informação escolher uma situação que tem maiores probabilidades de nos levar à solução. Isto aumenta significativamente a eficiência do algoritmo. Por exemplo, quando procuramos um vector ortogonal a cada linha de uma dada matriz e não podemos reduzir uma situação actual, a única saída consiste no exame sequencial dos valores possíveis de todas as componentes desconhecidas deste vector. Contudo, para encontrar a solução o mais rápido possível, recomenda-se escolher primeiro a componente que corresponde à coluna mais determinada da matriz, i.e. à coluna que tem o número mínimo de valores '-'.

Na fig. 4.13 apresenta-se uma versão do algoritmo exaustivo que é *depth-first search*. Este método corresponde à análise dos vértices da árvore de pesquisa pela ordem ilustrada na fig. 4.12.

```
Depth-first (vértice i)
{
    visitar i;
    for (cada descendente j do vértice i)
    {
          Depth-first (j);
    }
}
```

Fig. 4.13. Pseudocódigo da pesquisa *depth-first*.

Neste caso constrói-se uma árvore de pesquisa que é percorrida começando da sua raiz a analisando recursivamente todos os vértices adjacentes. Durante o processo de pesquisa algumas regiões do

espaço de pesquisa podem ser eliminadas ao verificar se um vértice pode conter a solução. Se pode conter, a pesquisa continua para frente; caso contrário, o controlo volta ao "pai" deste vértice (este processo é referenciado por retrocesso - *backtracking*).

4.4.2.2 Dividir-e-conquistar

Uma das técnicas utilizadas na resolução de problemas complexos é dividi-los num conjunto de subproblemas cuja solução contribui para a construção da solução completa [Kreher99]. Obviamente, a aplicação desta técnica só é razoável se o tempo de decomposição e de solução de cada um dos subproblemas é menor que o tempo necessário para resolver o problema inicial. Os algoritmos desta classe são normalmente implementados de maneira recursiva. Exemplos de aplicação desta técnica são os bem conhecidos algoritmos de ordenação dos elementos de um conjunto *MergeSort* e *QuickSort* [Handbook00].

O pseudocódigo do método *dividir-e-conquistar* está representado na fig. 4.14. O problema original *P* é decomposto em *n* subproblemas cada um dos quais é posteriormente também decomposto em subproblemas de maneira recursiva. Este processo continua até que o tamanho de subproblemas fique tão pequeno que estes podem ser resolvidos de maneira trivial. A partir deste ponto começa o movimento para trás, i.e. as soluções obtidas são combinadas a fim de construir soluções dos subproblemas maiores.

```
Divide_and_Conquer (problema P)
{
    dividir o problema P em sub-problemas P1,...,Pn;

    for (unsigned i = 1; i <= n; i++)
    {
          if (tamanho do Pi < ρ)
                solução si = resolver Pi;
          else
                solução si = Divide_and_Conquer (Pi);
    }

    construir a solução final a partir das soluções s1,...,sn;
}
```

Fig. 4.14. Pseudocódigo da técnica *dividir-e-conquistar*.

4.4.2.3 Programação dinâmica

A programação dinâmica baseia-se também na ideia de divisão de um problema em subproblemas. Primeiro são resolvidos os subproblemas mais simples e no fim do algoritmo obtém-se a solução global. Sendo assim, a programação dinâmica segue a estratégia "de baixo para cima" (*bottom-up*). A diferença principal entre a programação dinâmica e o método *dividir-e-conquistar* consiste em que esta nunca resolve o mesmo subproblema mais do que uma vez. As soluções a cada um dos subproblemas são memorizadas e utilizadas posteriormente para construir as soluções dos subproblemas maiores. O esqueleto dum programa dinâmico depende do problema particular.

Na fig. 4.15 apresentamos um exemplo de aplicação desta técnica para calcular os números de Fibonacci [Handbook00]. É de notar que caso recorrêssemos ao método *dividir-e-conquistar*,

precisávamos de efectuar *2n-1* chamadas recursivas da função respectiva enquanto a programação dinâmica requer o cálculo de apenas *n* números. Contudo, para alguns casos a aplicação da programação dinâmica pode ser muito intensiva computacionalmente.

```
unsigned Fibonacci (unsigned n)
{
     F[0] =0;
     F[1] =1;

     for (unsigned i = 2; i <= n; i++)
        F[i] = F[i-1] + F[i -2];

     return F[n];
}
```

Fig. 4.15. Pseudocódigo da aplicação da técnica de programação dinâmica para calcular os números de Fibonacci.

4.4.3 Algoritmos aproximados

4.4.3.1 Algoritmos *greedy*

Os algoritmos *greedy* operam sobre as soluções parciais e constroem a solução completa passo a passo. Em cada passo é seleccionada uma *variável de decisão* e a esta atribui-se um valor tal que implique o maior proveito (de acordo com alguma heurística). Obviamente, o facto de tomar as decisões óptimas em cada passo do algoritmo não assegura o encontro do óptimo global. Na fig. 4.16. apresentamos o pseudocódigo do algoritmo *greedy* que pode ser utilizado para resolver o problema de coloração dum grafo com *n* vértices.

```
void ColoracaoGrafoGreedy()
{
   seleccionar a primeira cor disponível c;

   while (existem vértices não pintados)
   {
      for (unsigned i = 0; i < n; i++)
       {
          if (vertices[i] não está pintado)
             if (vertices adjacentes não estão pintados com a cor c)
                   pintar o vertices[i] com a cor c;
       }
      actualizar c seleccionando a cor seguinte;
   }
}
```

Fig. 4.16. Pseudocódigo do algoritmo *greedy* que pode ser utilizado para colorir um grafo com *n* vértices.

Os algoritmos *greedy* são muito simples de implementar e rápidos a executar. Contudo estes são pouco eficientes e, em geral, incapazes de encontrar boas soluções. Uma característica comum a todos os algoritmos *greedy* é que estes constroem apenas uma solução potencial portanto a ordem inicial dos elementos (dos quais se compõe a solução) influencia drasticamente o resultado atingido.

4.4.3.2 Pesquisa local

A pesquisa local explora cuidadosamente a vizinhança de um ponto no espaço de pesquisa. O processo inclui normalmente os quatro passos seguintes:

1) Seleccionar uma solução possível no espaço de pesquisa, avaliá-la e designá-la por *solução corrente*.
2) Efectuar uma transformação na solução corrente a fim de gerar uma solução nova. Avaliar a solução obtida.
3) Caso a solução nova seja melhor (segundo alguma medida) que a corrente, substituir a solução corrente pela solução nova. Caso contrário, rejeitar a solução nova.
4) Repetir os passos 2 e 3 enquanto se verificam os melhoramentos na solução corrente.

A diferença principal entre os vários algoritmos de pesquisa local está no tipo de transformação aplicada. Dado que a solução nova é seleccionada na vizinhança da solução corrente, os métodos de pesquisa local apresentam um compromisso entre o tamanho da vizinhança e a eficiência de pesquisa. Caso o tamanho da vizinhança seja relativamente pequeno, o algoritmo é capaz de analisar todos os pontos-vizinhos com muita rapidez. Contudo, a probabilidade de parar num óptimo local torna-se maior. Se aumentarmos o tamanho da vizinhança (no caso extremo este pode atingir o tamanho do espaço de pesquisa), é mais provável que encontremos a solução óptima, contudo o número de avaliações pode atingir um valor inaceitável, inviabilizando deste modo a aplicação do algoritmo na prática.

Hill-climbing - Métodos de procura do extremo

Os métodos de procura do extremo bem como todos os métodos de pesquisa local utilizam a técnica de melhoramento iterativo. Esta técnica é aplicada a um ponto x no espaço de pesquisa. Durante cada iteração um ponto novo y é seleccionado na vizinhança de x. Caso a avaliação do ponto y produza os resultados melhores que a avaliação do ponto x, o ponto y substitui o ponto x. Caso contrário, é seleccionado e avaliado qualquer outro ponto na vizinhança $N(x)$. O processo termina quando se torna impossível aperfeiçoar a solução corrente ou se esgota o tempo disponível.

Obviamente, os métodos de procura do extremo só são capazes de encontrar os óptimos locais e os resultados obtidos dependem fortemente da selecção do ponto inicial. A fim de ultrapassar parcialmente estas restrições, estes métodos são executados várias vezes com pontos iniciais diferentes. Neste caso a probabilidade de encontrar o óptimo global aumenta. Os pontos iniciais são normalmente escolhidos de maneira aleatória, segundo um padrão regular ou com base em alguma informação adicional disponível.

Uma versão do método de procura do extremo (*steepest-ascent*) está representada na fig. 4.17. Inicialmente, selecciona-se uma solução possível *best*. A seguir, executam-se *MAX* iterações em cada uma das quais se escolhe um ponto x, analisa-se a sua vizinhança $N(x)$, encontra-se o óptimo local, e caso o óptimo local seja melhor que a solução *best*, este substitui o ponto *best*.

As características principais dos métodos de procura do extremo são as seguintes [Michalewicz00]:

- Os métodos normalmente são capazes de encontrar somente os óptimos locais.

- Não existe informação alguma que represente o desvio do óptimo local encontrado do óptimo global.
- A solução obtida depende da configuração inicial.
- Em geral, é impossível estimar o tempo de computação.
- Os métodos são muito simples de aplicar.

```
MetodoProcuraExtremo ()
{
    unsigned iter = 0;
    inicializar best;

    do
    {
           bool LocalMaximum = false;
           seleccionar ponto x;
           avaliar x;
           do
           {
                  encontrar todos os pontos em N(x);
                  avaliá-los;
                  seleccionar o ponto y que tem o melhor valor
                  da função de avaliação;

                  if ( aval(y) > aval (x) )
                         x = y;
                  else
                         LocalMaximum = true;

           } while (!LocalMaximum);

           iter++;

           if ( aval(x) > aval (best) )
                  best = x;

    } while (iter < MAX);

    return best;
}
```

Fig. 4.17. Pseudocódigo da versão *steepest-ascent* do método de procura do extremo.

Simulated annealing

O método *simulated annealing* baseia-se na analogia adoptada da termodinâmica [Kirkpatrick83]. A estrutura geral do algoritmo está representada na fig. 4.18 onde se pode ver que este é parecido com o método de procura do extremo. Contudo existem diferenças importantes que são as seguintes:

- No ciclo interior do algoritmo *simulated annealing* é analisado apenas um ponto e o ciclo é executado até que seja atingida alguma condição de terminação enquanto no método de procura do extremo este ciclo requer o exame de todos os pontos existentes na vizinhança e, consequentemente, o encontro do óptimo local.

- O ponto *y* que substitui o ponto *x*, não é necessariamente melhor. Mesmo que o novo ponto *y* seja pior que o ponto *x*, este pode ser aceite com a probabilidade baseada na temperatura *T* e na diferença entre os pontos *x* e *y*.
- O parâmetro *T* é periodicamente actualizado. Inicialmente, o valor deste parâmetro é grande fazendo com que a pesquisa seja quase aleatória. Nas iterações subsequentes, o valor *T* é gradualmente diminuído o que resulta no comportamento do algoritmo semelhante ao método de procura do extremo.
- Para escapar do óptimo local o método de procura do extremo selecciona um novo ponto *x* e começa a pesquisa do início, enquanto o algoritmo *simulated annealing* permite diminuir sucessivamente a qualidade da solução corrente possibilitando deste modo afastar-se do óptimo local e explorar regiões novas no espaço de pesquisa.
- *Simulated annealing* é um método probabilístico pois pode encontrar soluções diferentes mesmo quando as configurações iniciais são idênticas.

```
SimulatedAnnealing ()
{
    unsigned iter = 0;
    inicializar T;

    seleccionar ponto x;
    avaliar x;
    best = x;

    do
    {
        do
        {
            seleccionar um ponto y em N(x);
            avaliá-lo;

            if ( aval(y) > aval (x) )
            {
                   x = y;
                   if (aval(x) > aval(best))
                          best = x;
            }
                                                  aval(x)−aval(y)
            else  com probabilidade de  e        ---------------
                                                        T
                   x = y;

        } while (!CondiçãoTerminação);

        T = actualizar (T, iter);
        iter++;

    } while (!CondiçãoParagem);

    return best;
}
```

Fig. 4.18. Pseudocódigo do algoritmo *simulated annealing*.

Tabu search

O algoritmo *tabu search* também permite que o novo ponto *y* seleccionado na vizinhança do ponto *x* seja aceite mesmo que seja pior que *x*. A condição de aceitação é baseada na história da pesquisa [Glover89, Glover90]. Portanto, neste caso é a memória que força explorar áreas novas no espaço de pesquisa. As soluções que foram examinadas recentemente são adicionadas à memória e tornam-se proibidas (*tabu*) evitando deste modo ciclos possíveis no processo de pesquisa. O pseudocódigo do algoritmo está representado na fig. 4.19. É de notar que *tabu search* (na sua versão base) é um método determinístico, embora seja possível a inclusão de elementos probabilísticos [Glover89].

```
TabuSearch ()
{
    unsigned iter = 0;

    inicializar best;

    do
    {
           seleccionar ponto x;
           avaliar x;

           do
           {
                  encontrar todos os pontos em N(x);
                  avaliá-los;

                  seleccionar o ponto y que tem o melhor valor da
                  função de avaliação e que não esteja proibido;

                  actualizar a memória (proibir o ponto y);

                  x = y;
                  if ( aval(x) > aval(best) )
                         best = x;

           } while (!CondiçãoTerminação);

           iter++;

    } while (iter < MAX);

    return best;
}
```

Fig. 4.19. Pseudocódigo do algoritmo *tabu search*.

4.4.3.3 Algoritmos evolutivos

Todos os métodos considerados acima em cada momento só operam uma única solução. Esta solução ou serve de base para a exploração futura do espaço de pesquisa, ou representa a *solução parcial* que está a ser construída. Ao contrário disso, os algoritmos evolutivos trabalham com uma *população* de soluções e incorporam o conceito de competição [Holland75, Goldberg89]. Estes métodos têm na sua origem as ideias de selecção natural e evolução. A função de avaliação determina a qualidade relativa de todas as soluções e aquelas que possuem a melhor qualidade

servem de base para a geração seguinte. A geração seguinte compõe-se dos pontos que estão na vizinhança das soluções seleccionadas (de acordo com algum critério) da geração anterior bem como na vizinhança de pares daquelas soluções (pode-se considerar também trios, quartetos, etc.). As soluções menos promissoras eliminam-se. Sendo assim, ocorre uma competição entre as soluções geradas e as soluções-pais, formando os vencedores a geração seguinte. Após uma série de gerações a pesquisa converge para uma solução quase-óptima. O pseudocódigo de um algoritmo evolutivo está representado na fig. 4.20. Em mais detalhe estes algoritmos são considerados no capítulo 7.

```
AlgoritmoEvolutivo ()
{
    unsigned iter = 0;
    inicializar a população;
    avaliar todas as soluções da população;
    best = melhor solução da população;

    do
    {
           seleccionar soluções para reprodução;

           produzir soluções novas;
           avaliá-las;

           criar uma população nova composta pelas soluções-
           descendentes e soluções-pais seleccionadas com alguma
           probabilidade baseada na sua qualidade relativa;

           if ( aval(best) < aval(melhor solução da população nova) )
                  best = melhor solução da população nova;

           iter++;

    } while (iter < MAX);

    return best;
}
```

Fig. 4.20. Pseudocódigo do algoritmo evolutivo.

4.5 Análise de algoritmos

Frequentemente, os problemas combinatórios têm dimensões muito elevadas. Por essa razão, o desenvolvimento de algoritmos rápidos é de grande importância. Para estimar quantos recursos (em termos do tempo de execução e da memória necessária) requer um algoritmo utilizam-se métodos matemáticos especiais. Estes permitem prever a adequabilidade dum dado algoritmo sem ter de o implementar. Isto é importante e útil porque permite poupar trabalho, pois torna possível comparar vários algoritmos sem necessidade de implementação de cada um deles.

A análise dum algoritmo descreve a maneira como o tempo de execução do algoritmo se comporta em função do tamanho da instância do problema. O *tamanho* duma instância do problema *n* é a quantidade dos dados de entrada necessários para descrever esta instância. O aumento do tempo de execução dum algoritmo em função do tamanho de instância é referenciado por *complexidade* do

algoritmo. A *função de complexidade temporal* dum algoritmo expressa o tempo que este requer para resolver cada instância do problema.

Para especificar a complexidade utilizam-se normalmente as notações O, Θ e Ω [Handbook00]. Definamos a notação Θ de maneira seguinte: para as duas funções de complexidade $f:R\rightarrow R$ e $g:R\rightarrow R$, $f(n)$ é $\Theta(g(n))$ se existirem constantes $c_1,c_2 > 0$, $n_0 \geq 0$ tais que

$$0 \leq c_1 \times |g(n)| \leq |f(n)| \leq c_2 \times |g(n)| \qquad \text{para todos } n \geq n_0$$

Se $f(n)$ é $\Theta(g(n))$, então as funções f e g têm a mesma complexidade. De acordo com a função de complexidade, os algoritmos são normalmente divididos em duas classes. *Os algoritmos de tempo polinomial* têm a sua função de complexidade definida como $\Theta(p(n))$, onde p é uma função polinómia. Todos os algoritmos que não satisfazem esta restrição são denominados por *algoritmos de tempo exponencial* (é de notar que esta definição inclui também algumas funções de complexidade que não são normalmente consideradas exponenciais, por exemplo $n^{\log n}$ [Garey79]).

Normalmente, calculam-se dois tipos de complexidade de algoritmos: a complexidade no caso pior (*worst-case complexity*) que revela o tempo máximo de execução dum algoritmo, e a complexidade média (*average-case complexity*) que é determinada ao calcular o tempo que o algoritmo requer para todas as permutações possíveis dos dados de entrada e depois calcular a média destes valores. É de notar que existem algoritmos que embora possuam complexidade exponencial no caso pior, conseguem demonstrar resultados muito bons na prática.

A complexidade dos algoritmos permite dividir os problemas combinatórios em várias classes. É de salientar que esta divisão é orientada aos problemas de *decisão*, i.e. aos problemas que requerem uma resposta em forma "sim" ou "não". Descrevemos brevemente estas classes.

A classe *P* abrange todos os problemas de decisão para os quais existem algoritmos de complexidade polinómia.

A classe *NP* abrange todos os problemas de decisão que podem ser resolvidos em tempo polinomial com a ajuda de algoritmos não determinísticos. Por outras palavras, esta classe inclui problemas de decisão, para os quais caso a resposta seja "sim" é possível verificar se alguma solução produz a resposta "sim" em tempo polinomial. $P \subseteq NP$ e acredita-se que a classe *NP* é maior que a classe *P*, contudo isto não está provado [Kreher99].

A classe *NP-complete* é composta pelos problemas "mais difíceis" da classe *NP*. Em termos mais formais, um problema Π é *NP-complete* se $\Pi \in NP$ e para qualquer problema $\Pi' \in NP$, $\Pi' \propto \Pi$, i.e. existe a transformação polinómia de Π' para Π [Garey79]. O facto de existir algum algoritmo de tempo polinomial que resolve algum problema *NP-complete*, implica que todos os problemas da classe *NP* podem ser resolvidos em tempo polinomial. E, ao contrário, caso não exista nenhum algoritmo de tempo polinomial que resolve algum problema *NP*, então todos os problemas *NP-complete* não são resolúveis em tempo polinomial.

A classe *NP-hard* inclui todos os problemas, pertencentes à classe *NP* ou não, nos quais se pode transformar um problema *NP-complete* [Garey79]. Sendo assim, a classe *NP-hard* abrange todos os problemas que são pelo menos tão "difíceis" como os problemas *NP-complete*. Em termos mais formais, um problema Π é *NP-hard* caso exista algum problema $\Pi' \in$ *NP-complete* tal que $\Pi' \propto_T \Pi$,

i.e. existe a redução de Turing de Π' para Π [Kreher99]. Problemas *NP-hard* não podem ser resolvidos em tempo polinomial a não ser que *P=NP*. É de notar que o conceito de *NP-hardness* aplica-se não só a problemas de decisão mas também aos problemas de pesquisa e de optimização.

4.6 Conclusões

Neste capítulo fez-se a análise e classificação de problemas combinatórios bem como dos algoritmos típicos utilizados na sua solução.

A resolução de problemas combinatórios é relacionada com a construção e exame (total ou seleccionada) das soluções possíveis. Os problemas são normalmente formulados como de decisão ou de pesquisa e optimização.

Os problemas combinatórios podem ser especificados sobre modelos matemáticos tais como conjuntos, grafos, matrizes e funções lógicas, que são mutuamente transformáveis uns nos outros. Após a análise de vários modelos foi seleccionado o modelo matricial, pois as matrizes têm uma estrutura que as torna bastante convenientes para armazenamento e processamento em dispositivos digitais. Para além disso, a arquitectura que pretendemos propor para a resolução de problemas combinatórios precisa de ser reconfigurável (para poder abordar problemas e instâncias diferentes). O modelo matricial é mais adequado a este objectivo porque permite facilitar bastante o processo de reconfiguração e ajuda a diminuir o tempo de reconfiguração.

Os problemas de optimização combinatória possuem inúmeras aplicações em áreas tão diversas como desenvolvimento de circuitos digitais, diagnóstico técnico, cartografia, reconhecimento de padrões, planeamento de circulação de transporte, biologia, etc. Tomando em consideração a gama de aplicações, torna-se muito importante o desenvolvimento de algoritmos e de arquitecturas capazes de reduzir o tempo de resolução destes problemas através de uma utilização eficiente dos recursos de dispositivos programáveis.

Todos os problemas considerados no âmbito deste trabalho pertencem às classes *NP-complete* e *NP-hard*. Isto implica que estes problemas não são resolúveis em tempo polinomial (a não ser que *P=NP*, o que não está provado). Obviamente, a implementação com base em hardware reconfigurável não poderá eliminar o efeito do tempo exponencial da solução do problema, mas irá permitir minorá-lo, possibilitando deste modo a solução de problemas com dimensões maiores [Gu97].

5 Processador combinatório reconfigurável

Sumário

É conhecido que a computação reconfigurável atinge bons resultados para as aplicações que exibem muito paralelismo intrínseco e requerem processamento de dados representados em formatos que não são directamente suportados por processadores de uso geral ou DSPs convencionais. Neste capítulo exploramos as capacidades da computação reconfigurável em acelerar a solução de problemas combinatórios. Os problemas combinatórios considerados são os de pesquisa e de optimização e pertencem à classe *NP-hard*.

Devido à sua universalidade e ampla gama de aplicações, os problemas combinatórios têm sido objecto de estudos profundos. Por causa de necessidade do exame de muitas variantes no processo de pesquisa da solução, as implementações em software requerem consumos significativos de tempo. Os modelos matemáticos normalmente empregues são de bastante difícil representação na memória de computadores, portanto deve-se aplicar métodos especiais para a organização, armazenamento e processamento de dados específicos. A implementação dos algoritmos combinatórios em hardware também revela muitas dificuldades. Por um lado cada problema é único em termos do conjunto de operações que requer. O número de operações é normalmente bastante limitado e estas são aplicadas a estruturas uniformes de dados. Por outro lado, é muito difícil propor algum dispositivo universal que possa ser usado para a solução de *qualquer* problema arbitrário. Isto inviabiliza a construção de hardware dedicado porque este seria muito caro e utilizado de uma maneira ineficiente. Contudo o uso de sistemas reconfiguráveis seria justificado e eficaz. Assim, propomos neste capítulo a arquitectura de um coprocessador reconfigurável que pode ser reprogramado para a solução de problemas combinatórios diferentes.

5.1 Introdução

A maioria de sistemas implementados em hardware reconfigurável são baseados em algoritmos relativamente simples. Neste trabalho propomos aplicar a computação reconfigurável a problemas que envolvem uma sequência de operações de controlo muito mais complexa, em particular aos problemas combinatórios. É de notar que existem alguns sistemas reconfiguráveis destinados à solução de tais problemas [Abramovici00, Dandalis99, Graham95, Perkowski02, Platzner98, Plessl01, Zhong98a, Zhong99a]. A maioria destes sistemas baseia-se na ideia do hardware *orientado para a instância*, i.e. para cada instância individual do problema é gerada uma configuração específica da FPGA.

Neste capítulo sugerimos a abordagem *orientada para o domínio* que permite tratar uma série de problemas da área das computações combinatórias. A técnica proposta é baseada no *Processador Combinatório Reconfigurável* (RCP – *Reconfigurable Combinatorial Processor*) que será utilizado para resolver problemas combinatórios formulados sobre matrizes discretas. A abordagem adoptada recorre ao uso de modelos (HT – *Hardware Templates*) que permitem que as mesmas unidades de execução e de controlo sejam utilizadas para uma variedade de problemas e que a sua arquitectura não precise de ser modificada de tarefa a tarefa. Neste caso, é apenas necessário configurar as operações computacionais básicas e os algoritmos de controlo respectivos para cada problema combinatório.

Como demonstrámos no capítulo 4, a solução de problemas combinatórios está frequentemente relacionada com a construção de uma árvore de pesquisa. Uma característica distintiva desta abordagem é que em cada vértice da árvore de pesquisa resolve-se o mesmo problema. Os vários vértices só se distinguem pelos dados de entrada. Isto significa que o problema completo se reduz à execução de um grande número de operações repetidas sobre um conjunto de dados que é modificado periodicamente.

De entre os modelos matemáticos formais considerados no capítulo 4 e utilizados para a especificação de tarefas combinatórias escolhemos as matrizes discretas porque estas são de uma representação bastante fácil em software e em hardware. Para além disso a arquitectura do processador combinatório que pretendemos propor, precisa de ser reconfigurável (para poder abordar uma variedade de problemas e das suas instâncias). O modelo matricial serve bem para este fim pois simplifica essencialmente o processo de reconfiguração e ajuda a minimizar o tempo da mesma. É conhecido que muitos problemas combinatórios que surgem no desenvolvimento de sistemas digitais e em inteligência artificial podem ser formulados sobre matrizes lógicas (booleanas, ternárias e outras) [Zakrevski98]. Todas estas matrizes são discretas dado que o número de valores dos seus elementos é limitado. Na prática, normalmente são utilizados três valores que são *0*, *1* e *'-'* (*don't care*).

Como notámos no capítulo 3, é difícil implementar eficientemente um algoritmo combinatório só em FPGA. A razão disso é que a lógica reconfigurável não se adequa a algumas computações. Portanto, os melhores resultados podem ser atingidos ao combinar os recursos de um processador de uso geral e uma FPGA, conforme representado na fig. 5.1 (por exemplo, porções de algoritmos activadas raramente podem ser atribuídas ao software enquanto as computações regulares sobre

grandes quantidades de dados podem ser executadas em hardware reconfigurável). Para além de pouca eficiência de FPGAs na realização de certas porções de algoritmos combinatórios, existe uma outra razão que justifica a partição da aplicação entre software (implementado num processador de uso geral) e hardware reconfigurável. Esta tem a ver com os recursos de hardware reconfigurável que estão sempre limitados o que impede a solução de *qualquer* instância dum problema combinatório. Este assunto será analisado em mais detalhe na secção 5.7.

Os problemas combinatórios são muito diversificados dificultando a construção de um acelerador *universal* que assegurasse a sua solução eficiente. Obviamente, seria muito problemático arranjar um compromisso entre a complexidade deste dispositivo e a redundância dos seus componentes [Skliarova00b]. Dada a heterogeneidade de problemas combinatórios, o desenvolvimento dum dispositivo universal seria muito sofisticado e dispendioso e a sua funcionalidade não ia ser aproveitada a fundo. Portanto a arquitectura do RCP deve ser *reconfigurável* para permitir a solução de um conjunto de problemas combinatórios e, no mesmo tempo, não ser demasiado complexa. A característica de reconfiguração é de grande importância porque o número de operações primárias diferentes necessárias para suportar todos os algoritmos relevantes é muito elevado, mas qualquer algoritmo particular só requer um número muito limitado destas operações [Perkowski02, Skliarova00b]. Uma plataforma excelente para os dispositivos deste tipo é baseada em circuitos reconfiguráveis cuja arquitectura pode ser personalizada para um problema específico através da reconfiguração. A complexidade de FPGAs recentes permite implementar um coprocessador combinatório completo com a arquitectura desejada.

Conforme descrito no capítulo 3, muitas das FPGAs disponíveis comercialmente são dispositivos de contexto único que não suportam a reconfiguração dinâmica parcial. Contudo, os *hardware templates* possibilitam a implementação da reconfiguração dinâmica parcial dos dispositivos reconfiguráveis estaticamente. A ideia principal dum HT é construir uma unidade computacional parametrizável que inclua componentes modificáveis e fixos com as ligações fixas entre estes (ver fig. 5.1) [Sklyarov84b, Sklyarov03]. A personalização da unidade faz-se configurando os componentes com a funcionalidade modificável sendo o número destes componentes minimizado. Para possibilitar a configuração selectiva (i.e. parcial), os componentes modificáveis podem ser baseados em blocos de memória que sempre que reprogramados, asseguram uma funcionalidade diferente.

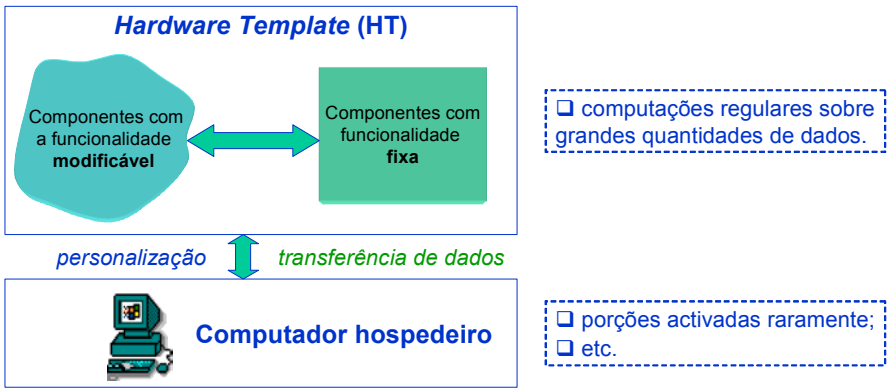


Fig. 5.1. Modelo de computação adaptado: a aplicação é partilhada entre software e hardware. A parte de hardware é baseada num HT que é personalizado para cada problema recorrendo à reconfiguração parcial.

5.2 Requisitos

Com base na análise de modelos matemáticos utilizados na optimização combinatória [Skliarova01a], das operações primárias e das estruturas de algoritmos relevantes [Skliarova00b], foi possível formular os requisitos seguintes ao processador combinatório reconfigurável.

Por causa da heterogeneidade de problemas combinatórios, o processador deve ser reconfigurável dinamicamente, i.e. há-de fornecer a possibilidade de modificar a sua funcionalidade em *run-time*. Para reduzir o tempo de reconfiguração, o RCP tem de ser baseado em HT. Neste caso serão apenas modificadas as operações computacionais primárias e os algoritmos de controlo correspondentes. Todos os componentes restantes bem como as ligações entre estes não serão alterados. A fim de realizar esta ideia propomos a utilização de HTs baseados em RAM de tal modo que a personalização da funcionalidade do RCP é atingida através da reprogramação dos blocos de RAM relevantes.

Consequentemente, o RCP deve ser construído com base em FPGA que inclui células de memória distribuídas (LUTs) ou blocos de memória embutidos. As séries XC4000 e Virtex da Xilinx [Xilinx] são bons representantes destes tipos de FPGAs. As células de memória distribuídas podem ser agrupadas em blocos com as dimensões necessárias a fim de guardar as matrizes de modo semelhante ao empregue pelos registos de uso geral em computadores convencionais para guardar os operandos e os resultados intermédios das computações. Os blocos de memória embutidos também servem para a implementação de componentes modificáveis do HT. O processo de reconfiguração é facilitado por circuitos auxiliares que controlam o carregamento dos blocos de RAM.

5.3 Arquitectura

A arquitectura do RCP que satisfaz os requisitos especificados e pode ser usada para a solução de problemas combinatórios formulados sobre matrizes booleanas e ternárias está representada na fig. 5.2. Os componentes cinzentos da fig. 5.2 são personalizáveis para as necessidades de uma aplicação particular. Os componentes restantes possuem funcionalidade fixa e realizam os procedimentos comuns que são partilhados por muitos problemas combinatórios.

Os componentes primários para a execução de operações são a unidade de controlo reconfigurável (RCU – *Reconfigurable Control Unit*) e a unidade funcional reconfigurável (RFU – *Reconfigurable Functional Unit*). Ambos os componentes são construídos de blocos reprogramáveis tais como células de FPGA baseadas em RAM. Sendo assim a modificação do conteúdo do bloco de RAM respectivo permite alterar a funcionalidade de qualquer componente.

Os registos servem para guardar os resultados intermédios das operações. Os circuitos combinatórios permitem acelerar significativamente as operações frequentes, tais como o teste se uma linha (ou coluna) da matriz contém um valor específico. Estes circuitos geram condições lógicas que são enviadas para a unidade de controlo a fim de especificar a selecção do ramo desejado do algoritmo de controlo. Dependendo dos valores das condições lógicas, a unidade de

controlo produz a sequência de sinais de controlo utilizados pela unidade funcional para executar as operações necessárias.

Consideremos cada bloco reconfigurável da fig. 5.2 em mais detalhe.

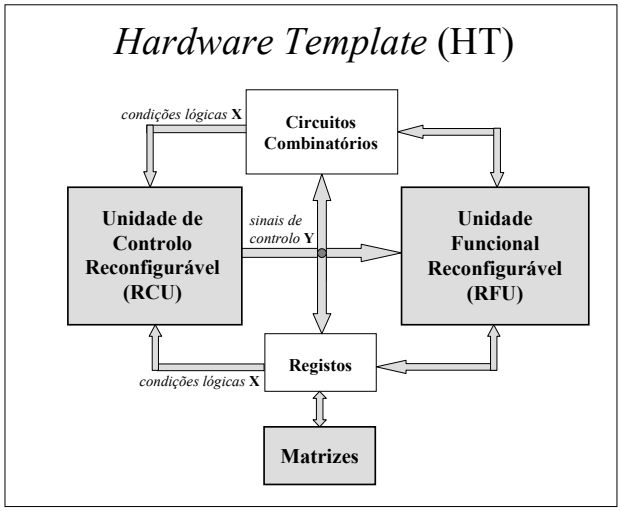


Fig. 5.2. Arquitectura do processador combinatório reconfigurável.

5.3.1 Matrizes reprogramáveis

O bloco *Matrizes* pode guardar até três matrizes lógicas de dimensões $m \times n$. Para cada matriz lógica, **U**, são mantidas duas cópias físicas, a primeira das quais guarda a matriz original, **U**, e a segunda guarda a matriz transposta, $\mathbf{U}^{\mathrm{T}}$. Obviamente, esta abordagem duplica os recursos necessários para armazenar os dados da matriz. Contudo, deste modo o desempenho do RCP é melhorado significativamente porque cada linha e coluna são acessíveis num só ciclo de relógio. Cada matriz física é composta por dois blocos de dimensões iguais: *U_ones* e *U_zeros*. Os blocos *U_ones* (U^T*_ones*) contêm *1*s em posições onde a matriz lógica original **U** ($\mathbf{U}^{\mathrm{T}}$) tem *1*s, e contêm *0*s em todas as posições restantes. De modo semelhante, os blocos *U_zeros* (U^T*_zeros*) guardam *1*s apenas em posições em que a matriz **U** ($\mathbf{U}^{\mathrm{T}}$) tem *0*s. Como resultado, cada elemento u_{ij}, i=1,…,m, j=1,…,n, da matriz **U** é codificado da seguinte maneira:

- se u_{ij}=*1*, então *U_ones*[*i*][*j*] = *1*, *U_zeros*[*i*][*j*] = *0*, U^T*_ones*[*j*][*i*] = *1*, e U^T*_zeros*[*j*][*i*] = *0*.
- se u_{ij}=*0*, então *U_ones*[*i*][*j*] = *0*, *U_zeros*[*i*][*j*] = *1*, U^T*_ones*[*j*][*i*] = *0*, e U^T*_zeros*[*j*][*i*] = *1*.
- se u_{ij}='-', então *U_ones*[*i*][*j*] = *0*, *U_zeros*[*i*][*j*] = *0*, U^T*_ones*[*j*][*i*] = *0*, e U^T*_zeros*[*j*][*i*] = *0*.

A fig. 5.3. ilustra a codificação da matriz **U** seguinte:

$$\mathbf{U} = \begin{bmatrix} - & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & - & \mathbf{1} \end{bmatrix}$$

Antes da execução os dados de matrizes relevantes são transferidos para o bloco *Matrizes*. Se for necessário resolver algum outro problema (ou alguma outra instância do mesmo problema), o bloco pode ser facilmente reprogramado com os dados diferentes.

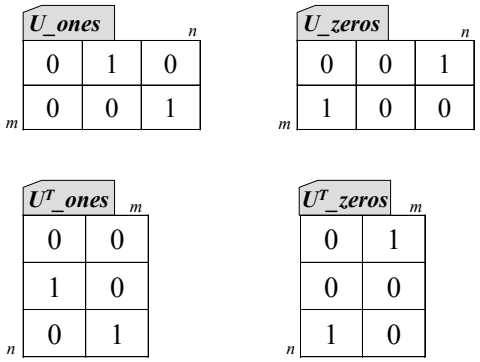


Fig. 5.3. Exemplo de representação de uma matriz ternária em blocos de memória da FPGA.

5.3.2 Unidade de controlo reconfigurável

A unidade de controlo (RCU) implementa os algoritmos de controlo necessários. A unidade pode ser modelada por uma máquina de estados finitos (FSM) com o comportamento modificável dinamicamente que gera uma sequência de operações para o algoritmo combinatório que está a ser executado. Uma FSM pode ser representada ao nível estrutural como uma composição do circuito combinatório que calcula os estados seguintes e as saídas e dum registo que mantém o estado corrente, conforme representado na fig. 5.4.

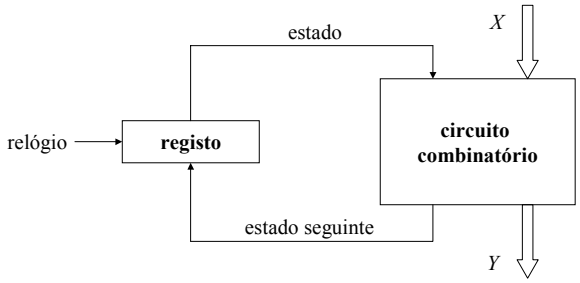


Fig. 5.4. Uma FSM pode ser representada ao nível estrutural como uma composição do circuito combinatório e um registo.

Uma FSM pode ser especificada por um sêxtuplo $(S, X, Y, \varphi, \psi, s_1)$, onde $S=\{s_1,...,s_M\}$ é o conjunto finito de estados, $X=\{x_1,...,x_L\}$ é o conjunto de entradas, $Y=\{y_1,...,y_N\}$ é o conjunto de saídas, φ é a função de transição, ψ é a função de saída e $s_1 \in S$ é o estado inicial. Os valores L e N, que correspondem respectivamente ao número de entradas e de saídas da FSM, são fixados estaticamente. Os parâmetros restantes tais como S, φ e ψ podem ser modificados dinamicamente conforme representado na fig. 5.5. Isto permite alterar o comportamento da RCU e, em consequência, implementar conjuntos diferentes de operações com base num único circuito. As linhas de entrada $x_1,...,x_L$ e as de saída $y_1,...,y_N$ possuem ligações predeterminadas com outros componentes do processador combinatório. Contudo, a análise de entradas e a geração de saídas podem ser feitas de maneiras diferentes através da modificação dos parâmetros S, φ e ψ.

A fim de suportar a reconfiguração a FSM deve ser baseada em RAM, i.e. o circuito combinatório é construído dos blocos de RAM (ver fig. 5.6a). Cada código de estado lido do registo da FSM é combinado com as variáveis de entrada do conjunto $X=\{x_1,...,x_L\}$ formando desta maneira o

endereço da *RAM da FSM*. No endereço obtido activa-se uma palavra da *RAM da FSM* que especifica o estado seguinte. Uma RAM adicional (*Y RAM* na fig. 5.6a) serve para produzir os sinais de saída com base nos códigos dos estados (consideramos o modelo de Moore da FSM). Sendo assim é muito fácil reprogramar o circuito construído. Para tal é suficiente recarregar o conteúdo de dois blocos de RAM.

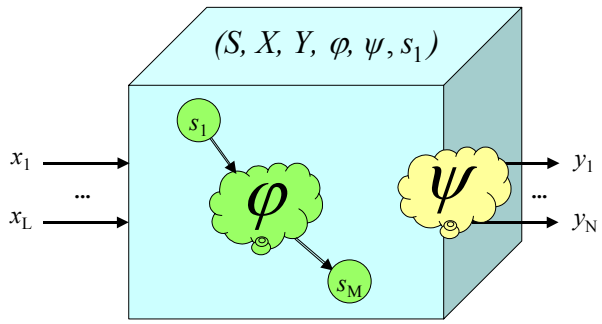


Fig. 5.5. FSM modificável dinamicamente.

Suponhamos que K é o número mínimo de bits necessários para codificar os estados da FSM. Então o tamanho da *Y RAM* é $2^K \times N$ e o da *RAM da FSM* é $2^{(K+L)} \times K$. Contudo, caso o número de condições lógicas L seja suficientemente grande, esta implementação requereria muitos recursos (i.e. a *RAM da FSM* ocupava uma área da FPGA demasiado extensa). Portanto, a desvantagem principal desta abordagem é a de que só possibilita a implementação de algoritmos de controlo muito simples. Se considerar uma tecnologia particular é relativamente fácil calcular o número de CLBs da FPGA que precisam de ser usados (se construirmos a RCU com base em LUTs). Por exemplo, os CLBs das FPGAs da família XC4000 podem ser configurados para implementar um par de RAMs de 16×1, uma RAM de 32×1, ou uma RAM *dual-port* de 16×1 [Xilinx00]. Usando esta informação, pode-se estimar o número de CLBs necessários para FSMs baseadas em RAM com a estrutura da fig. 5.6a.

Para reduzir significativamente o tamanho dos blocos de RAM da fig. 5.6a aplicámos a técnica especial de codificação de estados que permite fazer com que a dependência funcional dos códigos de estados das condições lógicas seja diminuída [Sklyarov00]. Para tal introduzimos um bloco novo P que é usado para combinar os sinais de entrada do conjunto X com os códigos de estados (ver fig. 5.6b) e permite reduzir essencialmente a largura do barramento de endereços para a *RAM da FSM*. Que K' denote a largura do barramento obtida. Neste caso a expressão seguinte é válida: $K \le K' << (K+L)$ e para muitas aplicações práticas $K' \rightarrow K$. Como resultado, o tamanho da *RAM da FSM* torna-se igual a $2^{K'} \times K'$.

Para garantir a reprogramabilidade da FSM o bloco P da fig. 5.6b deve ser decomposto conforme representado na fig. 5.6c. O bloco *A RAM* implementa as transições condicionais, i.e. as transições que são influenciadas por variáveis de entrada do conjunto X. Para todas as transições incondicionais, o vector de saída **a** é igual a *0*. O bloco *P RAM* da fig. 5.6c calcula o vector de saída **p** que depende das entradas do conjunto X e do vector **a**. O recarregamento de todas as RAMs (i.e. da *A RAM*, *Y RAM*, *P RAM* e da *RAM da FSM*) permite implementar algoritmos de controlo diferentes com base no mesmo hardware.

Calculemos o tamanho da memória necessária. Que $S^x \subset S$ seja o conjunto de estados da FSM dos quais existem transições condicionais e que $G = \lceil \log_2(|S^x| + 1) \rceil$. Então o tamanho da *Y RAM* é $2^{K'} \times N$, o da *RAM da FSM* é $2^{K'} \times K'$, o da *A RAM* é $2^{K'} \times G$ e o da *P RAM* é $2^{G+L} \times K'$.

A eficiência da técnica proposta foi avaliada com a ajuda de experiências diferentes que mostraram que para a maioria de aplicações práticas $K \leq K' \leq (K+1)$. A redução da área ocupada pela RCU conseguida com a estrutura proposta (ver fig. 5.6c), comparando-a com a RCU da fig. 5.6a, depende dos parâmetros K, K', N, L e $|S^x|$. Supondo que os valores destes parâmetros foram restringidos da maneira seguinte: K=7, N=32, L=5, $|S^x|$=15, o tamanho total dos blocos de memória da fig. 5.6a será de 32.768 bits ($2^{7+5} \times 7 + 2^7 \times 32$), enquanto para a estrutura da RCU ilustrada na fig. 5.6c o tamanho dos blocos de memória seria ou de $2^8 \times 32 + 2^8 \times 4 + 2^{4+5} \times 8 + 2^8 \times 8$ = 15.360 bits (se $K'=K+1$), ou de $2^7 \times 32 + 2^7 \times 4 + 2^{4+5} \times 7 + 2^7 \times 7$ = 9.088 bits (se $K'=K$). Como resultado, para este caso específico, os recursos de memória iam ser deduzidos por um factor F, sendo $2 \leq F \leq 3.5$.

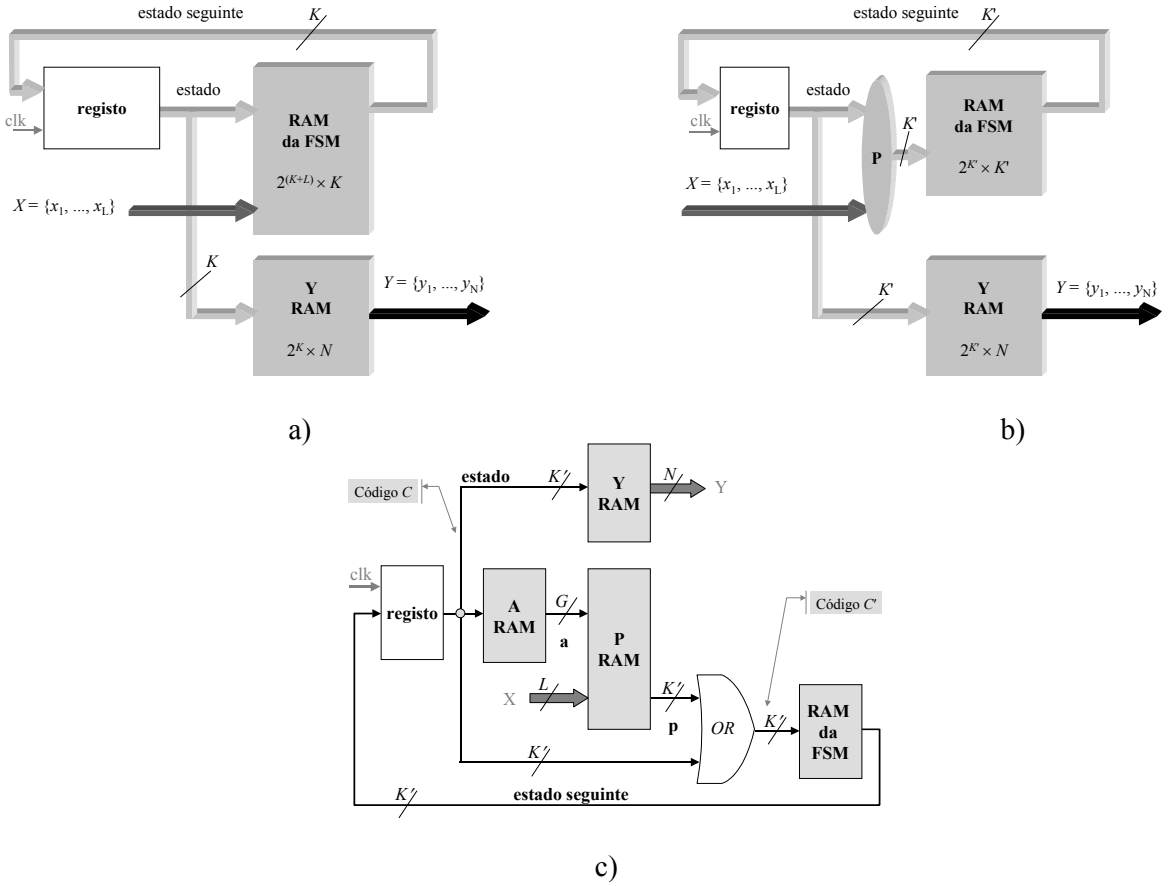


Fig. 5.6. Estrutura trivial da FSM baseada em RAM (a); Combinação dos sinais de entrada com os códigos dos estados (b); Estrutura final da RCU (c).

5.3.2.1 Síntese da RCU

A síntese da RCU é composta por uma sequência de passos que permite gerar o conteúdo dos blocos de RAM da fig. 5.6c a partir duma especificação comportamental. É importante que os resultados da síntese estejam representados em forma dum conjunto de padrões de zeros e uns que especificam o conteúdo de cada bloco de memória e que devem ser carregados no HT. Isto

significa que não é permitido modificar a estrutura do circuito da fig. 5.6c bem como todas as ligações entre os seus componentes. Como resultado, não é necessário repetir o processo de projecto do circuito que poderia levar a situações imprevisíveis durante os passos de mapeamento, colocação e encaminhamento. Para além disso, dado que o circuito é previamente testado, podemos garantir que este funciona correctamente só sendo necessários procedimentos muito simplificados de verificação.

Suponhamos que o comportamento da RCU é especificado em forma de esquemas de grafos (GSs - *Graph-Schemes*) [Baranov94]. Inicialmente um dado GS transforma-se numa tabela estrutural [Sklyarov00]. De seguida os passos seguintes devem ser executados:

1) Aplicação do método de codificação.
2) Substituição das variáveis de entrada e a sua combinação com os códigos dos estados.
3) Síntese do conteúdo para cada bloco de RAM da fig. 5.6c.

Como já foi mostrado, a área ocupada pela RCU depende fortemente da largura do barramento de endereços da *RAM da FSM*. Esta largura pode ser diminuída essencialmente através da utilização da codificação especial de estados. No algoritmo proposto tenta-se minimizar o número de bits do endereço de tal maneira que $K' \rightarrow \lceil \log_2 M \rceil$, onde M é o número de estados da FSM.

Denotemos por $T=\{t_1,...,t_R\}$ o conjunto de todas as transições de estados. Cada transição pode ser representada como $t_i=\{s_{de}, s_{para}, X(s_{de}, s_{para})\}$, $i=1,...,R$, onde $s_{de}, s_{para} \in S$, s_{de} é o estado de origem na transição t_i e s_{para} é o estado de destino, $X(s_{de}, s_{para})$ é a função de variáveis de entrada que causa a transição do s_{de} para s_{para}. Quando existem duas transições $t_i=\{s_{de\_i}, s_{para\_i}, X(s_{de\_i}, s_{para\_i})\}$ e $t_j=\{s_{de\_j}, s_{para\_j}, X(s_{de\_j}, s_{para\_j})\}$, e se $s_{de\_i}=s_{de\_j}$ então t_i e t_j têm o *estado de origem comum* e se $s_{para\_i}=s_{para\_j}$ então t_i e t_j têm o *estado de destino comum* [Hachtel96]. Estes conceitos estão ilustrados na fig. 5.7

$$T = \{t_1,\ldots,t_R\}, \qquad t_i = \{s_{de}, s_{para}, X(s_{de}, s_{para})\}, \qquad i=1,...,R$$

$$t_i=\{s_{de\_i}, s_{para\_i}, X(s_{de\_i}, s_{para\_i})\}$$
$$t_j=\{s_{de\_j}, s_{para\_j}, X(s_{de\_j}, s_{para\_j})\} \quad \Rightarrow \quad t_2 = \{s_2, s_3, x_2\}$$

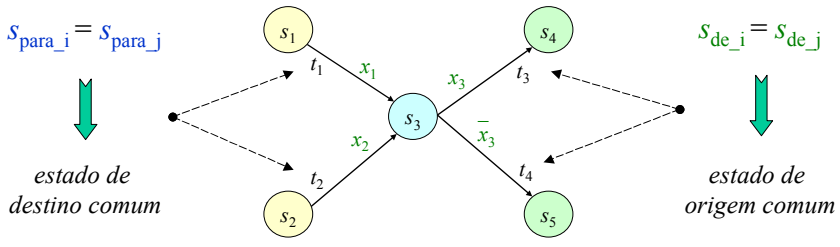


Fig. 5.7. Notação e terminologia utilizadas na codificação de estados.

Dividamos o conjunto T em M subconjuntos: $T=\{TFI_1,...,TFI_M\}$, $TFI_1 \cap \ldots \cap TFI_M = \varnothing$ e $TFI_1 \cup \ldots \cup TFI_M = T$, de tal maneira que cada subconjunto TFI_i, $i=1,...,M$, é composto por transições com o estado de origem comum. Sendo assim todas as transições do estado s_1 formam o conjunto

TFI_1, todas as transições do estado s_2 formam o conjunto TFI_2, etc. Agora consideremos o conjunto T como uma união dos subconjuntos seguintes: $T=\{TFIO_1,...,TFIO_Q\}$, $1\le Q\le M$, $TFIO_1\cap\ldots\cap TFIO_Q=\varnothing$ e $TFIO_1\cup\ldots\cup TFIO_Q=T$, sendo cada elemento $TFIO_i$, $i=1,\ldots,Q$, composto de tais subconjuntos TFI que incluam as transições com o estado de destino comum. Obviamente se o algoritmo de controlo não incluir transições condicionais, então $TFIO_i=t_i$, $i=1,...,R$.

Consideremos um exemplo. Suponhamos que o algoritmo de controlo está descrito por GS representado na fig. 5.8a. Este pode ser formalmente transformado numa tabela estrutural [Baranov86]. No nosso caso $M=9$, $L=2$, $N=8$, $R=14$, $Q=7$ e $T=\{t_1,..,t_{14}\}=\{TFI_1,...,TFI_9\}$, $TFI_1=\{s_1, s_2, 1\}$, $TFI_2=\{\{s_2, s_3, \bar{x}_1\}, \{s_2, s_8, x_1 x_2\}, \{s_2, s_9, x_1 \bar{x}_2\}\}$, etc. Todos os conjuntos TFI_i, $i=1,...,M$, são mostrados na fig. 5.8b. A fig. 5.8c representa os conjuntos $T=\{TFIO_1,..., TFIO_7\}$ (marcados com as letras *a*-*e*).

De acordo com o método [Sklyarov00] a cada transição t_i, $i=1,...,R$, deve ser atribuído um código $C'(t_i)$, e a cada conjunto TFI_i, $i=1,...,M$, deve ser atribuído o código $C(TFI_i)$ (é de notar que $C(TFI_i)=C(s_i)$), com as restrições seguintes:

- $C(TFI_1)$ *ort* … *ort* $C(TFI_M)$, i.e. os códigos de todos os estados da FSM devem ser mutuamente ortogonais. Os dois códigos consideram-se ortogonais se pelo menos numa das suas componentes um código tiver o valor *0* e outro o valor *1*.
- $C'(t_1)$ *ort* … *ort* C' (t_R), i.e. os códigos de todas as transições devem ser mutuamente ortogonais excepto se as transições respectivas possuírem o estado de destino comum.
- $C(TFI_i)=C'(t_j)\wedge\ldots\wedge C'(t_{|TFI_i|})$, $t_j,...,t_{|TFI_i|}\in TFI_i$, $i=1,...,M$.

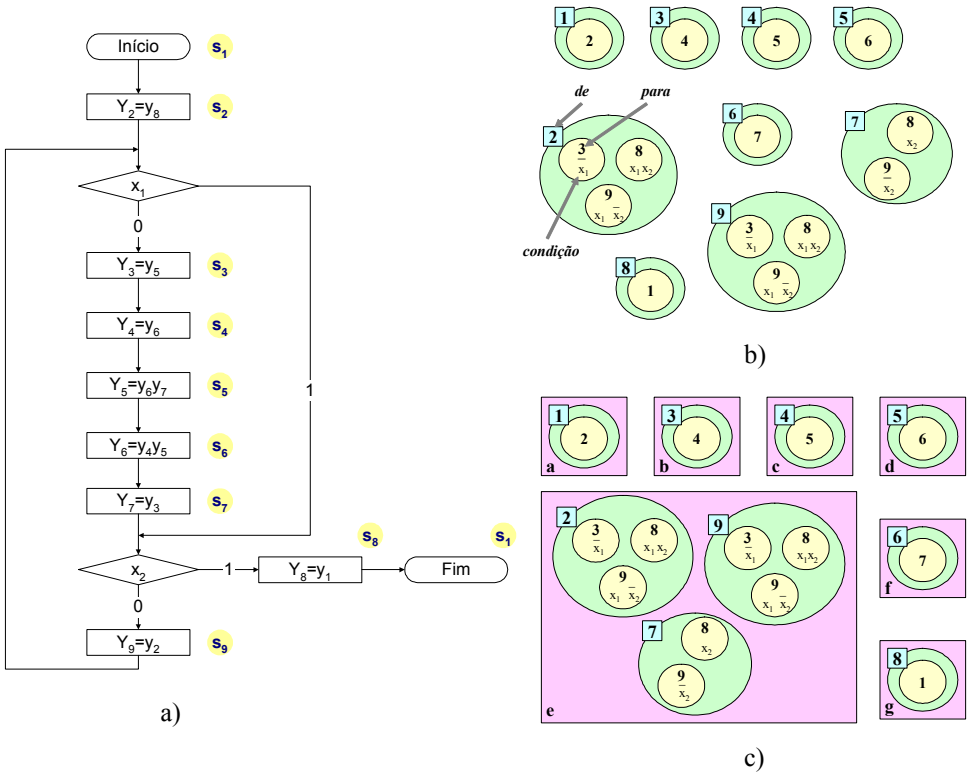


Fig. 5.8. GS que descreve o algoritmo de controlo (a); Os conjuntos TFI_i, $i=1,...,9$ (b); Os conjuntos $TFIO_i$, $i=1,...,7$ (c).

Quando atribuímos os códigos é conveniente fazer com que os códigos das transições pertencentes ao mesmo conjunto TFI_i tenham a distância de Hamming B mínima. Quanto menor for B, mais transições é possível codificar com uma dada quantidade de bits. O algoritmo final de codificação é composto pela sequência de passos seguintes:

1) Calcular o valor $K' = \lceil \log_2 |M| \rceil$.
2) Seleccionar dos conjuntos $TFIO_i$, i=1,…,Q, um conjunto $TFIO_{max}$ com o número máximo de elementos: $|TFIO_{max}| \geq |TFIO_i|$, $i \neq max$, i,max=1,...,Q.
3) Seleccionar dos conjuntos TFI_i, i=1,…,M, um conjunto TFI_{max} com o número máximo de elementos: $|TFI_{max}| \geq |TF_i|$, $i \neq max$, i,max=1,...,M, $TFI_{max} \subseteq TFIO_{max}$, $TFI_i \subseteq TFIO_{max}$.
4) Calcular o número mínimo de bits B precisos para distinguir os códigos das transições pertencentes ao TFI_{max}: $B = \lceil \log_2 |TFI_{max}| \rceil$.
5) Seleccionar no conjunto TFI_{max} a transição t_{max}, max=1,...,R, $t_{max} \in TFI_{max}$, que tenha o estado de destino comum com o número máximo dos elementos do $TFIO_{max}$.
6) Verificar se é possível atribuir à transição seleccionada t_{max} o código de uma transição já processada t_{exist}, $max,exist$=1,...,R. Isto só é possível quando são satisfeitas as duas condições seguintes:
   - As transições t_{max} e t_{exist} possuem o estado de destino comum;
   - $C(TFI_{max})$ *ort* $C(TFI_i)$, i=1,...,M, $i \neq max$, i.e. os códigos de todos os conjuntos TFI_i, i=1,...,M, $i \neq max$, (i.e. os códigos dos estados da FSM) continuam a ser mutuamente ortogonais.
7) Caso não seja possível reutilizar um código existente, selecciona-se o primeiro código atribuído no conjunto TFI_{max} (para a primeira transição a considerar usa-se sempre o código "*1...1*"). A seguir o código decrementa-se e verifica-se se este pode ser utilizado, i.e. se a distância de Hamming entre os códigos do conjunto TFI_{max} é menor ou igual a B. Caso esta condição seja satisfeita, verifica-se se os códigos de todos os conjuntos TFI_i, i=1,...,M, são mutuamente ortogonais, i.e. $C(TFI_{max})$ *ort* $C(TFI_i)$, i=1,...,M, $i \neq max$. Caso isto seja verdade o código obtido é atribuído à transição t_{max}. Em caso oposto, o código decrementa-se mais uma vez. O processo continua até que obtenhamos um código igual a "*0...0*". Se neste ponto a solução ainda não é encontrada, tentamos incrementar o código inicial até que este se torne igual a "*1...1*". Caso novamente seja impossível encontrar a solução, incrementamos o valor B e repetimos o ponto 7.
8) Caso seja impossível codificar todos os estados com K' bits, este valor é incrementado e todos os passos começando do ponto 2 são repetidos.

A aplicação do algoritmo proposto ao GS da fig. 5.8a permite encontrar a solução apresentada na tabela 5.1. A coluna s_{de} contém todos os estados, as colunas $C(s_{de})$ e $C(s_{para})$ guardam os códigos dos estados respectivos, a coluna $X(s_{de}, s_{para})$ representa as funções das variáveis de entrada que causam a transição do s_{de} para s_{para}, a coluna Y indica as microinstruções que devem ser geradas em cada estado. A coluna *FSM RAM* mostra os endereços da *RAM da FSM* onde os códigos respectivos dos estados de destino $C(s_{para})$ serão armazenados. Em cada código $C(s_{de})$ os símbolos i

apontam para aqueles bits que distinguem os endereços dos estados com o estado de origem comum. A coluna $X \rightarrow p$ contem os valores do vector **p** para cada transição. Por exemplo, na transição t do s_2 para s_9, temos $C(s_{de})=C(s_2)=1\mathbf{i}1\mathbf{i}$ e $C'(t)=1110$, portanto **p**=0**1**0**0**.

Tabela 5.1. Tabela estrutural com os resultados da síntese.

s_{de}	$C(s_{de}) = C(TFI_{de})$	$X(s_{de}, s_{para})$	Y	$X \rightarrow$ **p**	s_{para}	$C(s_{para})$	**FSM RAM =** $C'(t)$
s_1	0000	1	-	0000	s_2	1010	0000
s_2	1i1i = 1010	$\bar{x}_1$ x_1x_2 $x_1\bar{x}_2$	y_8	0000 0101 0100	s_3 s_8 s_9	0111 1101 1100	1010 1111 1110
s_3	0111	1	y_5	0000	s_4	1000	0111
s_4	1000	1	y_6	0000	s_5	1001	1000
s_5	1001	1	y_6y_7	0000	s_6	1011	1001
s_6	1011	1	y_4y_5	0000	s_7	1110	1011
s_7	111i = 1110	x_2 $\bar{x}_2$	y_3	0001 0000	s_8 s_9	1101 1100	1111 1110
s_8	1101	1	y_1	0000	s_1	0000	1101
s_9	11ii = 1100	$\bar{x}_1$ x_1x_2 $x_1\bar{x}_2$	y_2	0000 0011 0010	s_3 s_8 s_9	0111 1101 1100	1100 1111 1110

O algoritmo descrito foi implementado numa ferramenta de software desenvolvida em C++. A ferramenta recebe o nome de um ficheiro de texto que especifica o algoritmo de controlo da maneira seguinte:

```
Número do estado de origem <condição para a transição respectiva> Número do
estado de destino <valores de saídas>
```

Um fragmento do algoritmo de controlo da fig. 5.8a está ilustrado na parte inferior esquerda da fig. 5.9. A ferramenta desenvolvida efectua todos os cálculos necessários e gera um ficheiro que serve para a configuração da RCU (ver fig. 5.9). Para programar a *RAM da FSM* e a *Y RAM* (ver fig. 5.6c) são utilizados os códigos respectivos das colunas *FSM RAM* e $C(s_{de})$ da tabela 5.1. Para programar a *A RAM* todos os estados dos quais existem transições condicionais (i.e. os estados s_2, s_7 e s_9 da fig. 5.8a são codificados com códigos binários sequenciais de tamanho mínimo. Todos os estados com transições incondicionais recebem o código "*0…0*". Para programar a *P RAM* usa-se o conteúdo da coluna $X \rightarrow p$, i.e. no endereço que combina X e **a**, escreve-se o código da linha respectiva da coluna $X \rightarrow p$.

A fig. 5.10 ilustra como programar todos os blocos de RAM existentes na RCU para o exemplo considerado. As linhas tracejadas indicam que as posições respectivas de memória podem conter qualquer valor dado que isto não influencia a execução do algoritmo. Na fig. 5.10 o estado activo é o s_2 com código *1010*. No bloco *A RAM* a este estado corresponde o vector **a**=[01] que em combinação com a condição lógica **x**=[01] activa no bloco *P RAM* o vector **p**=[0100]. De seguida, a operação ($\mathbf{p} \vee C(s_2)$) é executada que forma o endereço do bloco *RAM da FSM* no qual se

encontra o código do estado seguinte - *1100*. Na tabela 5.1 este código corresponde ao estado s_8, o que permite concluir que a RCU está a funcionar correctamente.

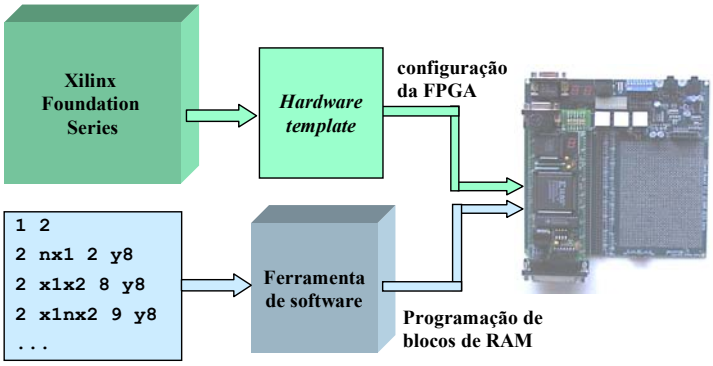


Fig. 5.9. Utilização da ferramenta desenvolvida para programar os blocos de RAM da RCU.

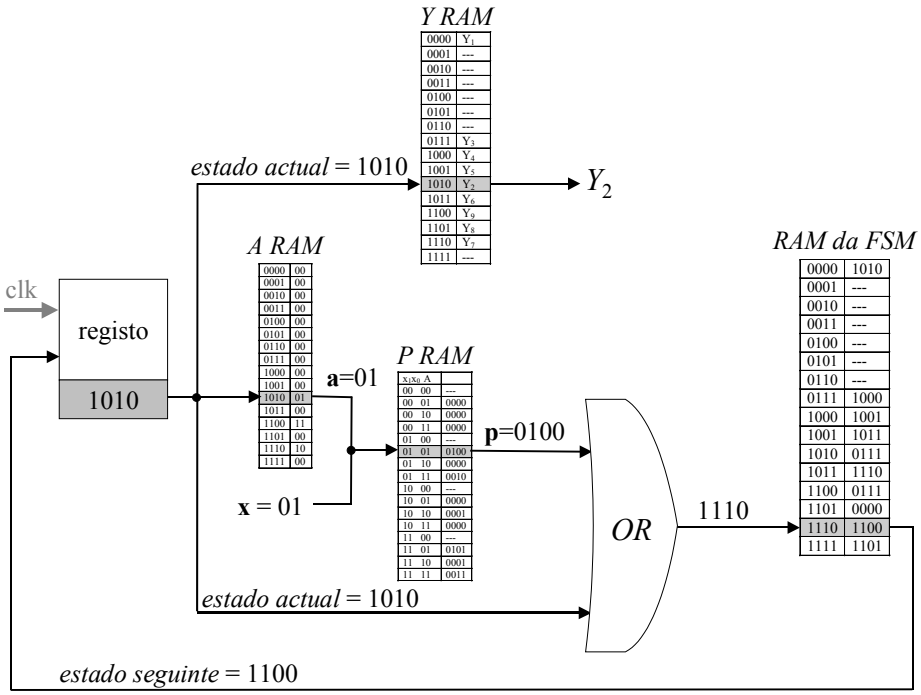


Fig. 5.10. Programação dos blocos de RAM da RCU para o algoritmo de controlo da fig. 5.8a.

As características do algoritmo proposto foram estimadas numa série de experiências executadas num PIII/800MHz/256MB com o sistema operativo Windows2000. A eficiência do algoritmo depende não só do número de estados e das condições lógicas mas também da complexidade de transições presentes no GS de entrada. Os resultados representados na tabela 5.2 mostram que a técnica descrita pode ser utilizada para aplicações práticas.

A abordagem proposta foi também comparada com as técnicas utilizadas no *Xilinx Foundation Software* (*Series 3.1i*). Para todos os exemplos considerados o método permite reduzir o número de CLBs necessários para a RCU. É muito importante que a estrutura da RCU proposta permite a implementação de *qualquer* algoritmo de controlo que satisfaça as restrições impostas (i.e. o número máximo de estados, condições lógicas, etc.).

Tabela 5.2. Resultados das experiências com o algoritmo de síntese da RCU.

Exemplo	***M* – número de estados da FSM**	***L* – número de condições lógicas**	***N* – número de sinais de controlo**	***K* – número mínimo de bits necessários para os códigos de estados**	***K'* - número de bits actualmente utilizados para os códigos de estados**	**Tempo de síntese (*s*)**
min_row	9	2	5	4	4	0.058
ex_t_m2	13	10	32	4	5	0.24
min_cover	29	5	24	5	5	0.15
gr_al	123	20	67	7	8	10.16

5.3.3 Unidade funcional reconfigurável

Cada matriz discreta pode ser considerada como um conjunto de vectores discretos. Neste capítulo utilizamos vectores cujos elementos podem receber um dos valores seguintes: *0*, *1* e *'-'* (*don't care*). A unidade funcional reconfigurável (RFU) realiza operações diferentes especificadas pela RCU sobre os vectores discretos. É de notar que existe um número de operações praticamente infinito que se pode executar sobre vectores. Contudo, cada problema real só requer um número muito limitado de operações. De facto a maioria de algoritmos combinatórios é baseada na repetição múltipla de operações semelhantes que devem ser aplicadas a linhas e colunas de matrizes quer sequencialmente quer em paralelo [Zakrevski81]. Sendo assim, não é razoável construir um bloco lógico complexo para as computações respectivas. O que propomos é assegurar a funcionalidade adequada através das modificações dinâmicas. Uma vez que para cada problema prático o número de operações repetidas é bastante elevado, o tempo de reconfiguração torna-se ínfimo em comparação com o tempo de execução.

A estrutura da RFU proposta está representada na fig. 5.11 [Skliarova00a]. Esta inclui registos para guardar os resultados intermédios, circuitos combinatórios para realização de computações comuns e os circuitos reconfiguráveis dinamicamente. A RFU gera as condições lógicas que são enviadas à RCU para determinar a sequência apropriada do algoritmo de controlo. Com base nestes valores a RCU gera os sinais de controlo que especificam operações que é necessário executar sobre as matrizes.

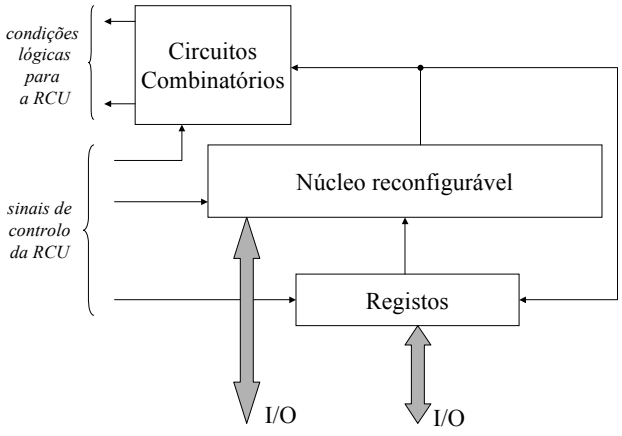


Fig. 5.11. Estrutura da RFU.

A fim de suportar a característica de reconfiguração, propomos construir a RFU com base em blocos de RAM. A análise das diferentes operações que são normalmente empregues pelos algoritmos combinatórios, permitiu dividi-las em três grupos que consideramos a seguir.

O primeiro grupo inclui as operações unárias e binárias que abrangem as operações lógicas tradicionais, tais como *NOT*, *OR*, *AND*, *XOR*, *NAND*, etc., e que são aplicadas a um ou dois vectores, respectivamente. O resultado deste tipo de operações é um novo vector. Neste caso são dados dois vectores $\mathbf{a} = [a_1,...,a_n]$ e $\mathbf{b} = [b_1,...,b_n]$ e pede-se para calcular um novo vector $\mathbf{r}=[r_1,...,r_n]$ e $\mathbf{r}=\mathbf{a}\beta\mathbf{b}$ ou $\mathbf{r}=\mu\mathbf{a}$, onde β representa uma operação lógica binária e μ representa uma operação lógica unária. Uma operação lógica binária aplicada a dois vectores **a** e **b** pode ser considerada como um conjunto de *n* operações semelhantes sobre os elementos relevantes de ambos os vectores. Portanto, a RFU é construída das *n* primitivas computacionais reprogramáveis $P_1,...,P_n$ (ver fig. 5.12). Cada P_i, $i=1,...,n$, possui duas entradas a_i e b_i e uma saída r_i. Cada variável a_i, b_i, $i=1,...,n$, é composta por dois bits o primeiro dos quais provem dos blocos *U_ones (*ou U^T*_ones*) e o segundo dos blocos *U_zeros* (ou U^T*_zeros*). Deste modo são suportadas quaisquer operações lógicas sobre vectores booleanos e ternários. As primitivas são construídas com base em 2 CLBs da FPGA XC4010XL. A reprogramação é conseguida através da configuração dos CLBs como *dual-port* RAM de 16×2. A primeira porta da RAM serve para implementar a funcionalidade desejada. A segunda porta permite efectuar a reprogramação da primitiva. Para as operações lógicas unárias utiliza-se a mesma estrutura de primitivas computacionais (um exemplo de tal operação está representado na parte inferior direita da fig. 5.12).

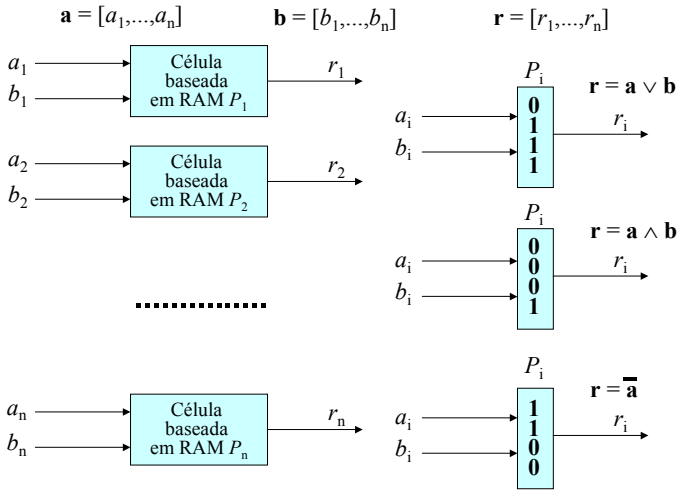


Fig. 5.12. Implementação de operações lógicas binárias e unárias sobre vectores booleanos.

O segundo grupo abrange as operações binárias cujo resultado pode ser representado na forma *sim* ou *não*. Por exemplo, para os dois vectores **a** e **b** precisa-se verificar se estes se encontram na relação de ortogonalidade ou se o vector **a** domina o vector **b**. Neste caso as operações primárias são executadas em primitivas computacionais $P_1,...,P_n$ e as saídas das primitivas são ligadas a um bloco lógico que executa um número muito limitado de operações lógicas elementares, tais como *AND* e *OR*, que são seleccionadas pelo código de operação respectivo (ver fig. 5.13), i.e. a RFU realiza a operação $f(\mathbf{r}=\mathbf{a}\beta\mathbf{b})\in\{0, 1\}$, onde *0* representa a resposta *não*, *1* corresponde à resposta *sim* e $f\in\{O_1, O_2,\ldots\}$, em que O_1, O_2,… são as operações permitidas do bloco lógico (ver fig. 5.13).

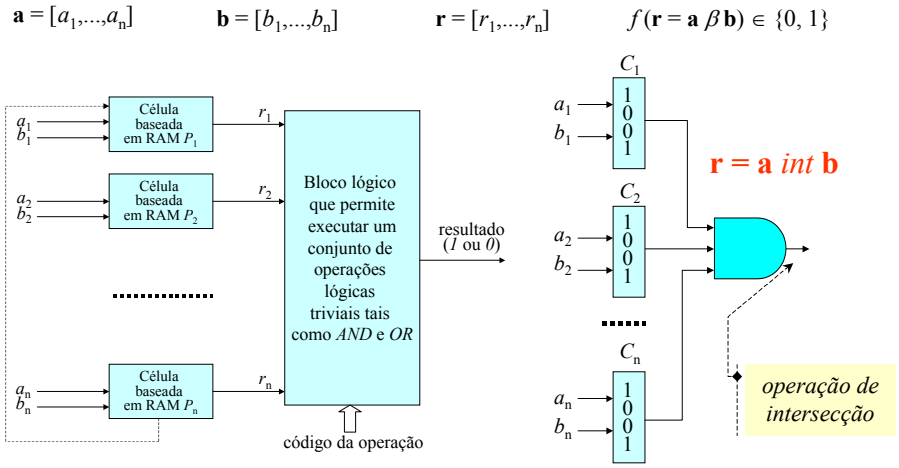


Fig. 5.13. Implementação de operações binárias cujo resultado é *sim* ou *não* sobre vectores booleanos.

O terceiro grupo inclui as operações cujo resultado é um número que normalmente tem mais que um bit mas menos que *n* bits. Um exemplo deste tipo de operações é o cálculo do número de *1*s num vector booleano [11010110] que é igual a $101_2=5_{10}$. Uma abordagem possível a implementar tais operações é a seguinte. O vector é dividido em segmentos de tamanho predeterminado (ver fig. 5.14). Cada segmento é analisado num grupo *G* de células baseadas em RAM. Os resultados de todos os segmentos entram num somador-acumulador que calcula o resultado final. A reprogramação das operações é conseguida ao reprogramar os blocos de RAM.

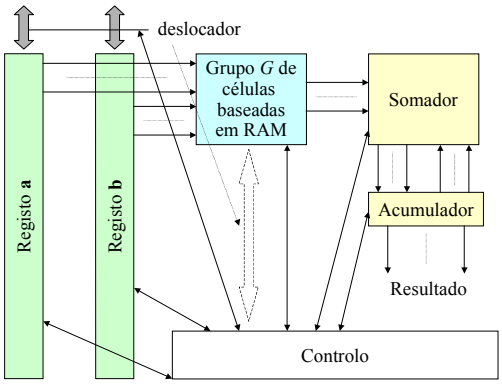


Fig. 5.14. Implementação do terceiro grupo de operações.

5.4 Implementação

Uma versão simples do RCP foi implementada com base em placa XStend da XESS [XESS] que contém uma FPGA XC4010XL da Xilinx [Xilinx00]. Os blocos reprogramáveis do RCP foram construídos a partir das LUTs disponíveis em blocos lógicos desta FPGA. A reconfiguração do RCP foi executada através da porta paralela conforme representado na fig. 5.15. Na parte inferior da fig. 5.15. está ilustrado o formato da instrução utilizado para a configuração da unidade de controlo, em que:

- *clk* – é o sinal de sincronização;
- *prog* – distingue os modos de execução e de configuração;

- *ram* – selecciona o bloco de RAM apropriado para ser reprogramado;
- *addr_data* – especifica os endereços/dados para uso no bloco de RAM respectivo.

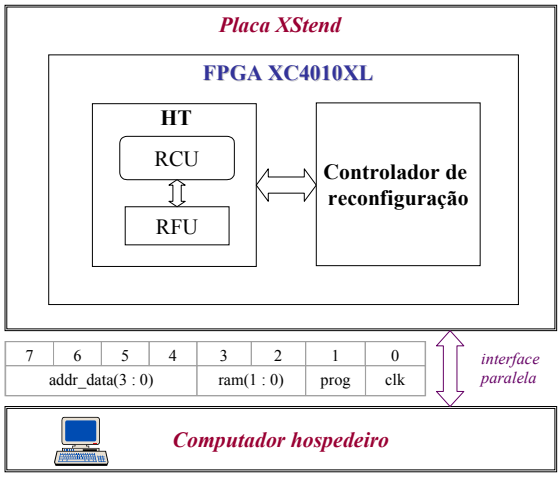


Fig. 5.15. Configuração do RCU.

Obviamente a reconfiguração através da porta paralela não permite aproveitar todas as capacidades da técnica proposta. Para além disso a FPGA XC4010XL possui recursos bastante limitados, consequentemente esta implementação só foi utilizada para validar a possibilidade de reconfiguração dinâmica parcial e para algumas experiências.

A FPGA é configurada conforme representado na fig. 5.16. As funções básicas de algoritmos combinatórios (tais como *find-max-column*, *find-ort-row*) relevantes são incluídas numa biblioteca. A especificação ao nível de sistema faz-se com a ajuda da linguagem C++, i.e. os algoritmos combinatórios são descritos em C++ aproveitando as funções da biblioteca. Quando necessário, as ferramentas de software auxiliares extraem a configuração correspondente da biblioteca e utilizam-na para programar o RCP. O HT em si é carregado logo no início e durante a execução de vários algoritmos só é preciso configurar os componentes modificáveis existentes em RCU e RFU. Deste modo o *overhead* da reconfiguração é substancialmente reduzido. Finalmente, os dados da matriz são transferidos para a FPGA para processamento. Quando as computações no RCP são concluídas, os resultados são despachados para o computador hospedeiro e a execução da aplicação é prosseguida podendo eventualmente encontrar uma outra função de hardware que irá desencadear acções semelhantes, i.e. o processo considerado acima será repetido.

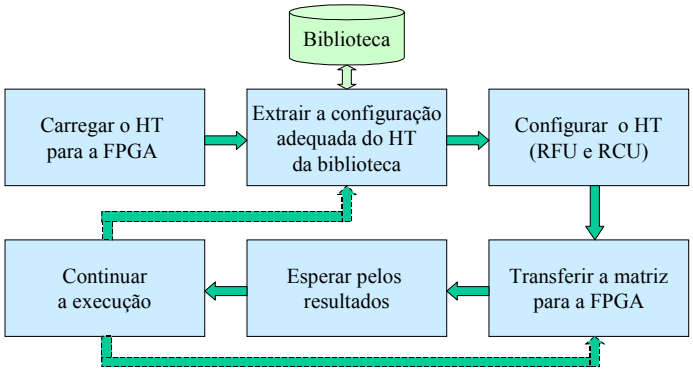


Fig. 5.16. Configuração do RCP.

5.5 Experiências

Para avaliar o desempenho do RCP realizámos uma série de experiências com dois problemas combinatórios que são a satisfação booleana [Gu97] e a cobertura [Cormen97].

5.5.1 Problema de satisfação booleana

O problema de satisfação booleana exige determinar se uma função booleana pode ser satisfeita (i.e. pode tornar-se igual a *1*) com alguma atribuição de valores às variáveis. Quando o problema é formulado como o de pesquisa, pede-se para encontrar pelo menos uma atribuição satisfatória. A função está normalmente representada em CNF (*Conjunctive Normal Form*), também conhecida sob o nome de *produto de somas* (POS – *Product-of-Sums*). A CNF é composta por conjunção de um número de cláusulas, onde a cláusula é a disjunção de algumas variáveis ou das suas negações. O problema de satisfação booleana possui um papel de grande importância na área de projecto assistido por computador e será analisado em muito detalhe no capítulo 6.

Consideremos a função seguinte que contém 4 variáveis e 3 cláusulas, e é satisfeita quando $x_1=0$, $x_2=0$, $x_3=1$ e $x_4=1$ (esta é uma das soluções possíveis):

$$(\bar{x}_1 \vee x_4)(x_3)(x_1 \vee \bar{x}_2 \vee \bar{x}_3)$$

Para resolver este problema com a ajuda do RCP devemos formulá-lo sobre uma matriz lógica **U**. Para tal vamos estabelecer uma correspondência entre as variáveis e cláusulas da função e as colunas e linhas de **U**, respectivamente. Cada elemento u_{ij}, $i=1,\ldots,m$, $j=1,\ldots,n$, da matriz é igual a:

- *0* – se a variável x_j entra na cláusula c_i com negação;
- *1* – se a variável x_j entra na cláusula c_i sem negação;
- *'-'* (*don't care*) – se a variável x_j não entra na cláusula c_i.

Aplicando estas regras de codificação pode-se construir a matriz **U** correspondente à função representada acima:

$$\mathbf{U} = \begin{array}{c} \begin{matrix} \mathbf{x_1} & \mathbf{x_2} & \mathbf{x_3} & \mathbf{x_4} \end{matrix} \\ \begin{bmatrix} \mathbf{0} & - & - & \mathbf{1} \\ - & - & \mathbf{1} & - \\ \mathbf{1} & \mathbf{0} & \mathbf{0} & - \end{bmatrix} \end{array} \begin{matrix} \mathbf{c_1} \\ \mathbf{c_2} \\ \mathbf{c_3} \end{matrix}$$

Implementámos a solução determinística deste problema que gera e examina exaustivamente todas as atribuições possíveis de valores às variáveis. É de notar que tal abordagem de força bruta não é muito eficiente. Contudo, esta pode ser usada para avaliar a eficácia da arquitectura proposta. Uma solução mais eficiente será apresentada no capítulo 6.

O algoritmo empregue foi descrito com a ajuda do GS ilustrado na fig. 5.17. O problema pode ser resolvido aplicando a sequência seguinte de acções. Primeiro, construímos a matriz **U**, carregamos o algoritmo de controlo da fig. 5.17 para a RCU e configuramos a RFU para que esta implemente a operação de verificação se dois vectores são ortogonais.

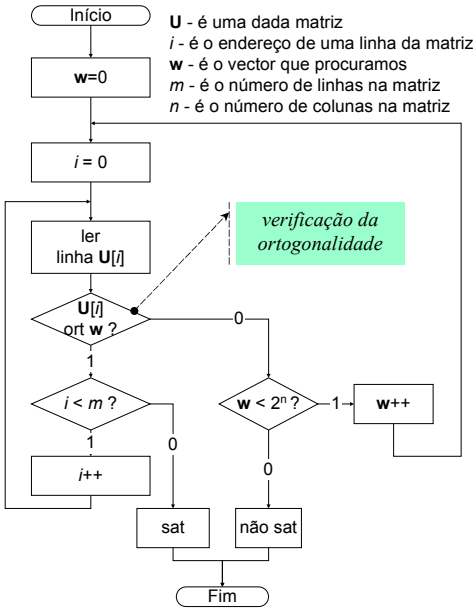


Fig. 5.17. Algoritmo implementado para o problema de satisfação booleana.

De acordo com o algoritmo da fig. 5.17 devemos encontrar um vector binário **w** que seja ortogonal a todas as linhas da matriz **U**. Dois vectores $\mathbf{a}=[a_1,\ldots,a_n]$ e $\mathbf{b}=[b_1,\ldots,b_n]$ consideram-se ortogonais se existe $j=1,\ldots,n$, tal que $a_j=0$ e $b_j=1$, ou $a_j=1$ e $b_j=0$. Se o vector **w** não pode ser encontrado, então o problema não tem solução, i.e. a função é insatisfazível. Em caso oposto, o vector **w** negado representa a solução, i.e. todas as suas componentes com valor *0* apontam para as variáveis que devem receber valor *1* e todos os seus elementos com valor *1* indicam que variáveis devem ser iguais a *0*.

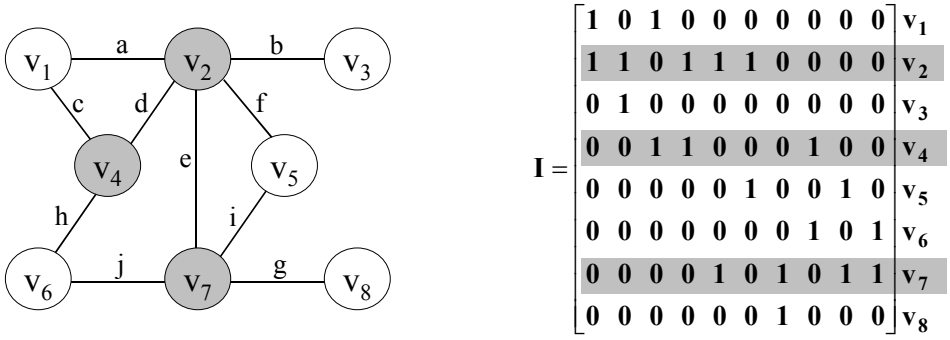
5.5.2 Problema de cobertura

A cobertura é um problema combinatório bem conhecido que possui muitas aplicações práticas. O problema requer encontrar num dado conjunto um subconjunto que satisfaça certas restrições. Quando formulado como o de optimização, pede-se para descobrir o subconjunto de potência mínima.

Suponhamos que é preciso encontrar a cobertura mínima de vértices (*minimum-size vertex cover*) dum grafo. É conhecido que a cobertura de vértices dum grafo $G=(V, E)$ é um subconjunto $V' \subset V$ tal que se uma aresta $(u,v) \in E$, então $u \in V'$ ou $v \in V'$ (ou ambas as condições são satisfeitas) [Cormen97].

Para resolver este problema com a ajuda do RCP devemos formulá-lo sobre uma matriz lógica. Para tal podemos construir a matriz de incidência **I**, cujas colunas correspondem às arestas do grafo e cujas linhas representam os vértices. Agora para resolver o problema basta encontrar a cobertura mínima da matriz **I** composta pelo número mínimo de linhas que contenham em conjunto pelo menos um valor *1* em cada coluna de **I**. As linhas seleccionadas correspondem aos vértices que devem ser incluídos na cobertura mínima de vértices.

Consideremos um exemplo. Para o grafo representado na fig. 5.18a a matriz de incidência **I** tem o formato ilustrado na fig. 5.18b.



a) b)

Fig. 5.18. Cobertura mínima de vértices do grafo (a); Matriz de incidência respectiva (b).

Para encontrar a cobertura mínima de uma matriz booleana aplicámos o algoritmo aproximado descrito na fig. 5.19 [Zakrevski81]. A operação básica executada pelo algoritmo consiste em calcular o número de *1*s em várias linhas e colunas da matriz. Assim, o problema pode ser resolvido aplicando os passos seguintes. Primeiro, a matriz **I** é construída, a seguir o algoritmo de controlo é carregado para a RCU. Finalmente, a RFU é configurada para implementar a operação *calcular-número-de-1s*.

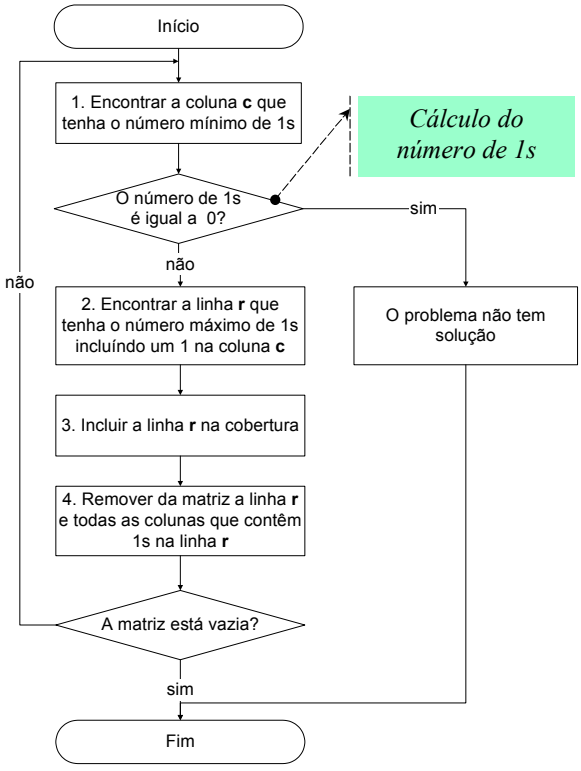


Fig. 5.19. Algoritmo implementado para o problema de cobertura da matriz.

O subconjunto encontrado está destacado com a cor cinzenta no grafo da fig. 5.18a e na matriz **I** ilustrada na fig. 5.18b. Neste exemplo foi possível descobrir a solução óptima mas no caso geral o algoritmo aplicado não garante isso.

5.5.3 Resultados

A tabela 5.3 contém os resultados das experiências com os dois problemas combinatórios descritos acima e resolvidos no RCP incluindo a sua comparação com a implementação dos mesmos algoritmos em software (executado num PIII/800MHz/256MB com o sistema operativo Windows2000). As linhas *Cob1*…*Cob6* contêm os resultados para as instâncias aleatórias do problema de cobertura, e as linhas *Sat1*...*Sat6* – para o problema de satisfação booleana.

A primeira linha na tabela 5.3 representa um exemplo da operação trivial que calcula o número de *1*s num vector booleano. Esta operação, quando executada no RCP, consome muito menos ciclos de relógio do que num computador de uso geral.

Para o problema de cobertura a aceleração obtida é bastante significativa. Como se pode ver na fig. 5.19 a operação primária do algoritmo considerado é o cálculo do número de *1*s em várias linhas e colunas da matriz. Conforme mencionado acima, esta operação é executada no RCP muito mais rapidamente do que em software. Contudo, o problema é bastante orientado para o controlo, portanto a aceleração da operação básica não conduz à aceleração total porque outras porções do algoritmo não são executadas mais eficientemente do que em software. Esta circunstância impede alcançar desempenho mais elevado.

Para o problema de satisfação booleana a aceleração conseguida é mais impressionante. Isto pode ser explicado pelo facto do algoritmo considerado (ver fig. 5.17) só requerer a leitura sequencial de diferentes linhas da matriz e, a seguir, a verificação se os vectores respectivos são ortogonais. Cada uma destas operações requer um ciclo de relógio. Em software a matriz foi construída como um *array* de inteiros. Consequentemente, a verificação se dois vectores se encontram em relação de ortogonalidade requer muitos acessos à memória e computações bastante intensivas.

É de salientar que os valores apresentados na tabela 5.3 reflectem somente o tempo de execução do algoritmo respectivo não tomando em consideração o tempo de configuração do RCP.

Tabela 5.3. Resultados das experiências com o RCP implementado em FPGA XC4010XL.

Problema	**Dimensões de vector/matriz**	**Tempo de execução em software (*ms*)**	**Tempo de execução no RCP (*ms*)**	**Aceleração**
Calcular o número de *1*s num vector	8	0.0059	0.00002487	237
Cob1	8 × 8	0.284	0.0110	25.82
Cob2		0.310515	0.0144	24.56
Cob3		0.217905	0.0065	33.52
Cob4		0.17684	0.00423	41.81
Cob5		0.292075	0.00924	31.61
Cob6		0.227125	0.00274	82.89
Sat1	8 × 8	4.54248	0.02971	152.89
Sat2		5.22301	0.01508	346.35
Sat3		0.24891	0.000588	423.32
Sat4		11.75764	0.07485	157.08
Sat5		2.72465	0.008059	338.09
Sat6		0.51543	0.002647	194.72

5.6 Recursividade

Uma das limitações do RCP descrito é que este não suporta algoritmos recursivos enquanto estes são muito comuns no domínio das computações combinatórias. Todos os algoritmos de retrocesso são normalmente implementados como recursivos. Portanto, é muito importante suportar esta característica no RCP.

5.6.1 Algoritmos de retrocesso

A fig. 5.20 representa a estrutura básica de algoritmos de retrocesso que serão utilizados para resolver problemas formulados sobre matrizes lógicas. Os algoritmos começam por aplicar técnicas de redução tentando simplificar o problema. A seguir, ou é efectuada a decomposição do problema original num conjunto de subproblemas e o mesmo algoritmo é invocado de modo recursivo para cada um destes, ou são novamente aplicadas as técnicas de redução. Se o algoritmo conseguir encontrar alguma solução boa esta é memorizada. Caso contrário, as reduções e eventuais decomposições são executadas até encontrar uma solução melhor ou até que a qualidade da solução (parcial) corrente fique pior que a qualidade de qualquer solução encontrada previamente. De seguida, o algoritmo retrocede (i.e. efectua um retorno hierárquico) até ao ponto de decomposição mais recente e analisam-se os subproblemas restantes.

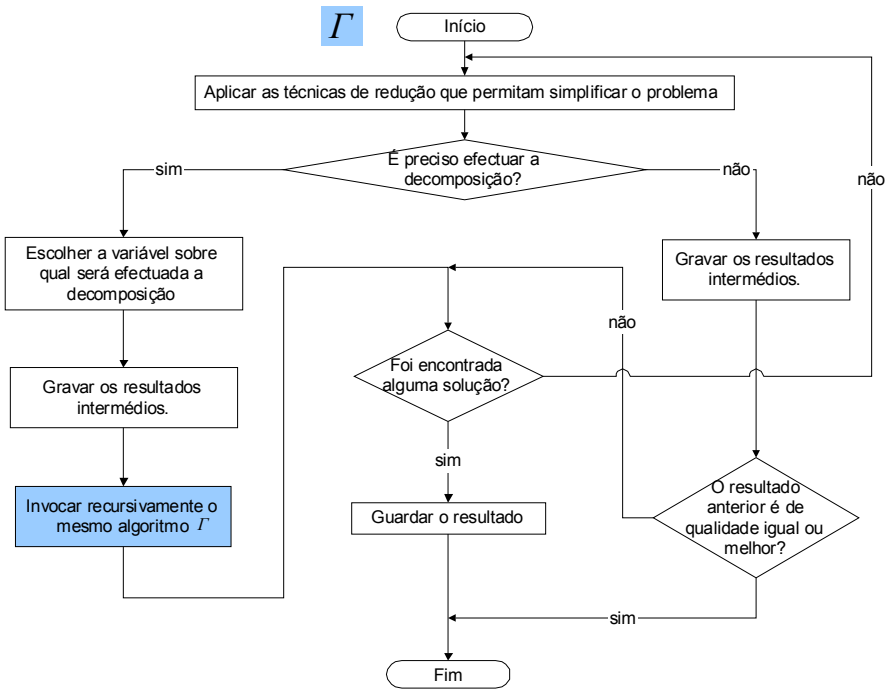


Fig. 5.20. Estrutura básica de algoritmos de retrocesso.

Demonstremos os passos principais deste tipo de algoritmos com a ajuda do problema de cobertura (composta por colunas) de uma matriz booleana. Para tal adaptámos o algoritmo exacto proposto em [Zakrevski81]. Suponhamos que é dada a matriz representada na fig. 5.21 cuja cobertura mínima é composta por colunas $\{c, g\}$. A fim de simplificar a matriz utilizemos as técnicas de redução seguintes:

- São eliminadas as linhas dominantes, i.e. se $linha_i \wedge linha_j = linha_j$, $i \neq j$, $i,j=1,\ldots,m$, então a $linha_i$ é removida da matriz. Por exemplo, $linha_8 \wedge linha_7 = linha_7$ (ver fig. 5.21), consequentemente a $linha_8$ pode ser apagada.
- São eliminadas as colunas dominadas, i.e. se $coluna_i \wedge coluna_j = coluna_i$, $i \neq j$, $i,j=1,\ldots,n$, então a $coluna_i$ é removida da matriz. Por exemplo, $coluna_d \wedge coluna_c = coluna_d$ (ver fig. 5.21), consequentemente a $coluna_d$ poder ser apagada.
- A solução não existe se houver alguma linha que não contenha nenhum valor *1*.

a	b	c	d	e	f	g	h	i	j	k	l	
0	0	1	1	1	0	1	0	1	0	1	1	1
0	1	0	0	1	1	1	1	0	1	0	0	2
0	0	1	1	0	0	0	1	0	0	0	1	3
1	0	1	0	1	1	1	0	0	1	0	0	4
1	0	0	0	1	0	1	1	0	1	1	1	5
1	1	1	1	0	0	0	0	1	0	1	0	6
0	0	1	0	1	1	1	0	1	0	0	1	7
1	0	1	1	1	1	1	1	1	1	1	1	8
1	1	1	1	1	1	1	1	1	1	1	1	9

Fig. 5.21. A cobertura mínima desta matriz é composta por colunas *c* e *g*.

Para a decomposição utilizemos as regras seguintes:

- Se uma linha contiver apenas uma componente igual a *1* então a coluna respectiva (i.e. a coluna na qual aparece este *1*) deve ser incluída na cobertura.
- Se todas as linhas contêm mais que um valor *1* então escolhe-se a linha com o número mínimo de *1*s. Para esta linha é necessário analisar todas as situações possíveis e o número *k* destas situações é igual ao número de *1*s. Portanto, o problema decompõe-se em *k* subproblemas cada um dos quais precisa de ser resolvido. Cada subproblema é processado chamando o mesmo algoritmo de maneira recursiva. A resolução de qualquer subproblema termina ao detectar que a qualidade da solução (completa ou parcial) encontrada é igual à qualidade de uma das soluções descobertas previamente (na resolução de subproblemas anteriores).

Naturalmente, todas as colunas incluídas na cobertura bem como todas as linhas que contêm *1*s nestas colunas, são eliminadas da matriz.

A fig. 5.22 ilustra todos os passos que é necessário executar a fim de encontrar a cobertura mínima da matriz da fig. 5.21. O caminho na árvore de pesquisa que conduz à solução óptima {*c*, *g*} está destacado na fig. 5.22 com as setas duplas. Nesta árvore de pesquisa há dois pontos de ramificação: *b*-*g*-*h* e *c*-*l*. Depois de descobrir a primeira solução {*b*, *c*, *l*} que inclui 3 elementos, só estamos interessados em coberturas compostas por 2 ou menos colunas. Sendo assim, não é necessário percorrer completamente todos os ramos da árvore, podendo suspender a pesquisa para diante em qualquer ponto que produz uma solução parcial que já inclui duas colunas (ver as etiquetas *parar* na fig. 5.22).

É de notar que os algoritmos de retrocesso são amplamente aplicados para a solução de muitos outros problemas combinatórios, tais como satisfação booleana, coloração de grafos, etc. As características distintivas destes algoritmos são as seguintes [Sklyarov03a]:

1) Estes são recursivos e consequentemente requerem uma unidade de controlo que suporte a recursividade.
2) No processo de pesquisa os dados iniciais (i.e. a matriz original) não são modificados. As técnicas de redução podem ser implementadas ao mascarar linhas e colunas relevantes (que devem ser eliminadas) e utilizar só o menor restante da matriz.
3) Os algoritmos requerem um número muito limitado de operações que são aplicadas a grandes quantidades dos dados.
4) Os subconjuntos de operações empregues por vários algoritmos são diferentes [Sklyarov01]. Portanto, as operações devem ser implementadas na unidade funcional reconfigurável para as necessidades de um dado algoritmo (conforme representado na secção 5.3.3).
5) Para efectuar a pesquisa para diante e suportar o processo de retrocesso, há-de existir uma pilha que guarde e reconstitua os resultados intermédios (por exemplo, os valores das máscaras para as linhas e colunas) em todos os pontos de ramificação.
6) Os algoritmos de retrocesso podem ser decompostos em dois níveis de operações de controlo. A sequência do nível superior é recursiva e é semelhante para problemas diferentes. A sequência do nível inferior permite executar operações sobre vectores lógicos, que são normalmente diferentes para todos os problemas.

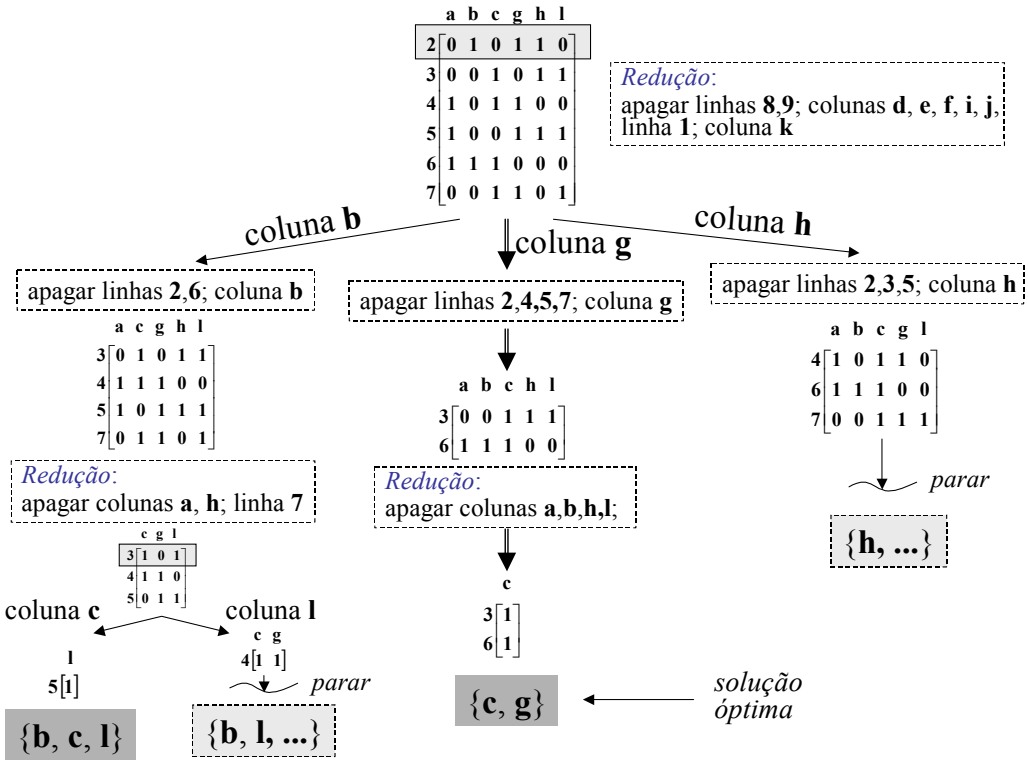


Fig. 5.22. Utilização do algoritmo exacto descrito para encontrar a cobertura mínima da matriz.

5.6.2 Arquitectura aperfeiçoada do RCP

Ao analisar todas as características comuns de algoritmos de retrocesso tornou-se possível propor uma arquitectura aperfeiçoada do processador combinatório, representada na fig. 5.23. O processador é composto pelas unidades seguintes:

1) Blocos de RAM que guardam os dados da matriz. Estes blocos são construídos conforme descrito na secção 5.3.1.
2) Cinco pilhas diferentes que permitem guardar os resultados intermédios em todos os pontos de ramificação a fim de suportar o retrocesso caso seja necessário. As pilhas servem para manter os dados seguintes:
   - Máscaras para as linhas da matriz.
   - Máscaras para as colunas da matriz.
   - Resultados intermédios em pontos de ramificação (i.e. as colunas que compõem a solução parcial corrente).
   - Máscaras para os ramos da árvore de pesquisa que já tenham sido percorridos.
   - Os valores dos critérios de avaliação que permitem escolher a variável sobre a qual será efectuada a decomposição (pilha auxiliar na fig. 5.23). Vamos referenciar esta variável por *variável de decisão*. Para o problema de cobertura considerado, a pilha auxiliar contém o número de *1*s existentes em cada linha e coluna da matriz. Estes valores devem ser actualizados sempre que seja eliminada alguma linha ou coluna com *1*s.
3) Registos de uso geral que servem para guardar os resultados intermédios.

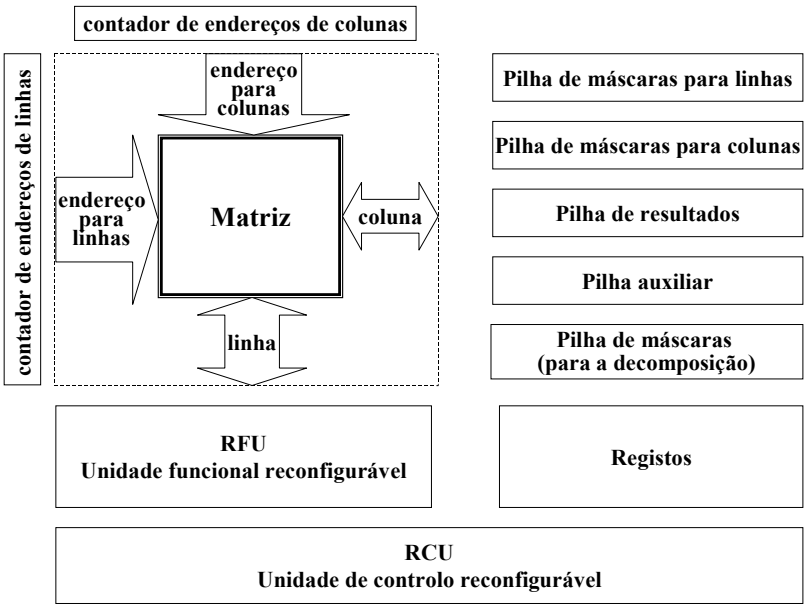


Fig. 5.23. Arquitectura do processador para implementação de algoritmos de retrocesso sobre matrizes lógicas.

4) Unidade funcional reconfigurável (ver secção 5.3.3). A característica de reconfiguração possibilita a utilização desta unidade para a solução de problemas combinatórios diferentes.

5) Unidade de controlo reconfigurável que gere o funcionamento de todos os outros componentes.

Consideremos como utilizar a arquitectura proposta para resolver o problema de cobertura descrito na secção anterior. A fig. 5.24 ilustra os passos iniciais de execução do algoritmo que permitem encontrar a primeira solução possível $\{b, c, l\}$. A fig. 5.25 ilustra o processo de retrocesso para o ponto de ramificação *c*-*l* (ver a parte inferior esquerda da fig. 5.22).

A fig. 5.26 representa as operações básicas do algoritmo de controlo do nível superior que são realizadas pela RCU ilustrada na parte inferior da fig. 5.23. Este algoritmo activa o algoritmo de retrocesso Γ mostrado na fig. 5.20.

Analisemos em mais detalhe os passos 1-12 das fig. 5.24 e 5.25 (os números dos passos estão incluídos em círculos):

1) Inicialmente, todos os registos são inicializados a zero (ver o bloco *Inicializar todos os registos* da fig. 5.26). A anulação das máscaras de linhas e de colunas significa que começamos por processar a matriz completa. De acordo com as regras de redução (ver secção 5.6.1) as linhas *8*, *9* e as colunas *d*, *e*, *f*, *i*, *j* são removidas da matriz. Como resultado, os dois bits da máscara de linhas que correspondem às linhas *8* e *9*, e os cinco bits da máscara de colunas que correspondem às colunas *d*, *e*, *f*, *i*, *j* mudam de *0* para *1*.

2) Pode-se aplicar as regras de redução mais uma vez e apagar a linha *1* e a coluna *k*.

3) De acordo com as regras de decomposição selecciona-se a linha *2* (a primeira linha com o número mínimo de *1*s). De seguida determina-se quais as colunas que têm valor *1* na linha *2* e escolhe-se a primeira destas – a coluna *b*. A linha seleccionada (*2*) é removida da matriz. Dado que este é um ponto de ramificação precisamos de guardar em pilhas todos os dados necessários para poder posteriormente retroceder a fim de examinar outros ramos da árvore de pesquisa (as colunas *g* e *h*). Os dados a guardar são os seguintes: a máscara 110000011 para as linhas; a máscara 000111001110 para as colunas; os números de *1*s para todas as linhas 003343300, onde os *0*s indicam que as linhas respectivas foram eliminadas da matriz e o valor mais pequeno (*3*) denota o número mínimo de *1*s; a máscara 010000000000 para marcar a variável de decisão corrente (que corresponde à coluna *b*); e finalmente a solução parcial que actualmente não contém nenhuma coluna. É de notar que as posições dos *0*s na pilha auxiliar coincidem com as posições de *1*s na pilha de máscaras para as linhas o que permite substituir estas duas pilhas por uma única.

4) A linha *6* (porque esta contém valor *1* na coluna *b*) e a coluna *b* são apagadas o que resulta em modificações dos valores de registos ilustradas na fig. 5.24.

5) De acordo com as regras de redução as colunas *a* e *h* são removidas da matriz. Como resultado, os dois bits do registo de máscaras para as colunas alteram de *0*s para *1*s.

6) Ao aplicar as regras de redução, apaga-se a linha *7*. Como resultado, um bit do registo de máscaras para as linhas muda de *0* para *1*.

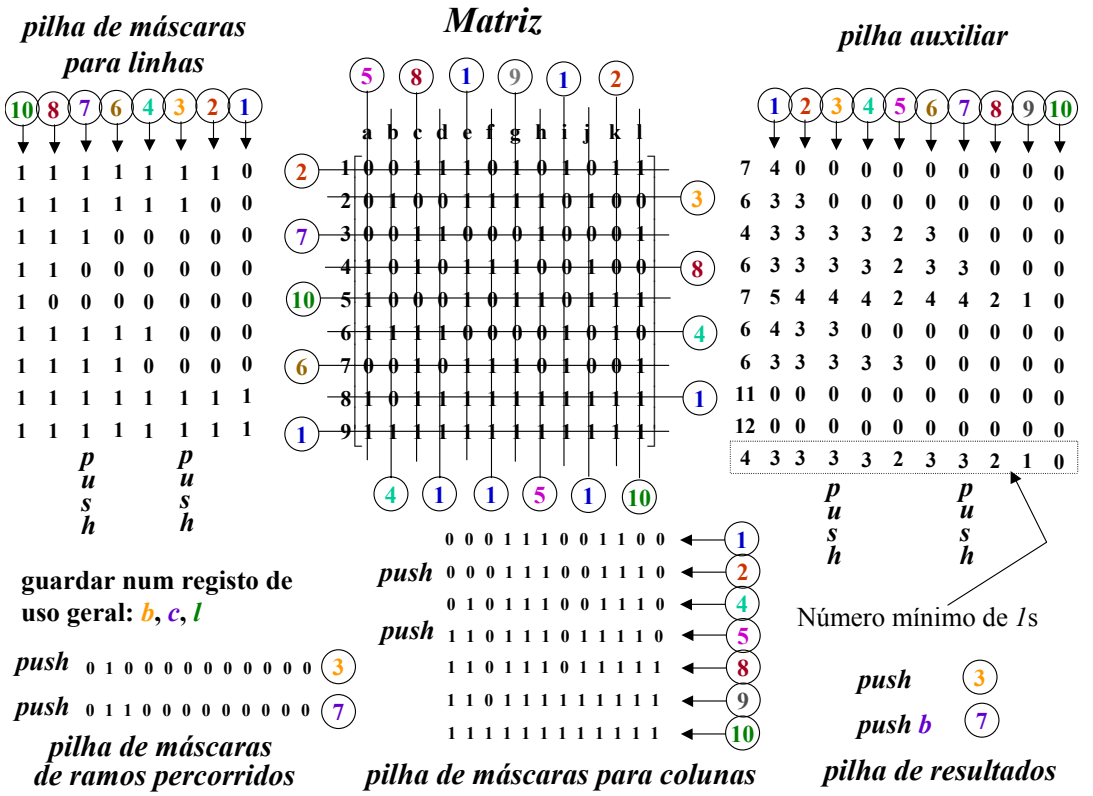


Fig. 5.24. Execução do algoritmo de retrocesso para resolver o problema de cobertura no processador combinatório.

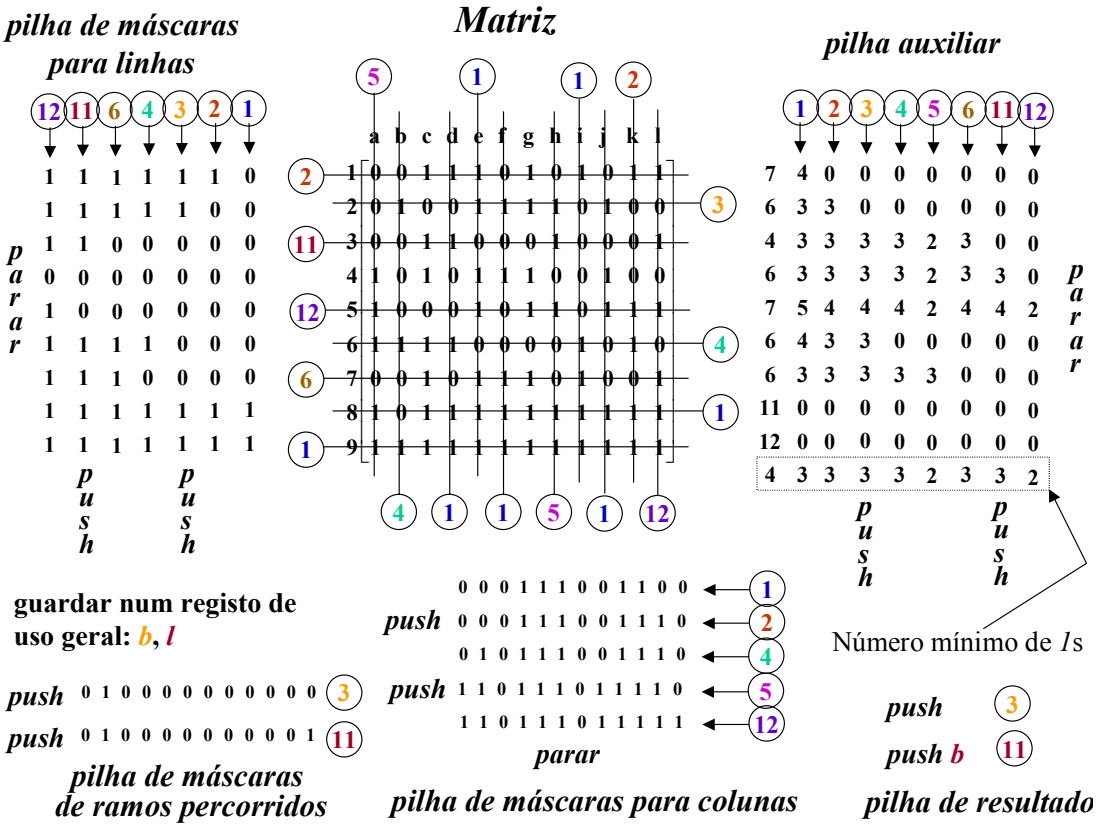


Fig. 5.25. Retrocesso para o ponto de ramificação *c-l*.

7) Dado que é impossível aplicar novamente as regras de redução, recorre-se à decomposição. Para tal selecciona-se a linha *3* (a primeira linha que contém o número mínimo de *1*s). A seguir escolhe-se a primeira coluna (*c*) que tem valor *1* nesta linha. A linha seleccionada (*3*) é removida da matriz. Para poder recuperar a situação actual no futuro (a fim de analisar as colunas restantes – *l* no nosso caso), os valores de todos os registos são guardados nas pilhas respectivas.

8) A linha *4* (porque esta contém valor *1* na coluna *c*) e a coluna *c* são apagadas o que resulta em modificações dos valores de registos ilustradas na fig. 5.24.

9) De acordo com as regras de redução a coluna *g* é removida da matriz.

10) A linha *5* contém apenas uma componente igual a *1*. Portanto a coluna *l* que corresponde a esta componente deve ser incluída na cobertura resultando na primeira solução possível: {*b*, *c*, *l*}. A solução encontrada bem como a sua qualidade (expressa pelo número de colunas que a compõem - 3) são guardados em registos de uso geral. Estes valores serão utilizados para cortar todos os ramos da árvore de pesquisa que conduzem a soluções constituídas por mais que 2 colunas.

11-12) Dado que temos encontrada uma solução possível, devemos retroceder a fim de explorar outras regiões do espaço de pesquisa com o objectivo de descobrir alguma solução de melhor qualidade. Portanto, o ponto de ramificação mais recente (*c*-*l*) é recuperado das pilhas, executando para tal a operação *pop* que reconstitui os valores de todos os registos relevantes. De seguida todos os passos do algoritmo Γ representado na fig. 5.20 são repetidos. Para o exemplo considerado, a primeira solução encontrada não é o óptimo global. Conforme ilustrado na fig. 5.22 existem duas soluções com a qualidade melhor: {*c*, *g*} e {*c*, *h*}.

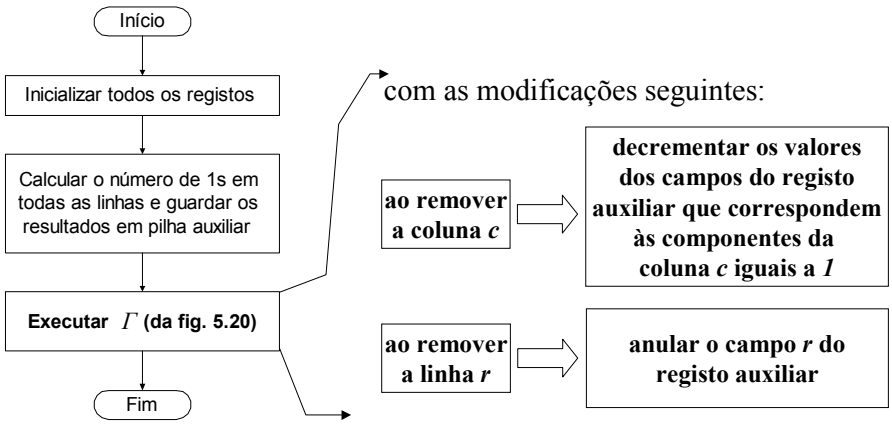


Fig. 5.26. Operações básicas do algoritmo de controlo do nível superior.

A arquitectura aperfeiçoada do processador combinatório foi modelada em C++ e alguns dos seus componentes foram implementados em FPGA. A especificação da arquitectura completa com a ajuda da linguagem Handel-C e a sua implementação em FPGA XC2S200 da família Spartan-II disponível na placa RC100 da Celoxica [Handel-C] foi proposta a alunos do quinto ano no âmbito do projecto do fim de curso.

5.7 Partição da aplicação entre software e hardware reconfigurável

A arquitectura descrita neste capítulo satisfaz a restrição imposta: esta não é alterada de problema para problema, só sendo necessário configurar a RFU e a RCU e transferir os dados para as matrizes lógicas. Em todos os exemplos considerados nas secções anteriores cada tarefa foi resolvida completamente em FPGA. É de salientar contudo que o RCP limita as dimensões máximas das matrizes e consequentemente não pode ser usado para abordar uma instância arbitrária. De acordo com as dimensões máximas da matriz permitidas podem ocorrer as três situações seguintes ilustradas na fig. 5.27:

1) A instância de problema é muito pequena e simples (ver fig. 5.27a). Neste caso o uso da FPGA torna-se despropositado dado que se gasta bastante tempo a reconfigurar a FPGA e o HT (no nosso caso), ou a gerar o circuito de hardware (na abordagem orientada para a instância). De qualquer modo, todas as vantagens do hardware rápido serão perdidas e uma implementação baseada em software seria muito mais eficiente.

2) A instância do problema é difícil e as suas dimensões enquadram-se nas dimensões da matriz permitidas em FPGA (ver fig. 5.27b). Neste caso possivelmente atinge-se um desempenho muito bom capaz de absorver o tempo de configuração do hardware. Contudo, os problemas reais raramente "cabem" nas dimensões limitadas do HT.

3) As dimensões do problema são muito grandes excedendo em muito a capacidade do hardware disponível. No domínio da abordagem orientada para a instância a solução seguinte é normalmente adoptada. Se um circuito não puder ser implementado numa única FPGA, são empregues várias FPGAs interligadas aplicando para tal métodos especiais de partição do circuito entre as FPGAs. Todavia, não existe garantia alguma que uma dada instância do problema seja resolvida eficientemente com os recursos de hardware reconfigurável disponíveis.

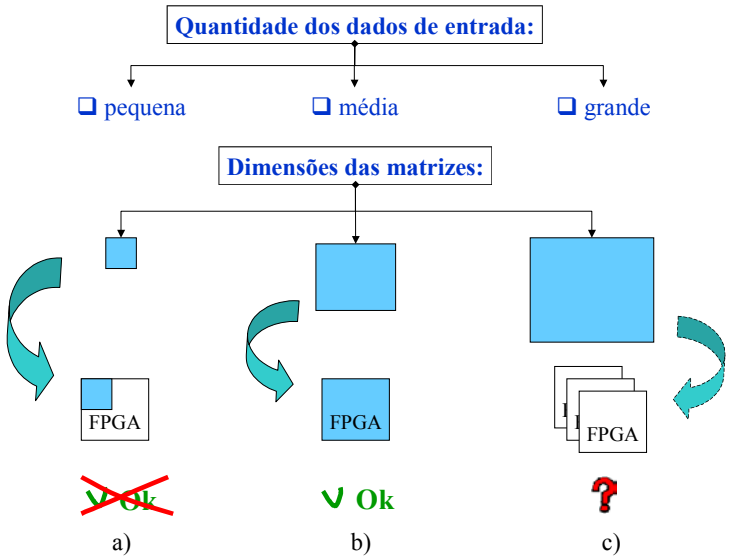


Fig. 5.27. Mapeamento de problemas em FPGA.

Para abordar esta questão propomos aplicar a estratégia seguinte que está representada na fig. 5.28. Como discutimos, a técnica comum de resolução de problemas combinatórios baseia-se em árvore de pesquisa. No processo de construção da árvore de pesquisa aplicam-se vários métodos de redução e de decomposição. Isto permite que as dimensões iniciais da matriz sejam gradualmente reduzidas (ao mover-se de cima para baixo ao longo de um dos ramos da árvore).

O RCP é baseado no HT com as restrições predeterminadas no número máximo de linhas $m_{\max}$ e colunas $n_{\max}$ da matriz. No início de execução as ferramentas de software auxiliares configuram a RCU e a RFU para o algoritmo respectivo. O algoritmo relevante é também implementado numa aplicação de software. De seguida, se as dimensões da matriz original satisfazem as restrições predefinidas, então esta poderá ser transferida para a FPGA e o problema será resolvido completamente em hardware. Em caso oposto a aplicação de software vai resolvendo o problema até que as dimensões da matriz intermédia (que deve ser construída no passo corrente do algoritmo) satisfaçam as restrições. A partir daí a FPGA ficará responsável por todos os passos seguintes. Se o hardware reconfigurável encontrar a solução, o problema está resolvido e o resultado será despachado para o computador hospedeiro. Se o ramo considerado da árvore de pesquisa não permitir encontrar a solução, o controlo regressará à aplicação de software que vai continuar a percorrer a árvore de pesquisa até atingir um outro ponto em que a matriz intermédia satisfaz as restrições impostas. Os dados da matriz serão então transferidos novamente para a FPGA e esta vai tentar resolver o subproblema. Os passos considerados são repetidos até encontrar a solução ou concluir que não existe solução nenhuma. Como resultado, a árvore de pesquisa será processada em software e em hardware conforme representado na fig. 5.29.

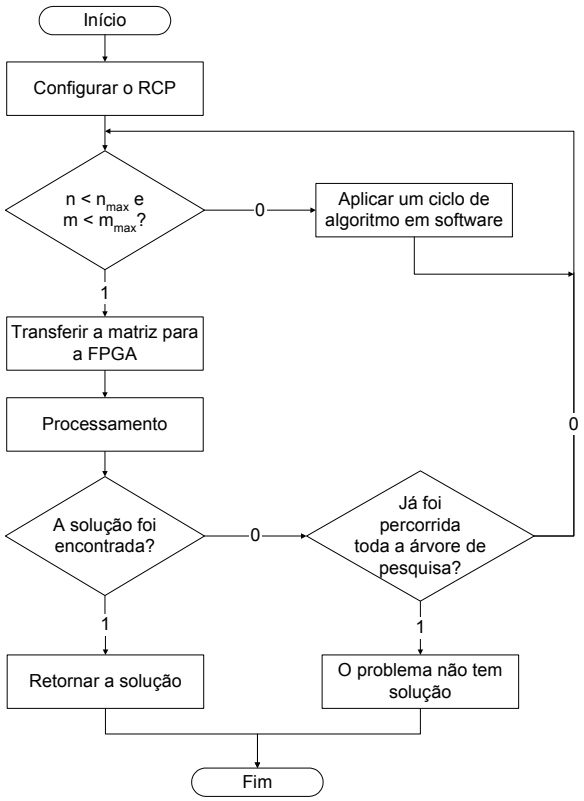


Fig. 5.28. Colaboração de software e hardware reconfigurável.

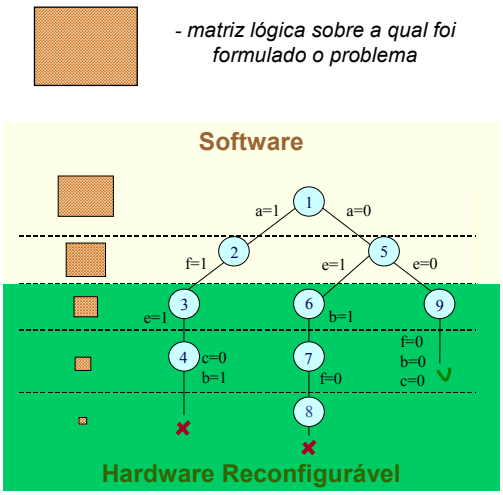


Fig. 5.29. Processamento da árvore de pesquisa em software e em hardware reconfigurável.

5.8 Discussão e trabalho relacionado

Recentemente vários grupos de investigação exploraram a possibilidade de acelerar a solução de problemas combinatórios com a ajuda do hardware reconfigurável. O problema de satisfação booleana [Abramovici00, Platzner98, Sousa01, Zhong99a] e o de cobertura [Plessl01] tornaram-se objecto da investigação mais profunda. Como mencionámos na introdução, muitas das arquitecturas propostas recorrem à abordagem orientada para a instância [Babb96, Platzner98, Plessl01, Kocan03].

Assim, no trabalho desenvolvido por Plessl *et al*. [Plessl01] propõe-se uma arquitectura específica para a instância de um acelerador destinado a resolver problemas de cobertura mínima. A arquitectura implementa um algoritmo de retrocesso semelhante ao descrito na secção 5.6.1 e inclui uma linha de FSMs que correspondem às variáveis, uma unidade de controlo, verificadores que descobrem as colunas dominadas e algumas outras condições que permitem reduzir o problema, e um bloco que calcula o custo das soluções encontradas. Para cada instância do problema é gerada (por uma aplicação de software desenvolvida pelos autores) uma descrição do circuito respectivo em VHDL que é utilizada para a síntese e implementação executadas pelas ferramentas comerciais.

A geração de uma configuração específica da FPGA para cada instância individual do problema permite aumentar o desempenho e assegura uma boa utilização dos recursos disponíveis. Neste caso o tempo total de solução dum problema é igual a "*tempo de geração do circuito de hardware*" + "*tempo de configuração da FPGA*" + "*tempo de execução*". É de notar que o tempo gasto em criar um circuito específico para a instância é bastante significativo, podendo mesmo exceder o tempo actual de execução e anular deste modo todas as vantagens da implementação rápida em hardware. Por exemplo, no acelerador de Plessl *et al*. descrito acima, o tempo de execução de umas instâncias representadas em [Plessl01] varia entre 1 *ms* e 2 *s* enquanto o tempo de mapeamento de cada uma destas instâncias na FPGA requer alguns minutos. Consequentemente, este método só pode ser utilizado eficientemente para problemas muito difíceis para os quais o tempo de compilação e de configuração do hardware é amortizado pelo tempo de execução longo. É por isso

que todos os esforços recentes são orientados para a eliminação de colocação e encaminhamento específicos para a instância. Para estes fins, são aplicadas técnicas especiais que permitem acelerar significativamente a compilação de configurações de FPGA e aumentar a frequência de funcionamento dos circuitos gerados. Estas técnicas contam com os estilos de projecto modulares [Abramovici00] e com os *hardware templates* que são personalizados automaticamente para cada instância do problema [Dandalis99, Zhong99a].

Por exemplo, Dandalis *et al*. [Dandalis99] propuseram uma técnica, orientada para os problemas formulados sobre grafos, cujo objectivo principal é eliminar o tempo excessivo despendido em compilação de hardware ao reduzir a intervenção das ferramentas de projecto assistido por computador na fase de mapeamento. Para cada instância de grafo é gerado um circuito particular. Mas os autores desenvolveram uma estrutura genérica, referenciada por esqueleto (*skeleton* - que pode ser considerado como um *hardware template* personalizável para cada instância do problema) que é derivado com base em características de um domínio específico. Um esqueleto consiste em módulos que correspondem aos elementos básicos de grafos, tais como vértices. Os módulos são organizados em projectos e a sua funcionalidade é determinada pelo algoritmo concreto. A interligação dos módulos é fixa e é bastante geral para poder servir qualquer instância de grafo. Cada esqueleto é personalizado em *run-time* para ser adaptado à precisão dos dados de entrada e à dimensão e estrutura do problema.

Contudo, os *hardware templates* empregues em aceleradores reconfiguráveis são normalmente orientados para um único problema e só podem ser personalizados para várias instâncias do mesmo problema. A fim de abordar uma tarefa diferente, mesmo que esta seja muito semelhante, é necessário projectar um circuito novo e modificar as ferramentas de software que servem para a personalização automática. Ao contrário desta abordagem, o RCP é orientado para o domínio e portanto pode ser utilizado para a solução de *vários* problemas combinatórios e de quaisquer instâncias destes problemas.

O RCP é um acelerador *parcialmente reconfigurável* dado que só pequenas partes das unidades de controlo e de execução precisam de ser reprogramadas para adequar um certo algoritmo. O tempo total de solução dum problema com a ajuda do RCP compreende três componentes: "*tempo de configuração do HT*" + "*tempo despendido em comunicação entre software e FPGA* " + "*tempo de execução*". Dado que a reconfiguração é parcial, o tempo de configuração é ínfimo comparando-o com o tempo de execução.

Uma abordagem semelhante foi seguida em CCM (*Cube Calculus Machine*) proposta em [Perkowski97, Perkowski02]. A CCM é um acelerador de hardware reconfigurável para as operações específicas do cálculo de cubos de múltiplos valores (*multiple-valued cube calculus*) que são empregues por alguns algoritmos. Para cada algoritmo deve ser instanciada uma estrutura apropriada da CCM. A CCM é orientada para a execução dos ciclos internos de algoritmos, i.e. dos ciclos que efectuam operações sobre cubos. Para cada operação, o processador hospedeiro carrega a instrução complexa respectiva na CCM e, de seguida, envia os cubos de dados relevantes para a CCM e recebe os cubos resultantes. Neste modelo de execução a comunicação entre o processador hospedeiro e a CCM é bastante intensiva e só pode ser aceitável para os sistemas fortemente interligados. Ao contrário disso, o RCP executa os algoritmos completos, não apenas o ciclo interno, reduzindo assim o *overhead* da comunicação.

Uma outra questão importante que afecta qualquer algoritmo implementado em hardware reconfigurável está relacionada com a capacidade lógica do dispositivo empregue, sendo necessárias técnicas eficientes para tratar da situação quando a instância do problema excede os recursos de hardware disponíveis. No domínio de aceleradores combinatórios foram exploradas as quatro possibilidades seguintes.

A primeira é a expansão da capacidade lógica através de interligação de várias FPGAs ficando o circuito original partilhado entre estas. É de salientar que a partição e o encaminhamento rápidos e eficientes entre dispositivos múltiplos constituem uma tarefa bastante complexa (obviamente os estilos de projecto modulares e extensíveis [Perkowski02, Zhong99a] podem atenuá-la).

O segundo método consiste na partição do problema numa série de configurações a executar em paralelo ou sequencialmente. A partição é executada decompondo o problema original num conjunto de subproblemas semi-independentes [Abramovici00]. Cada subproblema deve satisfazer as restrições de hardware impostas. A maior limitação deste método é que a eficiência da decomposição depende fortemente das características da instância do problema. Como resultado, para algumas instâncias o tempo de partição pode atingir valores inaceitáveis.

O terceiro método é baseado no esquema de hardware virtual proposto em [Sousa01, Reis02] que é semelhante ao mecanismo da memória virtual e conta com a divisão do circuito numa série de páginas de hardware que são executadas sucessivamente enquanto os resultados intermédios são guardados em blocos de memória externa. Desde que todas as páginas de hardware possuam a mesma estrutura com um número restringido de registos que são reconfiguráveis, a comutação de páginas é executada muito rapidamente.

O quarto método baseia-se na ideia de partição da aplicação entre software e hardware de acordo com a capacidade lógica disponível conforme descrito na secção 5.7 [Skliarova01b]. Obviamente, a eficiência da partição depende da estrutura do problema e da escala de implementação respectiva do RCP. No caso pior, o tempo gasto em comunicações pode constituir uma parte significativa do tempo de solução total. Contudo, as FPGAs recentes, tais como Virtex-II/Virtex-II Pro da Xilinx [Xilinx], representam as plataformas adequadas para a implementação completa do RCP e asseguram uma partição entre software e hardware mais eficiente. Para além disso, nem sequer é necessário implementar o RCP numa FPGA baseada em SRAM dado que a reconfiguração do acelerador é limitada a um número de blocos de RAM disponíveis em RCU e RFU. Portanto, o RCP pode ser construído como um ASIC equipado com os blocos de memória embutidos [Perkowski02]. Todavia, a implementação baseada numa FPGA reprogramável em sistema é mais efectiva e proveitosa dado que a FPGA pode ser utilizada para a execução de outras tarefas que serão eventualmente invocadas por outras aplicações.

5.9 Conclusões

Neste capítulo investigámos a possibilidade de utilização do hardware reconfigurável para acelerar a solução de problemas combinatórios. Ao explorar diferentes possibilidades de solução de problemas combinatórios chegou-se à conclusão que os melhores resultados podem ser obtidos com a colaboração entre uma aplicação de software executada num computador de uso geral e um circuito implementado em FPGA.

Como resultado foi desenvolvido um protótipo do coprocessador capaz de resolver problemas combinatórios formulados sobre matrizes discretas. As modificações dinâmicas necessárias para implementar diferentes operações, são efectuadas através da reprogramação das células da FPGA baseadas em RAM. As alterações de algoritmos são realizadas com a ajuda de uma unidade de controlo com comportamento modificável. Foi mostrado que a unidade de controlo pode ser sintetizada como uma FSM baseada em RAM. As vantagens do processador desenvolvido foram mostradas em experiências com dois problemas combinatórios.

A fim de suportar os algoritmos recursivos que são amplamente utilizados no âmbito da optimização combinatória, propusemos a arquitectura aperfeiçoada do processador. Para demonstrar o uso da nova arquitectura apresentámos um exemplo detalhado de resolução do problema de cobertura. O processador combinatório aperfeiçoado foi modelado em C++ e alguns dos seus componentes foram implementados em hardware.

Finalmente, propusemos o modelo de computação que permite uma colaboração eficiente entre software e hardware reconfigurável, resolvendo parcialmente deste modo o problema de capacidade lógica limitada inerente a todas as implementações específicas para a instância.

6 Aceleração de algoritmos de solução de SAT

Sumário

Neste capítulo exploramos a possibilidade de aceleração do problema de satisfação booleana (SAT) com a ajuda de hardware reconfigurável. Começamos por descrever os algoritmos completos que são normalmente utilizados para a solução deste problema. É mostrado que SAT tem inúmeras aplicações práticas especialmente na área de projecto assistido por computador. Sendo assim, o desenvolvimento e implementação de algoritmos eficientes assumem actualmente grande importância.

Neste capítulo sugerimos uma técnica de desenvolvimento orientada à aplicação e destinada a acelerar a solução de várias instâncias de SAT formuladas sobre matrizes discretas. As arquitecturas propostas baseiam-se no modelo de interacção de software e hardware reconfigurável considerado no capítulo 5. A técnica adoptada elimina completamente a compilação de hardware específica para a instância e permite processar problemas que excedem a capacidade lógica da FPGA disponível. Explora-se também a possibilidade de implementação em hardware reconfigurável de estratégias avançadas de pesquisa que são amplamente utilizadas em todos os programas de SAT contemporâneos.

A parte de hardware é realizada numa FPGA XCV812E da família Virtex-EM da Xilinx que incorpora uma grande quantidade de blocos de memória embutidos que suportam directamente a abordagem proposta. As experiências efectuadas com as instâncias de teste disponíveis do conjunto do DIMACS (*Center for Discrete Mathematics & Theoretical Computer Science*) mostram que é possível atingir um desempenho bastante bom para algumas classes de problemas.

Concluímos o capítulo com uma análise de outras implementações de vários algoritmos de solução de SAT baseadas em hardware reconfigurável e propostas no âmbito da investigação académica. A análise é feita de acordo com as características principais de sistemas reconfiguráveis consideradas no capítulo 3.

6.1 Introdução

O problema de *satisfação booleana* (SAT) é muito conhecido na área de optimização combinatória e consiste em determinar se uma função booleana é satisfazível, i.e. se existe pelo menos uma atribuição de valores às variáveis que faz com que a função tome valor *1*. Normalmente, a função é especificada em CNF, isto é como a conjunção de um número de cláusulas, sendo cada cláusula composta por disjunção de uns literais. Cada literal corresponde a uma variável ou à sua negação.

Por exemplo, a função (6.1) contém três variáveis e quatro cláusulas e é satisfeita quando $x_1=x_3=1$ e $x_2=0$:

$$(x_1 \vee x_2)(\bar{x}_1 \vee x_3)(\bar{x}_2 \vee \bar{x}_3)(x_1 \vee \bar{x}_2) \tag{6.1}$$

Por outro lado a função seguinte é insatisfazível:

$$(x_1 \vee \bar{x}_2)(x_2)(\bar{x}_1)$$

O SAT é um caso específico dos problemas de satisfação de restrições (*Constraint Satisfaction Problems*) [Gu97] que consistem em determinar se um conjunto de restrições sobre variáveis discretas pode ser satisfeito (aqui cada restrição corresponde a uma cláusula).

6.2 Algoritmos

Os algoritmos de solução de SAT podem ser divididos em *completos* e *incompletos*. Neste trabalho só consideramos os algoritmos *completos* que são capazes de encontrar sempre a solução ou garantir que esta não existe. Um dos métodos eficientes de solução de SAT consiste na substituição recursiva da função original por uma ou duas sub-funções de tal maneira que a atribuição de valores às variáveis que satisfaz uma das sub-funções, satisfaça também a função original. Logo que é encontrada a solução para uma das sub-funções, a pesquisa termina. A *geração de resolventes* e a *decomposição* são os dois métodos mais conhecidos para criar sub-funções.

6.2.1 Geração de resolventes

A *resolvente* de duas cláusulas $c_1 = (x_1 \vee \ldots \vee v_j \vee \ldots \vee x_{l1})$ e $c_2 = (y_1 \vee \ldots \vee \bar{v}_j \vee \ldots \vee y_{l2})$ é a cláusula $(x_1 \vee \ldots \vee x_{l1} \vee y_1 \vee \ldots \vee y_{l2})$. Na resolução é escolhida uma variável v_j, j=1,...,n, e as resolventes que é possível criar com a ajuda da v_j são adicionadas à função original. Este processo continua até que seja criada uma cláusula vazia (isto significa que a função original é insatisfazível) ou até que seja impossível gerar mais resolventes (o que quer dizer que a solução existe). Por exemplo, ao resolver a função (6.1) com a ajuda da variável x_1 obtemos a função seguinte:

$$(x_1 \vee x_2)(\bar{x}_1 \vee x_3)(\bar{x}_2 \vee \bar{x}_3)(x_1 \vee \bar{x}_2)(x_2 \vee x_3)(\bar{x}_2 \vee x_3)$$

Na resolução, a escolha da variável v_j influencia significativamente a rapidez da obtenção da solução. Consideremos algumas das técnicas que permitem acelerar a resolução do problema.

6.2.1.1 Absorção

Quando os literais numa das cláusulas são um subconjunto dos literais de outra cláusula, então a cláusula menor absorve a maior. Consequentemente, a cláusula maior pode ser apagada da função não modificando deste modo o conjunto de soluções possíveis. Isto deve-se ao facto de que cada atribuição de valores às variáveis que satisfaz a cláusula menor, irá também satisfazer a cláusula maior. Esta técnica é de grande importância na resolução porque ajuda a eliminar cláusulas grandes facilitando deste modo a pesquisa.

6.2.1.2 Resolução de Davis-Putnam

Ao formar todas as resolventes de uma variável v_j é possível apagar todas as cláusulas que contêm v_j ou $\overline{v}_j$ [Davis60]. Sendo assim, primeiro são geradas todas as resolventes para a primeira variável, depois para a segunda, etc. No final fica apenas uma variável, tornando-se claro qual o valor que esta deve tomar para satisfazer a função. Este valor incorpora-se na sub-função obtida no passo anterior. Como resultado obtemos novamente a função composta por uma só variável. Este processo continua até que todas as variáveis recebam um valor. Por exemplo, para a função (6.1) para a variável x_1 obtemos a sub-função seguinte:

$$(\overline{x}_2 \vee \overline{x}_3)(x_2 \vee x_3)(\overline{x}_2 \vee x_3) \tag{6.2}$$

A seguir, para a variável x_2 é gerada a sub-função:

$$(x_3)$$

É evidente que x_3 tem de ser igual a *1* para satisfazer a função. Ao incorporar este valor na sub-função (6.2) recebemos:

$$(\overline{x}_2)$$

Obviamente x_2 tem que ser igual a *0*. Ao incluir este valor na função (6.1) recebemos a sub-função:

$$(x_1)$$

Torna-se claro que x_1 tem de receber valor *1* para satisfazer a função original.

6.2.1.3 Literais puros

Um literal diz-se *puro* quando todas as suas ocorrências na função são positivas ou, ao contrário, negativas. Obviamente, é impossível gerar resolventes para os literais puros. Contudo todas as cláusulas que contêm literais puros podem ser eliminadas da função diminuindo deste modo o tamanho do espaço de pesquisa e perdendo algumas soluções possíveis mas não retirando a possibilidade de encontrar pelo menos uma solução (caso esta exista). À variável correspondente é atribuído o valor *0* (se o literal puro é negativo) ou o valor *1* (quando o literal puro é positivo).

6.2.2 Decomposição

Na decomposição, é escolhida uma variável x_j, j=1,...,n, cujo valor ainda não está determinado, e são formadas duas sub-funções, na primeira das quais x_j =*1* e na segunda x_j =*0*. Como resultado, cada sub-função contém todas as cláusulas da função original para além daquelas que tomaram o

valor *1*, e todos os literais da função original para além daqueles que tomaram o valor *0*. Sendo assim, nenhuma das sub-funções contém a variável x_j e a função original só é solúvel quando uma das sub-funções o é. Consideremos um exemplo.

Dada a função (6.1) suponhamos que foi escolhida a variável x_1. Recebemos assim duas sub-funções a primeira das quais é satisfazível e a segunda não:

- para x_1=*1*: $(x_3)(\bar{x}_2 \vee \bar{x}_3)$;
- para x_1=*0*: $(x_2)(\bar{x}_2 \vee \bar{x}_3)(\bar{x}_2)$.

As sub-funções geradas na decomposição são analisadas uma por uma. Se a análise da primeira sub-função possibilitar o encontro da solução, o processo de pesquisa para. Caso contrário o algoritmo retrocede um passo atrás e tenta resolver a segunda sub-função. Por isso os algoritmos baseados na decomposição são designados por *algoritmos de retrocesso* (*backtracking algorithms*). Como descrevemos no capítulo 4, a ordem de análise de situações diferentes em algoritmos deste tipo é normalmente representada em forma de uma árvore de pesquisa. A raiz da árvore corresponde à função original, os vértices interiores coincidem com várias sub-funções geradas no processo de pesquisa, e as folhas representam aquelas sub-funções que podem ser processadas directamente, não recorrendo à decomposição. Todos os vértices da árvore de pesquisa são interligados por arcos etiquetados com as atribuições respectivas de valores às variáveis. Por exemplo, para a função (6.1), supondo que foi escolhida a ordem de variáveis de decomposição x_1-x_2-x_3, obteremos a árvore de pesquisa representada na fig. 6.1. Devido ao facto de só estarmos interessados numa solução, não é necessário percorrer a parte restante da árvore de pesquisa.

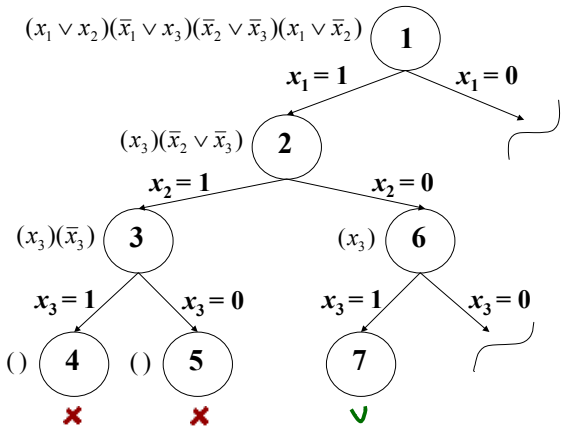


Fig. 6.1. Árvore de pesquisa para a função (6.1).

Os vários algoritmos de retrocesso distinguem-se pelo modo de selecção da variável à qual são atribuídos os dois valores (esta variável designa-se *variável de decisão*). Consideremos alguns dos mais conhecidos destes algoritmos [Gu97].

6.2.2.1 Retrocesso simples

Neste caso para seleccionar a variável de decisão escolhe-se a primeira variável x_j com o valor indeterminado. A seguir são geradas duas sub-funções numa das quais x_j =*1* e noutra x_j =*0*. Cada uma das sub-funções é processada de maneira recursiva. Se uma sub-função tiver uma cláusula vazia, o processamento desta sub-função deve ser suspenso, porque será impossível encontrar uma

solução nesta parte da árvore de pesquisa (situações 4 e 5 na fig. 6.1). Consequentemente, é necessário retroceder até ao último ponto de ramificação no qual o exame da variável de decisão não foi concluído, inverter o valor desta variável e continuar a percorrer a árvore de pesquisa. O facto de não existirem mais variáveis com valor indeterminado indica que a solução foi encontrada (situação 7 na fig. 6.1). A tentativa de retroceder para além da variável x_1 significa que todas as atribuições possíveis de valores às variáveis foram examinadas e que, consequentemente, a função original é insatisfazível.

6.2.2.2 Retrocesso de cláusula unitária

Caso uma cláusula c_i, i=1,...,m, só contenha uma variável, então é esta variável que é seleccionada como a de decisão e é-lhe atribuído um valor k tal que satisfaça a cláusula c_i (diz-se que a variável é *implicada* pelo valor k). As cláusulas com estas características são chamadas *cláusulas unitárias*. Quando não existem cláusulas unitárias, é seleccionada a primeira variável com o valor indeterminado (como no retrocesso simples). Esta técnica permite acelerar significativamente a pesquisa porque tenta satisfazer primeiro aquelas cláusulas que são mais "difíceis".

6.2.2.3 Retrocesso pela sequência de cláusulas

Neste caso para seleccionar a variável de decisão escolhe-se a primeira cláusula e dentro desta é eleita a primeira variável. Como resultado, o algoritmo às vezes evita a atribuição de valores a todas as variáveis, e a solução fica apresentada de uma forma compacta. Isto deve-se ao facto de serem atribuídos valores apenas àquelas variáveis que influenciam a satisfação de cláusulas, todas as outras variáveis ficam com o valor *'-'* (*don't care*).

6.2.2.4 Retrocesso de cláusula mais curta

Este algoritmo é idêntico ao anterior mas as cláusulas são escolhidas não pela sua ordem relativa na função, mas sim pela quantidade de literais que estas contêm (sempre se selecciona a cláusula mais curta). Como resultado, primeiro são satisfeitas as cláusulas mais "difíceis".

6.2.3 Algoritmo de Davis-Putnam

O algoritmo de Davis-Putnam (DP) [Davis62] considera-se clássico pois foi proposto nos anos sessenta e desde aí tornou-se muito utilizado. Esta versão do algoritmo é também conhecida sob o nome de Davis-Putnam-Loveland e baseia-se na decomposição (enquanto a versão original [Davis60] se baseia na geração de resolventes). O algoritmo começa com os valores de todas as variáveis indeterminados. Inicialmente, selecciona-se a primeira variável x_j, j=1,...,n, com o valor indeterminado e x_j torna-se igual a *0*. A seguir são descobertas todas as implicações directas e transitivas em outras variáveis. Quando uma variável recebe um valor (via decisão ou implicação), algumas cláusulas ficam satisfeitas e outras continuam com o valor indeterminado. Todas as cláusulas satisfeitas bem como todos os literais com valor *0* são removidos da função. Caso alguma variável seja implicada a dois valores opostos, estamos perante um *conflito*. Se não surgir conflito nenhum, selecciona-se novamente uma variável com o valor indeterminado e o processo continua (este chama-se *pesquisa para diante*). Caso apareça um conflito, o algoritmo retrocede até à variável de decisão mais recente e inverte o seu valor. Se novamente surgir um conflito, a variável

de decisão recebe o valor indeterminado e o algoritmo retrocede mais uma vez. A tentativa de retroceder para além da primeira variável de decisão indica que todas as atribuições possíveis de valores às variáveis foram esgotadas e que, consequentemente, a função é insatisfazível. A solução é encontrada quando todas as variáveis recebem um valor sem conflitos.

Os algoritmos recentes implementados em software exploram técnicas avançadas que permitem identificar e evitar a análise daquelas zonas do espaço de pesquisa que não contêm nenhuma solução. Estas técnicas incluem o retrocesso *não cronológico* [Silva99], geração e adição de cláusulas responsáveis por conflitos, etc. [Moskewicz01, Goldberg02b]. É de salientar contudo que estas técnicas não obtiveram uma grande aceitação no âmbito da computação reconfigurável por causa da sua implementação difícil em hardware.

6.3 Aplicações práticas

O SAT tem muitas aplicações práticas em síntese lógica [Brayton84, Gu95], colocação e encaminhamento de circuitos electrónicos [Devadas89, Wood98], teste de circuitos [Larrabee92, Stephan96], planeamento [Crawford94, Feldman90], telecomunicações [Wang96], matemática [Matula72], robótica [Chin86], processamento de texto [Huang93], arquitectura de computadores [Rao83], etc. [Gu96, Gu97]. Por conseguinte, o desenvolvimento de métodos eficientes de solução de SAT, bem como a sua implementação, tornou-se uma área vital de investigação. Consideremos duas aplicações que podem ser formuladas como instâncias de SAT.

6.3.1 Planeamento

O problema de planeamento surge em muitas aplicações práticas. Suponhamos que existe um conjunto de investigadores que devem resolver uma série de tarefas. Cada cientista só pode lidar com um subconjunto de tarefas. Obviamente, o objectivo é maximizar a eficiência do trabalho dos investigadores. Suponhamos que a execução de cada tarefa i ocupa tempo p_i. A solução do problema é um plano que define o tempo de início s_i de cada tarefa. São normalmente aplicadas as seguintes restrições [Crawford94]:

- Restrições de sequência $i \rightarrow j$ definem que a tarefa i deve ser concluída antes do início da tarefa j. Esta restrição pode ser reformulada como $s_i + p_i \leq s_j$, i.e. o tempo de início da tarefa i mais o tempo da sua execução devem ser iguais ou menores que o tempo do início da tarefa j.
- Restrições de recursos $c_{i,j}$ indicam que as tarefas i e j estão em conflito, i.e. ambas requerem o mesmo recurso (o mesmo investigador). Portanto as tarefas i e j não podem ser executadas ao mesmo tempo. A outra forma de especificar esta restrição é $(s_i + p_i \leq s_j) \vee (s_j + p_j \leq s_i)$, i.e. a tarefa i deve ser concluída antes do início da tarefa j, ou, ao contrário, a tarefa j deve ser concluída antes do início da tarefa i.
- Restrições de início r_i definem o tempo depois do qual a tarefa i pode ser iniciada, i.e. $s_i > r_i$.
- Restrições de prazo d_i especificam o momento até ao qual a execução da tarefa i deve ser finalizada, i.e. $s_i + p_i \leq d_i$.

Existe uma transformação óbvia do problema de planeamento em SAT [Crawford94]. Para tal criam-se as variáveis s_{it} que representam os tempos t de início de cada tarefa i ($s_{it}=1$ significa que a tarefa i começa em tempo t). A seguir são criadas as cláusulas que representam todas as desigualdades necessárias. É de notar que o espaço de pesquisa resultante será neste caso demasiado grande, por isso só são utilizadas as transformações que permitem ordenar as tarefas que estão em conflito porque os tempos de início de cada tarefa podem ser determinados posteriormente [Crawford94]. Não vamos abordar estas transformações aqui pois são bastante complexas. Contudo, é importante realçar que a função resultante embora seja grande, é facilmente resolvida. Isto acontece porque esta é sub-restringida, i.e. existem poucas restrições, portanto há muitas soluções e é fácil encontrar uma.

6.3.2 Teste de circuitos combinatórios

O SAT pode também ser utilizado para gerar padrões de teste para descobrir falhas permanentes (*stuck-at faults*) em circuitos combinatórios [Larrabee92]. Para detectar falhas num circuito são aplicados a este valores de entrada que produzem valores de saída diferentes dos do circuito sem falhas. Neste caso, para descobrir os valores de entrada adequados, é construída uma função que defina um conjunto de padrões de teste capazes de determinar uma falha e, de seguida, são encontrados os valores das variáveis que satisfazem esta função.

Para tal primeiro a estrutura do circuito combinatório é descrita com a ajuda de uma função booleana em CNF. Por exemplo, para a porta lógica *OR* com as entradas x e y e a saída z temos:

$$z = x \vee y$$

Tendo em conta que a função $p=q$ é equivalente a $(p \rightarrow q)(q \rightarrow p)$, obtemos:

$$(z \rightarrow (x \vee y)) \; ((x \vee y) \rightarrow z)$$

Como a função $p \rightarrow q$ é equivalente a $\overline{p} \vee q$ pode-se escrever:

$$(\overline{z} \vee x \vee y)(\overline{(x \vee y)} \vee z)$$

Finalmente, aplicando os leis de De Morgan obtemos:

$$(\overline{z} \vee x \vee y)(z \vee \overline{x})(z \vee \overline{y})$$

Da mesma maneira pode-se determinar as funções em CNF para todos os tipos de portas lógicas e, a seguir, construir a função do circuito inteiro. Por exemplo, para o circuito da fig. 6.2a obtemos a função seguinte:

$$(\overline{f} \vee \overline{c} \vee \overline{e})(f \vee c)(f \vee e)(\overline{c} \vee a \vee b)(c \vee \overline{a})(c \vee \overline{b})(d \vee e)(\overline{d} \vee \overline{e}) \qquad (6.3)$$

Na fig. 6.2b está representada a versão defeituosa do mesmo circuito (a linha *e* está *stuck-at 1*). A função do circuito com falha constrói-se da mesma maneira, mas as linhas que ficam entre a falha e a saída devem ser renomeadas (ver fig. 6.2b):

$$(\overline{f'} \vee \overline{c} \vee \overline{e'})(f' \vee c)(f' \vee e')(e')(\overline{c} \vee a \vee b)(c \vee \overline{a})(c \vee \overline{b}) \qquad (6.4)$$

Para poder detectar esta falha será necessário encontrar os valores de entrada que produzem valores de saída diferentes para os dois circuitos considerados (um com falha e outro sem falha). Para tal construímos a conjunção das funções (6.3) e (6.4) e adicionamos a disjunção exclusiva das saídas de ambos os circuitos (ver fig. 6.2c):

$$(\bar{f} \vee \bar{c} \vee \bar{e})(f \vee c)(f \vee e)(\bar{c} \vee a \vee b)(c \vee \bar{a})(c \vee \bar{b}) \wedge$$
$$\wedge (d \vee e)(\bar{d} \vee \bar{e})(\bar{f}' \vee \bar{c} \vee \bar{e}')(f' \vee c)(f' \vee e')(e') \wedge$$
$$\wedge (\bar{f} \vee f' \vee g)(f \vee \bar{f}' \vee g)(\bar{f} \vee \bar{f}' \vee \bar{g})(f \vee f' \vee \bar{g})$$

Obviamente, para poder descobrir a falha, a saída *g* deve ser igual a *1*. Assim obtemos a função:

$$(\bar{f} \vee \bar{c} \vee \bar{e})(f \vee c)(f \vee e)(\bar{c} \vee a \vee b)(c \vee \bar{a})(c \vee \bar{b}) \wedge$$
$$\wedge (\bar{f}' \vee \bar{c} \vee \bar{e}')(f' \vee c)(f' \vee e')(e')(d \vee e)(\bar{d} \vee \bar{e}) \wedge \qquad (6.5)$$
$$\wedge (\bar{f} \vee \bar{f}')(f \vee f')$$

A atribuição de valores às variáveis que satisfaz a função (6.5) define os padrões de teste necessários para detectar a falha considerada (por exemplo, *a=b=d =1*).

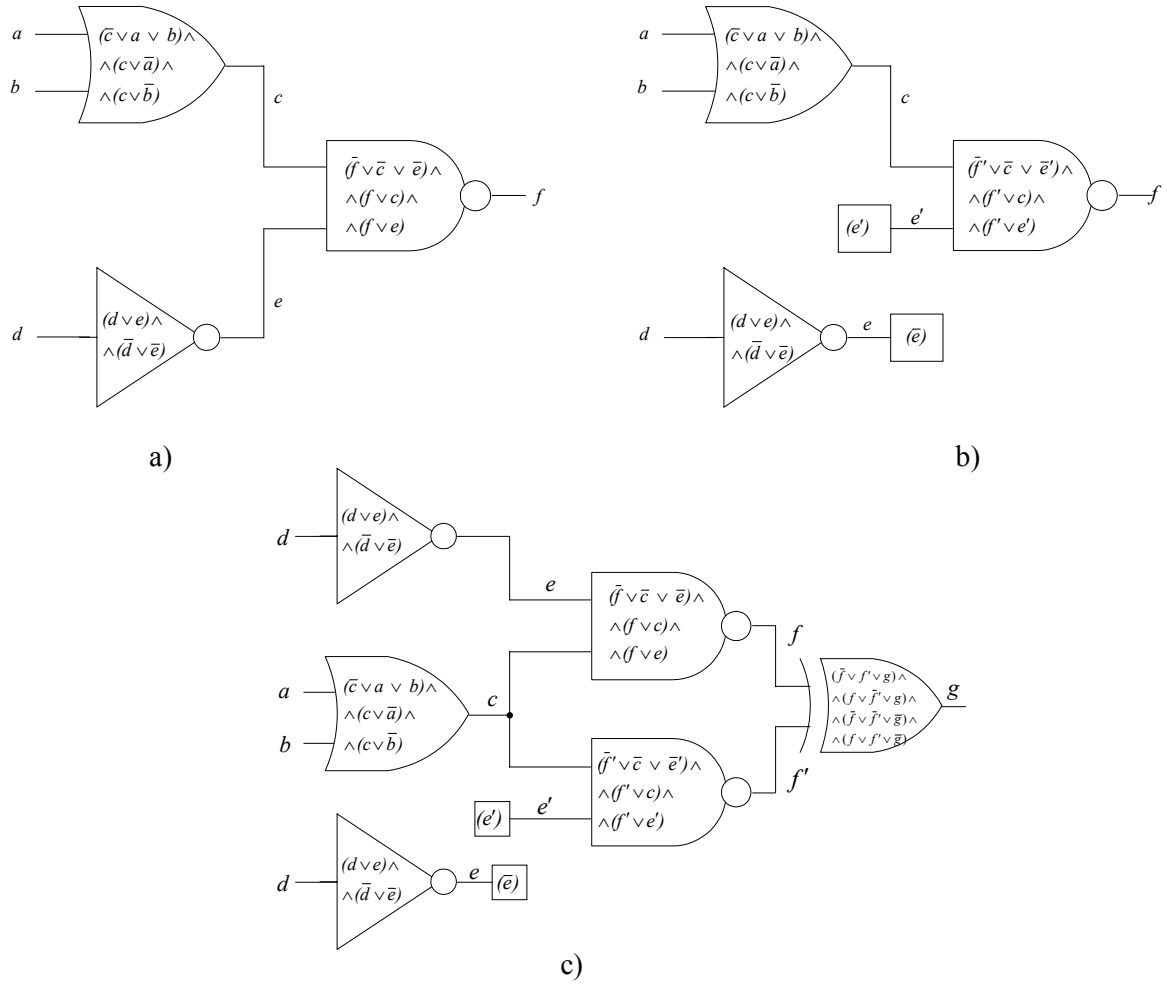


Fig. 6.2. Circuito combinatório correcto (a); Circuito combinatório com falha (b); Combinação dos circuitos defeituoso e correcto (c).

6.4 Implementação em hardware reconfigurável

O problema de SAT pode ser especificado sobre vários modelos matemáticos tais como funções booleanas e matrizes discretas. Para a implementação em hardware reconfigurável foi escolhida a representação em matrizes porque estas são de bastante fácil processamento em FPGA. Formulemos o problema de SAT sobre uma matriz ternária **U**. Para isso vamos fazer corresponder a cada cláusula c_i, i=1,...,m, uma linha da matriz $\mathbf{u}_i$ e a cada variável x_j, j=1,...,n, uma coluna da matriz $\mathbf{u}_j$. Se a variável x_j entra na cláusula c_i então o elemento respectivo da matriz u_{ij} é igual a *1*, se a variável x_j entra na cláusula c_i negada então o elemento respectivo da matriz u_{ij} é igual a *0*, e se a variável x_j não entra na cláusula c_i então o elemento respectivo da matriz u_{ij} é igual a *'-'* (*don't care*). Por exemplo, a função seguinte:

$$(x_1 \vee \bar{x}_3)(\bar{x}_1 \vee x_2)(\bar{x}_2 \vee \bar{x}_3)(x_1 \vee x_2) \tag{6.6}$$

pode ser representada com a matriz **U**:

$$\mathbf{U} = \begin{array}{c} \begin{matrix} \mathbf{x_1} & \mathbf{x_2} & \mathbf{x_3} \end{matrix} \\ \begin{bmatrix} \mathbf{1} & - & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & - \\ - & \mathbf{0} & \mathbf{0} \\ \mathbf{1} & \mathbf{1} & - \end{bmatrix} \end{array} \begin{matrix} \mathbf{c_1} \\ \mathbf{c_2} \\ \mathbf{c_3} \\ \mathbf{c_4} \end{matrix}$$

É de notar que resolver um problema de SAT é equivalente ao encontrar um vector ternário **v** que seja ortogonal a cada linha da matriz **U** respectiva. Se for impossível arranjar tal vector então a função correspondente é insatisfazível. Por outro lado caso encontremos o vector **v**, devemos invertê-lo. Os zeros e uns no vector invertido apontarão aquelas variáveis que devem ser iguais a *0* e *1* respectivamente para satisfazer a função. Para a matriz **U** apresentada acima a solução é o vector **v** = [-01]. Sendo assim, $\bar{\mathbf{v}}$ = [-10], i.e. x_2 =*1* e x_3 =*0*. É fácil verificar que a atribuição destes valores às variáveis satisfaz a função (6.6).

6.4.1 Algoritmo de retrocesso

Para encontrar o vector ortogonal a cada linha de uma matriz ternária adaptámos o algoritmo proposto em [Zakrevski81] apresentando-o em forma de uma árvore de pesquisa. Neste caso a cada vértice da árvore de pesquisa corresponde um vector ternário **v** (i.e. a uma solução parcial) e uma matriz **U′** composta por vários menores da matriz **U**. Inicialmente, o vector **v** está totalmente indeterminado, i.e. **v**=[-…-], e **U′**=**U**. A transição de um vértice da árvore de pesquisa para o seguinte efectua-se se reduzirmos a matriz **U′** ou atribuirmos valor *0* ou *1* a uma componente do vector **v**. Na fig. 6.3 está representado o fluxograma do algoritmo utilizado. O fluxograma mostra que durante a pesquisa se aplicam vários métodos de redução, depois, quando a redução se torna impossível, efectua-se a decomposição da situação corrente (o que corresponde à ramificação da árvore).

A seguir, aplicam-se novamente os métodos de redução e o algoritmo continua até encontrar a solução ou concluir que esta não existe. Descrevemos os métodos de redução envolvidos, mostrando como estes se relacionam com os descritos na secção 6.2:

- *Método 1*. Na matriz **U**′ apagam-se todas as colunas que são totalmente indeterminadas, i.e. não contêm *0*s nem *1*s. Este método permite detectar variáveis *insignificantes*, i.e. as variáveis que não influenciam de modo algum a satisfação da função e por isso podem ser excluídas de consideração futura.
- *Método 2*. Na matriz **U**′ apagam-se todas as linhas que são ortogonais ao vector **v** corrente. O método é equivalente à eliminação de cláusulas satisfeitas da função.
- *Método 3*. Na matriz **U**′ apagam-se todas as colunas que correspondem às componentes determinadas do vector **v**. Esta operação elimina de consideração futura as variáveis já atribuídas.
- *Método 4*. Caso na matriz **U**′ exista uma linha que tem uma só componente com o valor *0* ou *1* e todas as suas outras componentes iguais a *'-'*, então ao elemento correspondente do vector **v** atribui-se o valor inverso. A linha com esta característica representa uma cláusula unitária (ver secção 6.2.2.2).
- *Método 5*. Caso na matriz **U**′ exista uma coluna que não contém valor *0* (ou *1*), então este valor é atribuído à componente correspondente do vector **v**. Esta coluna representa um literal puro (ver secção 6.2.1.3).

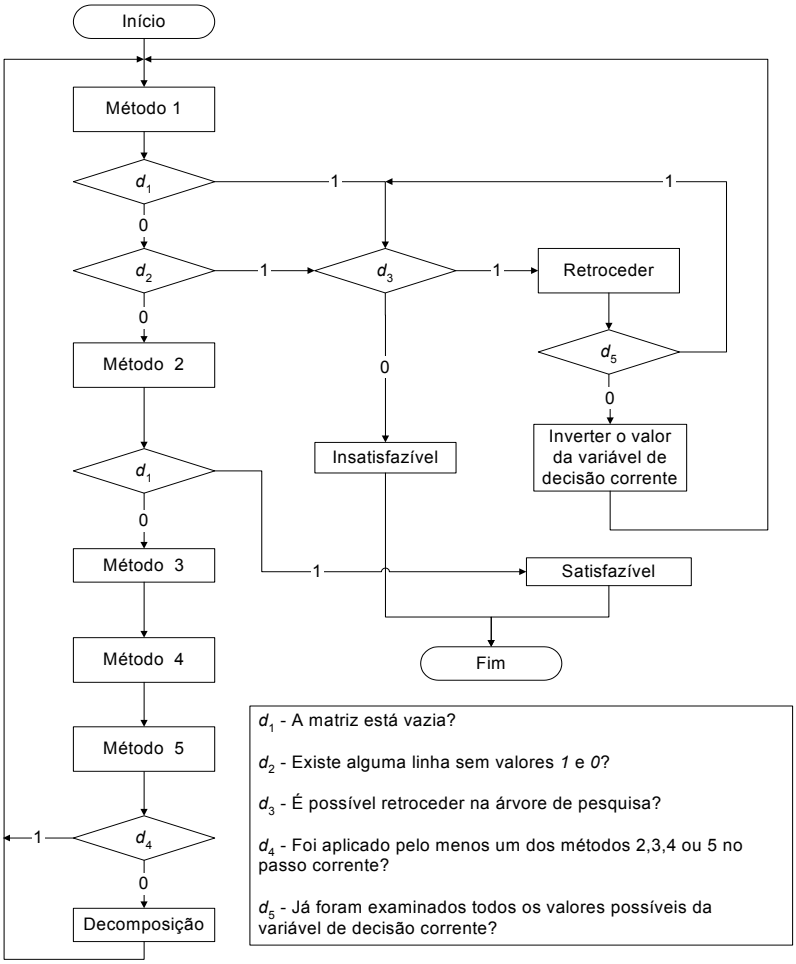


Fig. 6.3. Algoritmo de solução de SAT formulado sobre uma matriz discreta.

Na decomposição de uma situação efectua-se exame sequencial de valores *1* e *0* de alguma componente do vector **v**. Para tal escolhemos a componente que corresponda à coluna mais determinada da matriz **U**′, i.e. à coluna que tenha o número mínimo de valores ′-′. Sendo assim efectua-se a *selecção dinâmica* da variável de decisão seguinte. Para cada variável de decisão escolhe-se tal valor que faz com que o número máximo de linhas da **U**′ fiquem ortogonais ao vector **v**. Deste modo tenta-se satisfazer o número máximo possível de cláusulas. A selecção diz-se *dinâmica* porque inicialmente é desconhecida a ordem em que serão consideradas as variáveis, sendo esta só determinada em *run-time*. No caso da selecção *estática* a ordem das variáveis é estabelecida antes da execução do algoritmo (com um passo de pré-processamento).

Caso, depois de apagar uma linha, a matriz **U**′ se torne vazia então o valor corrente do vector **v** representa a solução. Por outro lado se a matriz **U**′ ficar vazia depois de eliminar uma coluna ou se **U**′ contiver uma linha sem valores *1* e *0*, então é impossível encontrar uma solução neste ramo da árvore de pesquisa. Esta situação corresponde à existência de uma cláusula vazia (sem literais) na função. Portanto é necessário retroceder até ao ponto mais recente de ramificação no qual o exame da variável de decisão não foi concluído, inverter o valor desta variável e continuar a percorrer a árvore de pesquisa, i.e. implementámos o retrocesso *cronológico*. Caso ao percorrer toda a árvore não seja possível encontrar o vector **v** ortogonal a cada linha da matriz **U** então a função booleana correspondente é insatisfazível.

Consideremos um exemplo. Vamos verificar se a função booleana seguinte é satisfazível:

$$\begin{aligned} f(x_1,\ldots,x_8) = {} & (\bar{x}_1 \vee \bar{x}_2)(\bar{x}_2 \vee \bar{x}_3)(x_1 \vee x_2 \vee x_3) \wedge \\ & \wedge (x_2 \vee \bar{x}_4 \vee \bar{x}_5)(x_2 \vee \bar{x}_5 \vee \bar{x}_6)(x_2 \vee x_4 \vee x_5 \vee x_6) \wedge \\ & \wedge (\bar{x}_7 \vee \bar{x}_8)(\bar{x}_2 \vee \bar{x}_7 \vee x_8)(x_7 \vee x_8)(x_1 \vee x_7 \vee \bar{x}_8) \end{aligned} \tag{6.7}$$

A função pode ser transformada na seguinte matriz **U**:

$$\mathbf{U} = \begin{array}{c} \begin{matrix} \mathbf{x_1} & \mathbf{x_2} & \mathbf{x_3} & \mathbf{x_4} & \mathbf{x_5} & \mathbf{x_6} & \mathbf{x_7} & \mathbf{x_8} \end{matrix} \\ \begin{bmatrix} \mathbf{0} & \mathbf{0} & - & - & - & - & - & - \\ - & \mathbf{0} & \mathbf{0} & - & - & - & - & - \\ \mathbf{1} & \mathbf{1} & \mathbf{1} & - & - & - & - & - \\ - & \mathbf{1} & - & \mathbf{0} & \mathbf{0} & - & - & - \\ - & \mathbf{1} & - & - & \mathbf{0} & \mathbf{0} & - & - \\ - & \mathbf{1} & - & \mathbf{1} & \mathbf{1} & \mathbf{1} & - & - \\ - & - & - & - & - & - & \mathbf{0} & \mathbf{0} \\ - & \mathbf{0} & - & - & - & - & \mathbf{0} & \mathbf{1} \\ - & - & - & - & - & - & \mathbf{1} & \mathbf{1} \\ \mathbf{1} & - & - & - & - & - & \mathbf{1} & \mathbf{0} \end{bmatrix} \begin{matrix} \mathbf{1} \\ \mathbf{2} \\ \mathbf{3} \\ \mathbf{4} \\ \mathbf{5} \\ \mathbf{6} \\ \mathbf{7} \\ \mathbf{8} \\ \mathbf{9} \\ \mathbf{10} \end{matrix} \end{array}$$

A aplicação do algoritmo considerado à matriz **U** pode ser representada com a árvore de pesquisa da fig. 6.4. A numeração de vértices reflecte a ordem por que são examinadas as situações intermédias. A fig. 6.4 mostra também a evolução do vector **v** e das matrizes intermédias **U**′ que são construídas no processo de pesquisa, e os números dos métodos que são aplicados em vários passos do algoritmo. É fácil verificar que o vector **v** encontrado é ortogonal a cada linha da matriz **U**. Sendo assim a função (6.7) é satisfazível com a seguinte atribuição de valores às variáveis: $x_1=1$, $x_2=0$, $x_3=1$, $x_4=1$, $x_5=0$, $x_6=1$, $x_7=0$ e $x_8=1$.

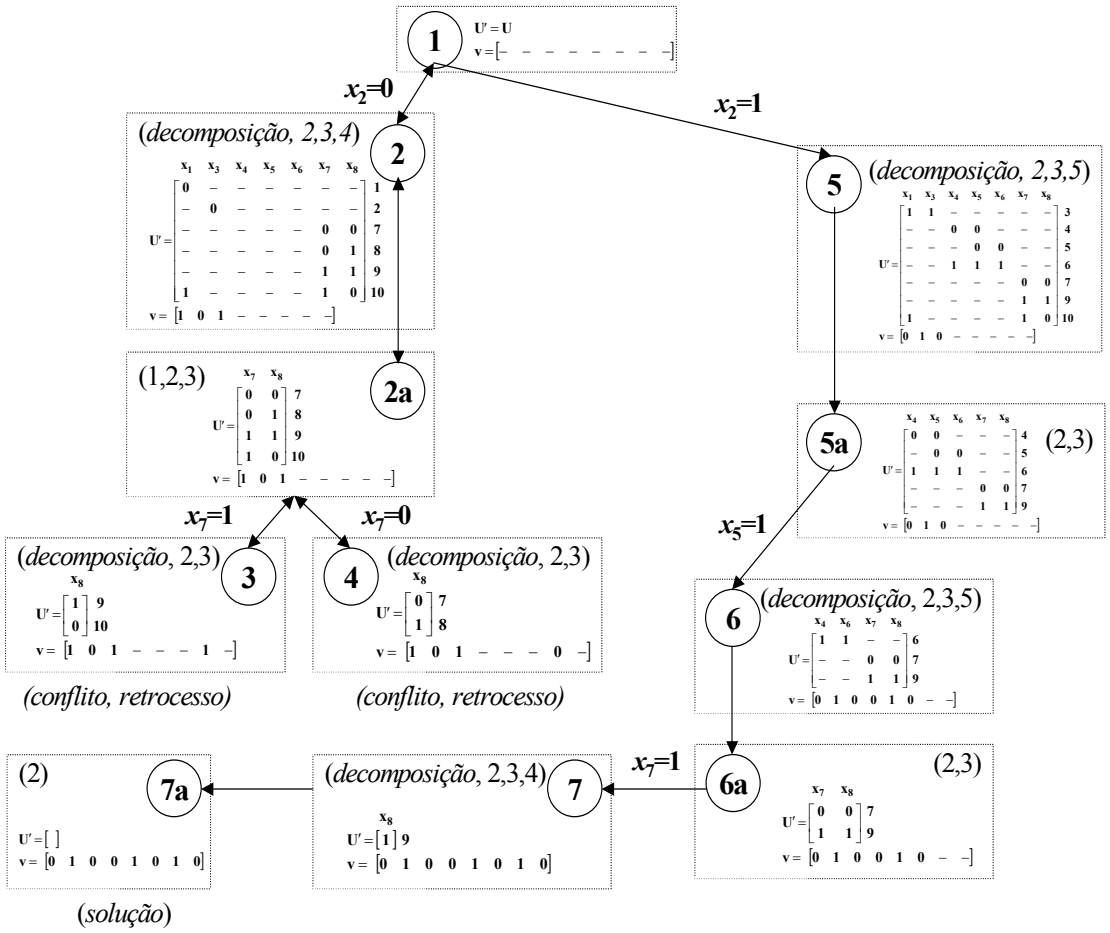


Fig. 6.4. Árvore de pesquisa para encontrar o vector **v** ortogonal a cada linha da matriz **U**.

6.4.2 Algoritmo baseado em intervalos

Suponhamos que cada linha da matriz **U** especifica um intervalo no espaço booleano de variáveis $x_1,\ldots,x_n$. Então a união de todos os intervalos (correspondentes a todas as linhas de **U**) define uma região no espaço booleano onde não há nenhuma solução. A região restante do espaço booleano contém o conjunto de todas as soluções do problema. Devido ao facto de cada vector especificar um intervalo no espaço booleano, vamos usar os termos *vector* e *intervalo* sem distinção.

O algoritmo proposto baseado em intervalos analisa sucessivamente todas as linhas da matriz **U** construindo deste modo a solução. Inicialmente, um *n*-cubo especifica um conjunto de soluções. Para encontrar todos os vectores ortogonais à primeira linha $\mathbf{u}_1$ de **U**, esta linha é subtraída do *n*-cubo. A seguir, do conjunto resultante de intervalos *S* seleccionam-se aqueles que não são ortogonais à linha seguinte ($\mathbf{u}_2$) e destes é subtraída esta linha. Os intervalos que são ortogonais à linha $\mathbf{u}_2$ permanecem no conjunto *S* sem alterações. O processo continua até que o conjunto de intervalos *S* fique vazio (o que significa que a função é insatisfazível) ou até que todas as linhas da matriz **U** sejam consideradas (neste caso o conjunto de intervalos *S* contém todas as soluções).

Se num passo do algoritmo, o conjunto *S* for composto por vectores $\mathbf{s}_1,\ldots,\mathbf{s}_K$, que não são ortogonais à linha $\mathbf{u}_i$, $i=1,\ldots,m$, a operação de subtracção da linha $\mathbf{u}_i$ é definida de maneira seguinte:

$$\mathbf{S}-\mathbf{u}_i=\begin{bmatrix}\mathbf{s}_1-\mathbf{u}_i\\ \mathbf{s}_2-\mathbf{u}_i\\ \dots\\ \mathbf{s}_K-\mathbf{u}_i\end{bmatrix}$$

A operação de subtracção para dois vectores $\mathbf{s}_k=[s_{k1}\ s_{k2}\ \dots\ s_{kn}]$, $k=1,\dots,K$, e $\mathbf{u}_i=[u_{i1}\ u_{i2}\ \dots\ u_{in}]$, $i=1,\dots,m$, é definida como:

$$\begin{bmatrix}s_{k1}-u_{i1} & s_{k2} & \dots & s_{kn}\\ s_{k1} & s_{k2}-u_{i2} & \dots & s_{kn}\\ \dots & \dots & \dots & \dots\\ s_{k1} & s_{k2} & \dots & s_{kn}-u_{in}\end{bmatrix}$$

Finalmente, o resultado de subtracção para componentes individuais s_{kj} e u_{ij}, $j=1,\dots,n$, é igual a:

- *1* - se $s_{kj}=$ '-' e $u_{ij}=0$;
- *0* - se $s_{kj}=$ '-' e $u_{ij}=1$;
- ∅ - em todos os casos restantes.

É de notar que se qualquer componente dum vector for igual a ∅, este vector é excluído do conjunto *S*.

Consideremos um exemplo. Suponhamos que é dada a matriz **U** representada na fig. 6.5a. O primeiro passo é subtrair a linha $\mathbf{u}_1$ do *3*-cubo. O conjunto *S* resultante especifica que a solução fica nos planos inferior e direito do *3*-cubo (ver fig. 6.5b). Depois de subtrair a última linha $\mathbf{u}_2$ do conjunto *S* recebemos as soluções representadas na fig. 6.5c.

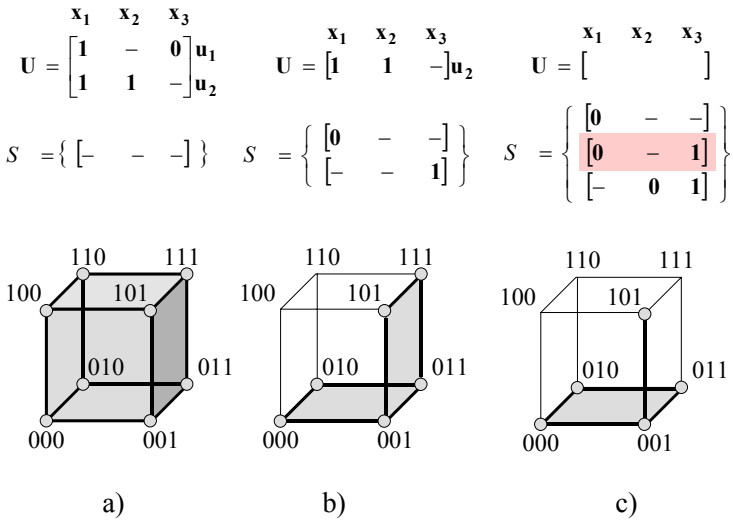


Fig. 6.5. Aplicação do algoritmo baseado em intervalos.

É importante que o algoritmo permite encontrar todas as soluções do problema apresentando estas de uma forma compacta. O encontro de todas as soluções possíveis é exigido em algumas aplicações práticas [Suyama01]. Contudo, o algoritmo possui uma desvantagem distintiva: o tamanho do conjunto *S* aumenta muito rapidamente. Por isso são aplicados métodos especiais capazes de limitar o ritmo deste aumento.

Consideremos estes métodos. Para tal é necessário definir a operação de absorção (ver secção 6.2.1.1). O vector **a** absorve o vector **b** se para cada *j*, *j*=1,..,*n*, uma das expressões seguintes é válida: a_j = '-' ou $a_j = b_j$. Sendo assim se um vector do conjunto *S* absorve um intervalo recém-gerado, este intervalo pode ser eliminado. Se ao contrário, um intervalo recém-gerado absorve alguns vectores do conjunto *S*, então estes vectores são excluídos de *S*. Por exemplo, o intervalo [0 – –] da fig. 6.5c absorve o intervalo [0 – 1], portanto este último pode ser eliminado do conjunto *S*, resultando na solução composta só por dois vectores. Finalmente, se existir uma linha na matriz **U** que absorve um intervalo de *S*, este intervalo pode ser apagado. É de notar contudo que mesmo a aplicação dos métodos propostos não torna o algoritmo aceitável para a solução de problemas práticos.

6.4.3 Algoritmo híbrido

Como mencionamos na secção anterior, o algoritmo baseado em intervalos é pouco eficiente. Contudo seria interessante explorar a possibilidade de combinação dos dois algoritmos descritos. De facto esta abordagem produziu resultados bastante promissores por isso descrevemo-la em mais detalhe.

Suponhamos que o tamanho máximo do conjunto *S* é limitado pelo valor s_{max}. Neste caso o algoritmo baseado em intervalos só poderá processar m' linhas da matriz **U** ($m' \leq m$). Se s_{max} for um número suficientemente pequeno, então todas as computações necessárias seriam executadas muito rapidamente (incluindo a detecção de vectores absorvidos e absorventes porque no caso de s_{max} grande esta operação torna-se muito dispendiosa). Se todas as linhas de **U** forem processadas (o que é impossível para problemas práticos) a solução é encontrada e a pesquisa termina. Caso contrário, os elementos do conjunto *S* formam a *lista de favorecidos* que contém apenas as atribuições *permitidas* de valores às variáveis.

A seguir o algoritmo de retrocesso é aplicado e em cada vértice da árvore de pesquisa verifica-se se o vector **v** corrente é compatível com a *lista de favorecidos*. O vector **v** é compatível se este é incluído pelo menos num intervalo $\mathbf{s}_k$, k=1,…,s_{max}, i.e. se $\mathbf{s}_k$ absorve **v**. Caso isto seja verdade o processo de pesquisa continua. Caso contrário, torna-se claro que o ramo corrente da árvore de pesquisa não pode levar à solução, por isso o algoritmo deve retroceder. Como resultado, o *retrocesso antecipado* é efectuado. Embora a técnica proposta seja bastante simples, esta tem uma grande influência sobre o tempo de execução pois permite reduzir o número de vértices percorridos da árvore de pesquisa.

6.4.4 Implementação

É de salientar que o tempo de compilação do circuito de hardware limita significativamente a gama de problemas para os quais a utilização de FPGA pode ser considerada razoável em comparação com a solução baseada em software. Tendo isto em atenção tentámos criar um circuito universal, ou um *hardware template*, que possua uma estrutura predefinida que pode ser reutilizada para várias instâncias do problema. Como resultado foi proposta a arquitectura representada na fig. 6.6. Implementámos duas versões do circuito. A primeira executa o algoritmo de retrocesso e a segunda corresponde ao algoritmo híbrido.

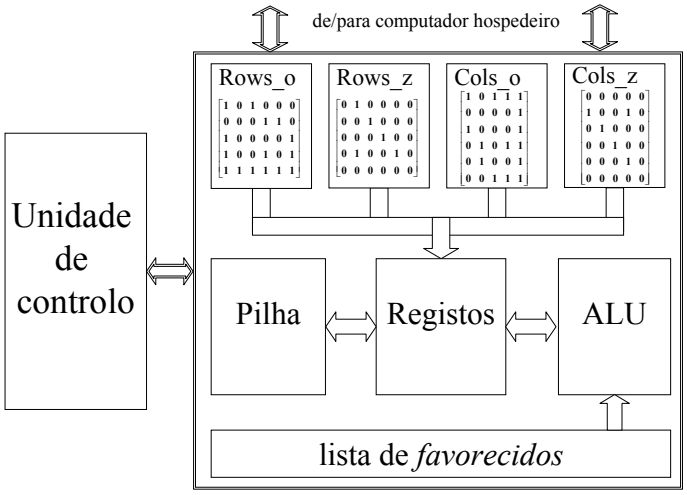


Fig. 6.6. Arquitectura do circuito.

Os componentes básicos da arquitectura proposta representados na fig. 6.6 são os seguintes. A *Unidade de controlo* central executa um dos algoritmos descritos acima. A unidade de controlo foi especificada com a ajuda do VHDL e implementada como uma máquina de estados finitos.

O bloco *Registos* consiste dos componentes seguintes:

- *cur_row_addr* – é um registo de $\log_2 m$ bits que indica o endereço da linha activa da matriz **U**.
- *cur_col_addr* – é um registo de $\log_2 n$ bits que guarda o endereço da coluna activa da matriz **U**.
- *cur_v* – é composto por dois registos de *n* bits que guardam o valor corrente do vector **v**. O primeiro destes registos (*cur_v_o*) contém *1*s só em posições em que o vector **v** é igual a *1*, enquanto o segundo registo (*cur_v_z*) contém *1*s em posições onde o vector **v** é igual a *0*. Como resultado, *cur_v_o*[*i*] =*1* só se **v**[*i*]=*1*, e *cur_v_z*[*i*] =*1* apenas se **v**[*i*]=*0*, *i*=1,…,*n*.
- *del_rows* – é um registo de *m* bits que indica (por *0*s) as linhas eliminadas da matriz **U**.
- *del_cols* - é um registo de *n* bits que indica (por *0*s) as colunas apagadas da matriz **U**.
- *des_var* – é um registo de $\log_2 n$ bits que guarda o número da variável de decisão corrente.
- *test_val* – é um registo de 2 bits que mantém o valor atribuído à variável de decisão corrente e indica se já foram testados ambos os valores.
- *aux_regs* – uma série de registos auxiliares.

Os dados da matriz são guardados em quatro blocos de memória (*Rows_o*, *Rows_z*, *Cols_o*, *Col_z* na fig. 6.6). Como os nomes indicam, os dois primeiros blocos contêm **U** e os dois últimos mantêm $\mathbf{U}^T$ (a transposta da **U**). Cada elemento u_{ij}, *i*=1,…,*m*, *j*=1,…,*n*, da matriz **U** é codificado da maneira seguinte:

- se $u_{ij} = 1$, então *Rows_o*[*i*][*j*] = *1*, *Rows_z*[*i*][*j*] = *0*, *Cols_o*[*j*][*i*] = *1*, e *Cols_z*[*j*][*i*] = *0*.
- se $u_{ij} = 0$, então *Rows_o*[*i*][*j*] = *0*, *Rows_z*[*i*][*j*] = *1*, *Cols_o*[*j*][*i*] = *0*, e *Cols_z*[*j*][*i*] = *1*.
- se u_{ij} = '-', então *Rows_o*[*i*][*j*] = *0*, *Rows_z*[*i*][*j*] = *0*, *Cols_o*[*j*][*i*] = *0*, e *Cols_z*[*j*][*i*] = *0*.

Obviamente, os quatro blocos de memória requerem mais recursos de hardware mas isto permite aceder a cada linha e coluna de **U** num só ciclo de relógio. As matrizes não são modificadas durante a execução do algoritmo. Todas as alterações possíveis (eliminação de linhas e colunas) são armazenadas em registos correspondentes. Sendo assim, não é necessário guardar as matrizes intermédias na pilha. As dimensões máximas da matriz **U** são fixas: $m_{max} \times n_{max}$. Para o caso de necessidade de processamento de uma matriz com dimensões menores existem dois registos especiais que guardam as dimensões actuais da matriz $m_{act} \times n_{act}$. De acordo com estes valores a unidade de controlo só vai forçar processamento da área adequada da matriz, conforme representado na fig. 6.7.

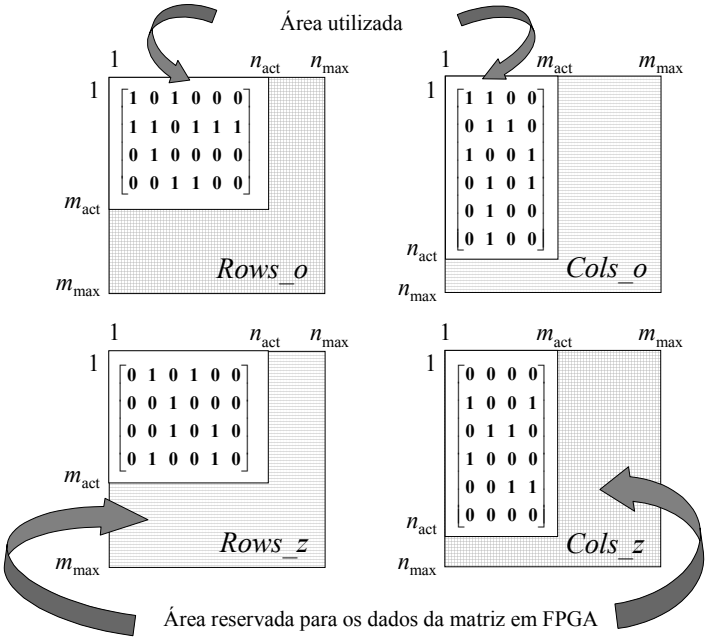


Fig. 6.7. Processamento da matriz em FPGA.

Os blocos restantes da fig. 6.6 têm a designação seguinte. A *ALU* é utilizada para calcular o número de *0*s e *1*s num vector ternário (i.e. em várias linhas e colunas da matriz – para os métodos 1, 4 e 5 na fig. 6.3), verificar se dois vectores ternários são ortogonais (i.e. se alguma linha da matriz é ortogonal ao vector **v** – para o método 2 na fig. 6.3), etc. Por exemplo, a fig. 6.8. representa um circuito que é empregue nos métodos 1 e 5 do algoritmo da fig. 6.3.

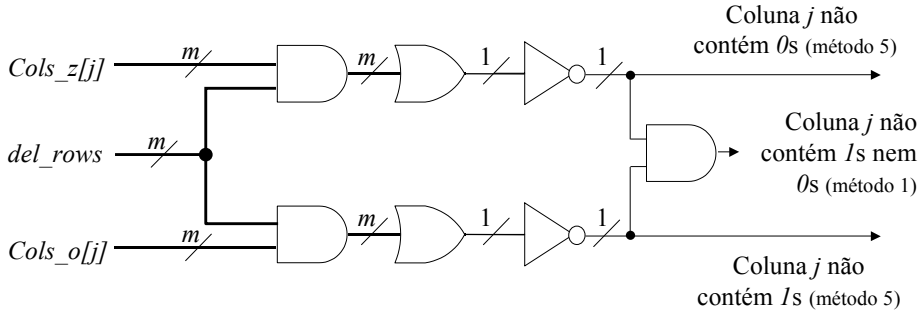


Fig. 6.8. Circuito utilizado nos métodos 1 e 5 do algoritmo da fig. 6.3.

A pilha é utilizada para suportar o processo de retrocesso na árvore de pesquisa. A pilha é composta por um contador de $\log_2(n\text{-}1)$ bits que guarda o nível de decisão corrente (i.e. a profundidade actual da árvore de pesquisa) e 6 blocos de RAM de profundidade (n-1) que mantêm os valores dos registos *cur_v_o*, *cur_v_z*, *del_rows*, *del_cols*, *des_var* e *test_val*, respectivamente para cada nível de decisão. Como resultado, a maior profundidade da árvore de pesquisa é (n-1). Obviamente, um número tão grande não é necessário e pode ser reduzido. Quando decompomos alguma situação, os valores de todos os registos são armazenados nos endereços respectivos dos blocos de RAM e o ponteiro da pilha é incrementado. Durante o retrocesso o ponteiro da pilha é decrementado e os valores necessários são recuperados, i.e. os dados são transferidos dos blocos de RAM para os registos. Para o caso do algoritmo híbrido é mantida a *lista de favorecidos* que tem a capacidade máxima de 64 entradas.

6.5 Experiências

Para as experiências foi utilizada a placa com interface PCI ADM-XRC da Alpha Data [Alpha] que contém uma FPGA XCV812E da família Virtex-EM da Xilinx [Xilinx_025]. Esta FPGA incorpora mais de 1 Mbits de memória organizada em blocos distribuídos por todo o dispositivo, e por isso, serve muito bem para a aplicação considerada porque podemos aproveitar a grande quantidade de memória disponível para guardar a matriz e organizá-la de maneira a assegurar a largura de banda necessária no acesso às linhas e às colunas. A interacção com a FPGA é feita com a ajuda da biblioteca de interface com a placa ADM-XRC que suporta inicialização e configuração da FPGA, transferência de dados, processamento de interrupções e erros, gestão de relógio, etc.

A reconfiguração do circuito (para personalizá-lo a uma dada instância do problema) é executada através do barramento PCI conforme representado na fig. 6.9. O *Controlador de reconfiguração* que é composto por uma máquina de estados finitos (FSM *F*) e um descodificador de endereços, recebe os dados do barramento PCI e transfere-os para os registos e os blocos de memória adequados. A FSM *F* implementa uma interface do tipo *Direct slave* na comunicação com o barramento local.

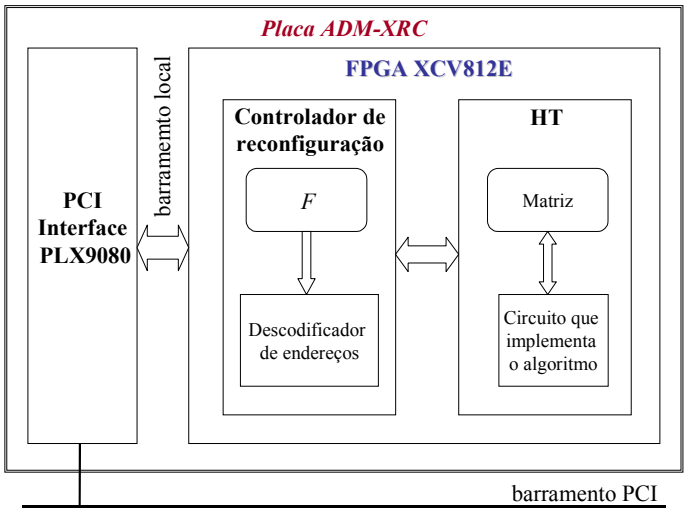


Fig. 6.9. Reconfiguração dinâmica parcial do circuito implementado.

6.5.1 Resultados obtidos com o algoritmo de retrocesso

Para o caso do algoritmo de retrocesso foram implementados 3 circuitos com as seguintes dimensões máximas da matriz ternária em FPGA: 64×32, 128×64 e 256×128 (vamos referenciar estes circuitos como *c64*, *c128* e *c256* respectivamente). A tabela 6.1 contém informação sobre a área ocupada e a frequência de relógio de cada um dos circuitos implementados. A área está expressa em número de *slices*, sendo cada CLB composto por dois *slices*.

Tabela 6.1. Parâmetros de circuitos implementados.

Circuito	Área ocupada (*slices*)	% dos recursos da XCV812E	Frequência de relógio (MHz)
c64	1557	16	40.725
c128	2848	30	32.858
c256	5158	54	30.516

Para estimar a eficiência da abordagem proposta realizámos uma série de experiências. Para tal escolhemos o problema *pigeon hole* do conjunto de instâncias de teste do DIMACS [DIMACS]. O problema consiste em determinar se é possível pôr n+1 bolas em n buracos sem ter duas bolas no mesmo buraco. Obviamente, isto é impossível, logo todas as instâncias são insatisfazíveis.

Este problema requer consumos significativos de tempo quando resolvido em software. As tabelas 6.2, 6.3 e 6.4 contêm os resultados da resolução do problema com a ajuda de uma aplicação desenvolvida em C++ que colabora de acordo com a estratégia descrita na secção 5.7 do capítulo anterior com os circuitos *c64*, *c128* e *c256* respectivamente implementados em FPGA. Vamos referenciar estas implementações partilhadas entre software e hardware reconfigurável como *soft*/*c64*, *soft*/*c128* e *soft*/*c256*. Se as dimensões iniciais da matriz excederem as capacidades do circuito implementado, o problema será primeiro processado pela aplicação de software que realiza o mesmo algoritmo. Pode-se ver que as dimensões iniciais da matriz para cada instância do problema considerado ultrapassam as capacidades dos circuitos *c64* e *c128*. Portanto, nestes casos cada problema é primeiro processado pela aplicação de software. Logo que a matriz intermédia satisfaça as restrições impostas pelo circuito, esta poderá ser transferida para a FPGA e processada lá. As colunas direitas das tabelas 6.2, 6.3 e 6.4 indicam quantas vezes é necessário utilizar a FPGA para cada instância e circuito. Como podemos ver, à medida que a área reservada para a matriz em hardware cresce, aumenta a parte da árvore de pesquisa que é processada em hardware (ver fig. 6.10) e o número de transferências da matriz para a FPGA diminui.

Tabela 6.2. Resultados das experiências com a arquitectura *soft*/*c64*.

Instância	$m \times n$	t_{GRASP} (*s*)	t_{config} (*s*)	t_{sw} (*s*)	t_{hw_comm} (*s*)	t_{total} (*s*)	Aceleração	FPGA
Hole6	133 × 42	0.14	0.31	0.0967	0.0763	0.483	0.289	60
Hole7	204 × 56	4.31		0.7771	0.5349	1.622	2.657	420
Hole8	297 × 72	51.574		7.544	4.314	12.168	4.238	3360
Hole9	415 × 90	531.96		72.1986	39.1474	111.656	4.764	30240
Hole10	561 × 110	5685.6		806.77	393.709	1200.789	4.734	302400

Tabela 6.3. Resultados das experiências com a arquitectura *soft*/*c128*.

Instância	$m \times n$	t_{GRASP} (*s*)	t_{config} (*s*)	t_{sw} (*s*)	t_{hw_comm} (*s*)	t_{total} (*s*)	Aceleração	FPGA
Hole6	133 × 42	0.14	0.35	0.0099	0.0256	0.3855	0.363	4
Hole7	204 × 56	4.31		0.1233	0.1789	0.6522	6.608	28
Hole8	297 × 72	51.574		1.1956	1.4374	2.983	17.289	224
Hole9	415 × 90	531.96		11.5529	12.9551	24.858	21.399	2016
Hole10	561 × 110	5685.6		122.993	129.982	253.325	22.444	20160

Tabela 6.4. Resultados das experiências com a arquitectura *soft*/*c256*.

Instância	$m \times n$	t_{GRASP} (*s*)	t_{config} (*s*)	t_{sw} (*s*)	t_{hw_comm} (*s*)	t_{total} (*s*)	Aceleração	FPGA
Hole6	133 × 42	0.14	0.37	0.0016	0.0176	0.3892	0.359	1
Hole7	204 × 56	4.31		0.0026	0.025	0.3976	10.840	1
Hole8	297 × 72	51.574		0.154	0.364	0.888	58.079	14
Hole9	415 × 90	531.96		1.6741	3.3139	5.358	99.284	126
Hole10	561 × 110	5685.6		17.7909	32.9481	51.109	111.245	1260

No nosso caso o tempo de resolução de um problema é:

$$t_{total} = t_{config} + t_{sw} + t_{hw\_comm}$$

O tempo de configuração da FPGA t_{config} varia de 0.31 s para o circuito *c64* a 0.37 s para o circuito *c256*, e começando com a instância *hole7* torna-se insignificante comparando-o com o tempo de execução em software "puro". É de salientar que o valor t_{config} pode ser omitido da consideração dado que a FPGA é configurada apenas uma vez e de seguida é utilizada para resolver uma série de problemas sem necessidade de ser reconfigurada. O valor t_{sw} é o tempo de solução da parte do problema em software (a parte superior da árvore de pesquisa da fig. 6.10). O valor t_{hw_comm} inclui o tempo de resolução da parte do problema em hardware reconfigurável (a parte inferior da árvore de pesquisa na fig. 6.10) e o tempo gasto em comunicações entre o computador hospedeiro e a FPGA (i.e. para transferir os dados da matriz para a FPGA e para receber o resultado da FPGA). A parte de software para todas as experiências foi executada num AMD Athlon/1GHz/256MB com o sistema operativo Windows2000 e a parte de hardware foi realizada na placa ADM-XRC ligada ao computador hospedeiro via barramento PCI.

Efectuamos uma comparação dos nossos resultados com os conseguidos em GRASP (*Generic seaRch Algorithm for the Satisfiability Problem*) [Silva99] que é um dos mais conhecidos algoritmos de solução de SAT implementados em software. O GRASP foi também executado num AMD Athlon/1GHz/256MB com o sistema operativo Windows2000 com as opções *+bD +dDLIS +S500* para as instâncias *hole6-hole9* e *+bD +dDLIS +S5000 +T5000* para a instância *hole10*. A aceleração conseguida é dada por expressão seguinte: t_{GRASP}/t_{total}, onde t_{GRASP} é o tempo de solução gasto pelo GRASP. Devido ao facto do circuito em FPGA ser especializado para as dimensões dos dados utilizados pelo algoritmo, as operações são executadas muito mais rápida e eficientemente do que em software. Como resultado, com o crescimento das dimensões máximas da matriz em

FPGA, observa-se o aumento da aceleração da nossa implementação em comparação com o GRASP (ver fig. 6.11).

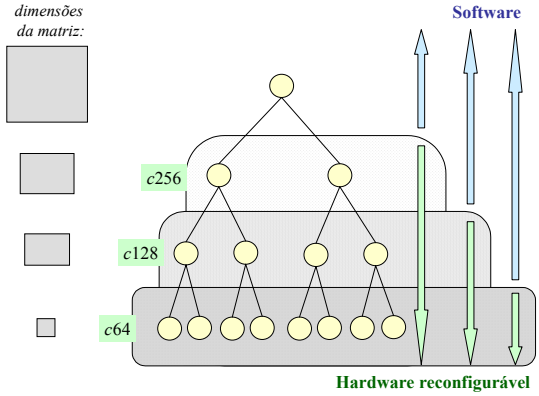


Fig. 6.10. Processamento da árvore de pesquisa em software e em hardware.

Na fig. 6.12 está apresentada a comparação da nossa arquitectura *soft*/*c256* com a arquitectura descrita em [Mencer99, Mencer01] que usa a abordagem orientada para a instância do problema. A frequência do relógio varia de 26 MHz a 65 MHz. O tempo de resolução de cada tarefa é igual a:

$$t = t_{\text{comp}} + t_{\text{config}} + t_{\text{hw}}$$

onde t_{comp} é o tempo de compilação do circuito de hardware (este domina em todas as instâncias consideradas), t_{config} é o tempo de configuração da FPGA e t_{hw} é o tempo de execução em hardware. Os valores do tempo de execução (com e sem o tempo de compilação) e da aceleração ganha em comparação com o GRASP foram reproduzidos de [Mencer99, Mencer01]. Contudo, em [Mencer01] o GRASP foi executado num PII/300MHz/128MB com o sistema operativo Linux. Parece-nos no entanto não ser possível efectuar uma comparação mais exacta porque ao mudar de plataforma de software, o tempo de compilação do hardware também será modificado.

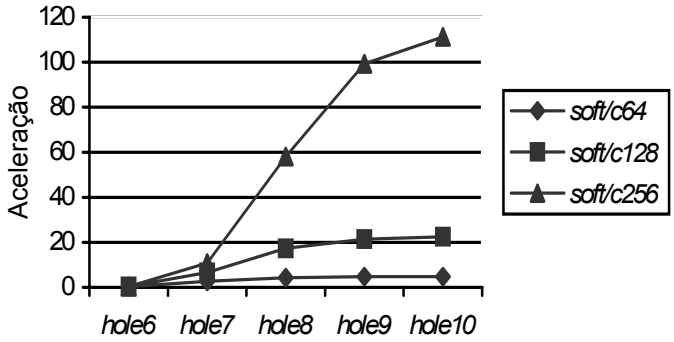


Fig. 6.11. Aceleração atingida com os três circuitos considerados em comparação com o GRASP.

É de notar que os resultados atingidos com um só conjunto de instâncias de teste não podem ser considerados como representativos. Por isso, foi realizada uma série de experiências com outras famílias de instâncias de teste disponíveis do DIMACS [DIMACS]. Os resultados respectivos que foram obtidos com a ajuda da arquitectura *soft*/*256* estão representados na tabela 6.5. O significado dos nomes das colunas é igual ao das tabelas 6.2-6.4. Contudo, na tabela 6.5 o valor t_{config} foi omitido porque a FPGA é configurada com o circuito *c256* apenas uma vez e a mesma

configuração é reutilizada por todas as instâncias. Como se pode ver da tabela 6.5 as instâncias de teste consideradas são facilmente resolúveis por GRASP. Consequentemente, a nossa abordagem não assegura uma aceleração útil para problemas que podem ser resolvidos em fracções do segundo por uma aplicação de software. Este facto pode ser explicado pelas razões seguintes. Primeiro, o GRASP é baseado num algoritmo mais avançado e inclui muitas técnicas complexas, tais como retrocesso não cronológico e adição dinâmica de cláusulas [Silva99]. A complexidade da FPGA XCV812E permite implementar algumas destas técnicas em hardware o que pode resultar em melhoramento do desempenho. A segunda razão é que para instâncias simples o *overhead* da comunicação é bastante significativo. Contudo, é possível aumentar o tamanho das matrizes que são processadas em hardware dado que a capacidade das FPGAs contemporâneas continua a crescer rapidamente. Consequentemente, o tempo de comunicação entre a FPGA e o computador hospedeiro pode ser reduzido substancialmente.

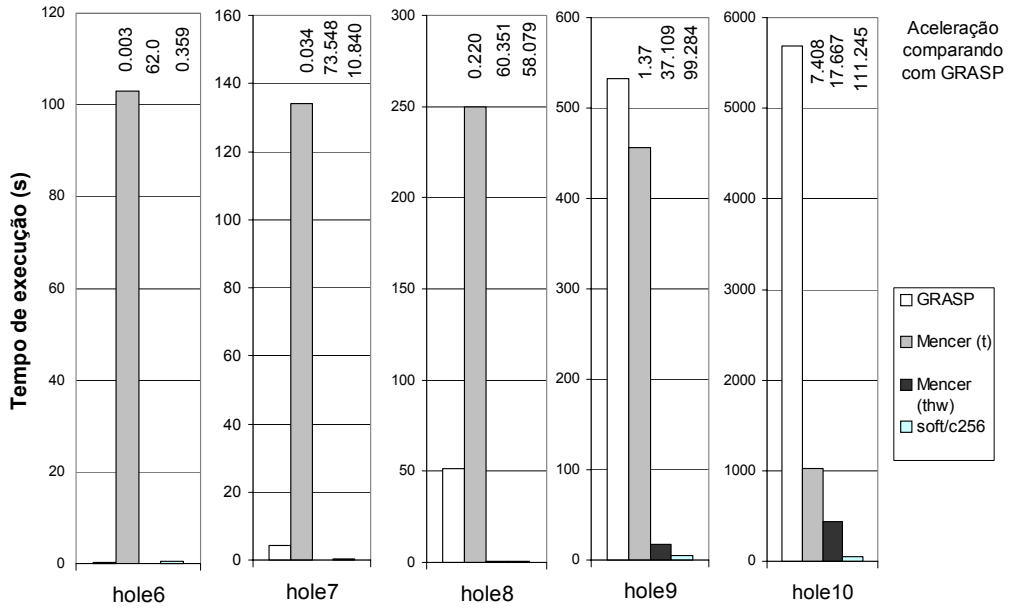


Fig. 6.12. Comparação com a arquitectura proposta em [Mencer99, Mencer01].

Tabela 6.5. Resultados das experiências com as instâncias de teste do DIMACS com a arquitectura *soft/c256*.

Instância	**Dimensões iniciais da matriz *(m × n)***	t_{GRASP} **(*s*)**	t_{total} **(*s*) (sem** t_{config}**)**	**Aceleração**
aim-100-3_4-yes1-4	340 × 100	0.03	0.08274	0.36
dubois20	160 × 60	0.02	0.01151	1.74
flat30-100	300 × 90	0.01	0.02290	0.44
ii8a1	186 × 66	0.01	0.00914	1.09
jnh1	850 × 100	0.08	0.04868	1.64
par8-4-c	266 × 67	0.02	0.01912	1.05
uf20-010	91 × 20	0.01	0.00504	1.98
2bitcomp_5	310 × 125	0.02	0.02061	0.97

6.5.2 Resultados obtidos com o algoritmo híbrido

Para o caso do algoritmo híbrido foi implementado um circuito que suporta as matrizes com as dimensões máximas de 256×128 (vamos referenciar este circuito como *hibr*/*c256*). A tabela 6.6 contém os resultados conseguidos para a classe de instâncias *pigeon hole* com a ajuda da aplicação desenvolvida em C++ que colabora de acordo com a estratégia descrita na secção 5.7 com o circuito *hibr*/*c256* implementado em FPGA. Como se pode ver das tabelas 6.4 e 6.6 o algoritmo híbrido produz melhores resultados para problemas de dimensões grandes enquanto para instâncias de dimensões pequenas o desempenho de ambas as arquitecturas é semelhante.

Tabela 6.6. Resultados das experiências com a arquitectura *hibr*/*c256*.

Instância	$m \times n$	t_{GRASP} (*s*)	t_{congif} (*s*)	t_{sw} (*s*)	t_{hw_comm} (*s*)	t_{total} (*s*)	**Aceleração**	**FPGA**
Hole6	133 × 42	0.14	0.36	0.0349	0.0074	0.4022	0.35	1
Hole7	204 × 56	4.31		0.0617	0.0113	0.4331	9.95	1
Hole8	297 × 72	51.574		0.2129	0.1443	0.7172	71.91	12
Hole9	415 × 90	531.961		1.3318	1.4476	3.1394	169.45	124
Hole10	561 × 110	5685.6		12.192	13.003	25.555	222.48	1246

6.6 Retrocesso não cronológico

Como já mencionámos todas as aplicações actuais destinadas a resolver problemas de SAT e implementadas em software incorporam técnicas avançadas que permitem evitar a exploração daquelas regiões do espaço de pesquisa que não contêm nenhuma solução. Nesta secção abordamos a possibilidade de implementação de algumas destas técnicas, tais como retrocesso não cronológico e adição dinâmica de cláusulas, em hardware reconfigurável.

Nos algoritmos descritos acima, caso ocorra um conflito, o processo de controlo retrocede sempre para a variável de decisão mais recente. Tal retrocesso é referenciado por *cronológico*. Contudo, a análise detalhada de causas de conflitos permite identificar as variáveis de decisão que são directamente responsáveis pelo conflito. Aproveitando esta informação, o algoritmo é capaz de retroceder logo para a mais recente das variáveis de decisão responsáveis cortando deste modo alguns ramos da árvore de pesquisa. Este processo é normalmente referenciado por *retrocesso não cronológico*.

Ilustremos esta característica com a ajuda da função seguinte:

$$\begin{aligned}&(\bar{x}_1 \vee \bar{x}_2)\ (\bar{x}_2 \vee \bar{x}_3)\ (x_1 \vee x_2 \vee x_3)\ (x_2 \vee x_7)\ (x_2 \vee x_8)\ (x_2 \vee x_9) \wedge \\ &\wedge (\bar{x}_4 \vee \bar{x}_5)\ (\bar{x}_5 \vee \bar{x}_6)\ (x_4 \vee x_5 \vee x_6)\ (x_3 \vee x_{13})\ (x_5 \vee x_8)\ (x_5 \vee x_7) \wedge \\ &\wedge (x_5 \vee x_9)\ (\bar{x}_{11} \vee \bar{x}_{12})\ (\bar{x}_{11} \vee x_{12})\ (x_{11} \vee \bar{x}_{12})\ (\bar{x}_2 \vee x_{11} \vee x_{12})\end{aligned} \tag{6.8}$$

Para encontrar a solução podíamos aplicar um algoritmo que descobre as cláusulas unitárias e efectua a selecção dinâmica de variáveis de decisão baseada na heurística de ocorrência máxima em cláusulas, descrita na secção 6.4.1. A árvore de decisão para a função (6.8) está representada na fig. 6.13a e requer que sejam visitados 12 vértices.

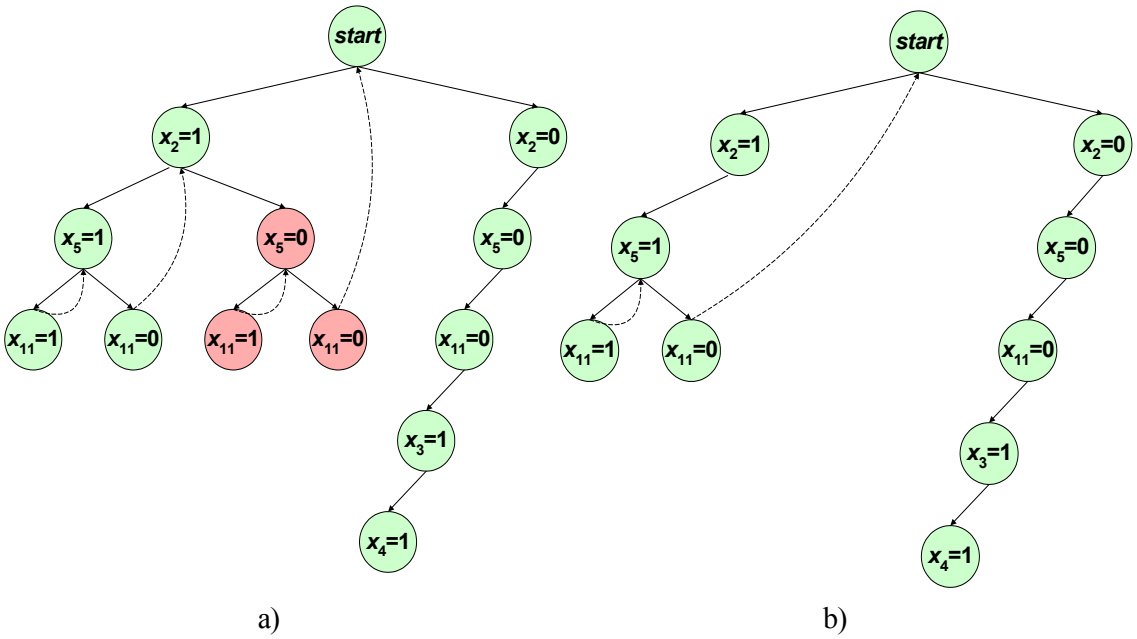


Fig. 6.13. Árvore de pesquisa construída para a função (6.8) aplicando o retrocesso cronológico (a); aplicando o retrocesso não cronológico (b).

A técnica de retrocesso não cronológico para SAT foi proposta em GRASP [Silva99]. Baseia-se na construção de um *grafo de implicação* que representa a sequência de implicações geradas durante a pesquisa. Quando aparecer um conflito, o grafo de implicação é analisado a fim de determinar aquelas decisões que são directamente responsáveis por este conflito. Isto requer a construção de cláusulas *conflituosas* (*conflict-induced*). De seguida, em vez de executar o retrocesso cronológico, o algoritmo pode "saltar" logo para a variável de decisão mais recente que faz parte da cláusula conflituosa. Como resultado, alguns ramos da árvore de pesquisa são cortados, poupando deste modo tempo que seria necessário para os explorar. A aplicação desta técnica à função (6.8) conduz à construção da árvore de pesquisa representada na fig. 6.13b. Neste caso apenas 9 vértices precisam de ser visitados.

Para além da aplicação no retrocesso não cronológico, as cláusulas conflituosas podem ser armazenadas permitindo deste modo prevenir a ocorrência de conflitos semelhantes posteriormente durante a pesquisa. Este processo é normalmente referenciado por *adição dinâmica de cláusulas*.

Para implementar as características descritas em hardware reconfigurável propomos a arquitectura representada na fig. 6.14. Nesta arquitectura, as cláusulas conflituosas são guardadas num registo de n bits e são geradas quando ambos os valores de uma variável de decisão foram testados sem sucesso (i.e. levaram a conflitos). A fim de construir a cláusula conflituosa, deve-se seguir e armazenar a sequência de implicações produzida durante o processo de pesquisa. Esta sequência é mantida no bloco de memória *Matriz de implicação* (IM) de dimensões $n \times n$ ilustrado na fig. 6.14.

Quando alguma variável é implicada, a matriz de implicação deve ser actualizada. Se a variável x_j, j=1,…,n, é implicada por causa da cláusula unitária c_i, i=1,…,m, então o valor que está no endereço j do bloco IM precisa de ser modificado. O valor novo deve reflectir a sequência completa de implicações que finalmente influenciaram a implicação da variável x_j. Se a cláusula inicial c_i for

igual a ($l_j \vee l_k \vee \ldots \vee l_r$), onde l_j, l_k,…,l_r representam literais, j,k,…,r=1,…,n, então no endereço j do IM escreve-se o valor seguinte: IM[j]=$c_j^B \vee$IM[k]$\vee \ldots \vee$IM[r], onde c_j^B é um vector booleano de n bits que contém *1*s em posições que correspondem aos literais encontrados em c_j, e contém *0*s em todas as posições restantes.

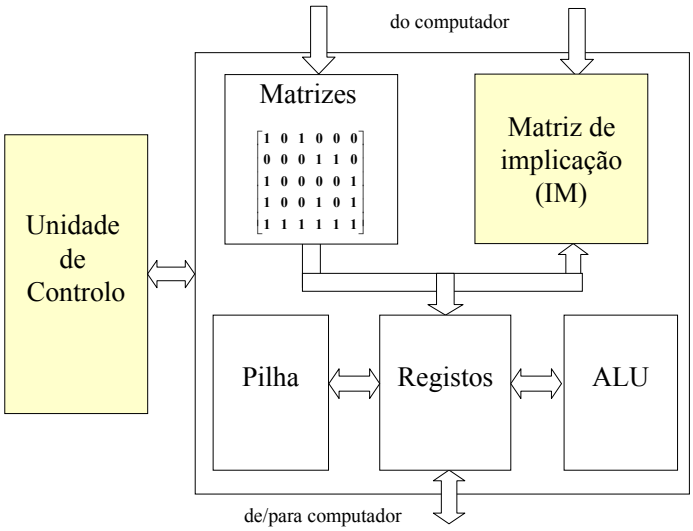


Fig. 6.14. Arquitectura aperfeiçoada.

Durante o retrocesso devemos apagar todas as acções executadas depois da decisão mais recente. Estas acções incluem:

- restauração das linhas e colunas eliminadas recentemente;
- reconstituição do valor anterior do vector ternário que estamos a procurar;
- anulação da sequência de implicações recentes.

Os dois primeiros pontos são de implementação fácil. Para estes fins os valores prévios dos registos respectivos são recuperados da pilha. Para implementar o último ponto, o conteúdo do bloco IM precisa de ser actualizado de maneira à variável de decisão corrente ficar removida da sequência de implicações. A realização desta tarefa requer a execução da operação seguinte para j=*1*,…,n: $\mathrm{IM[j]} = \mathrm{IM[j]} \wedge \overline{cur\_des\_var}$, onde *cur_des_var* é o valor binário descodificado do registo de $\log_2 n$ bits que guarda o número da variável de decisão corrente. Obviamente, esta operação é bastante dispendiosa por isso é melhor executá-la com a ajuda dum registo de n bits que marque com *1*s as colunas eliminadas do bloco IM.

Para construir a cláusula conflituosa precisamos de efectuar a disjunção daquelas linhas do bloco IM que correspondem aos literais encontrados na cláusula que é a causa do conflito.

Quando o processo de retrocesso é activado, devem ser seleccionadas apenas aquelas variáveis que entram na cláusula conflituosa corrente porque até que o valor de uma destas variáveis não seja invertido, irá aparecer o mesmo tipo de conflito. O algoritmo que incorpora o retrocesso não cronológico está representado na fig. 6.15.

Este algoritmo não foi implementado em hardware reconfigurável e só foram feitas algumas experiências num simulador de software.

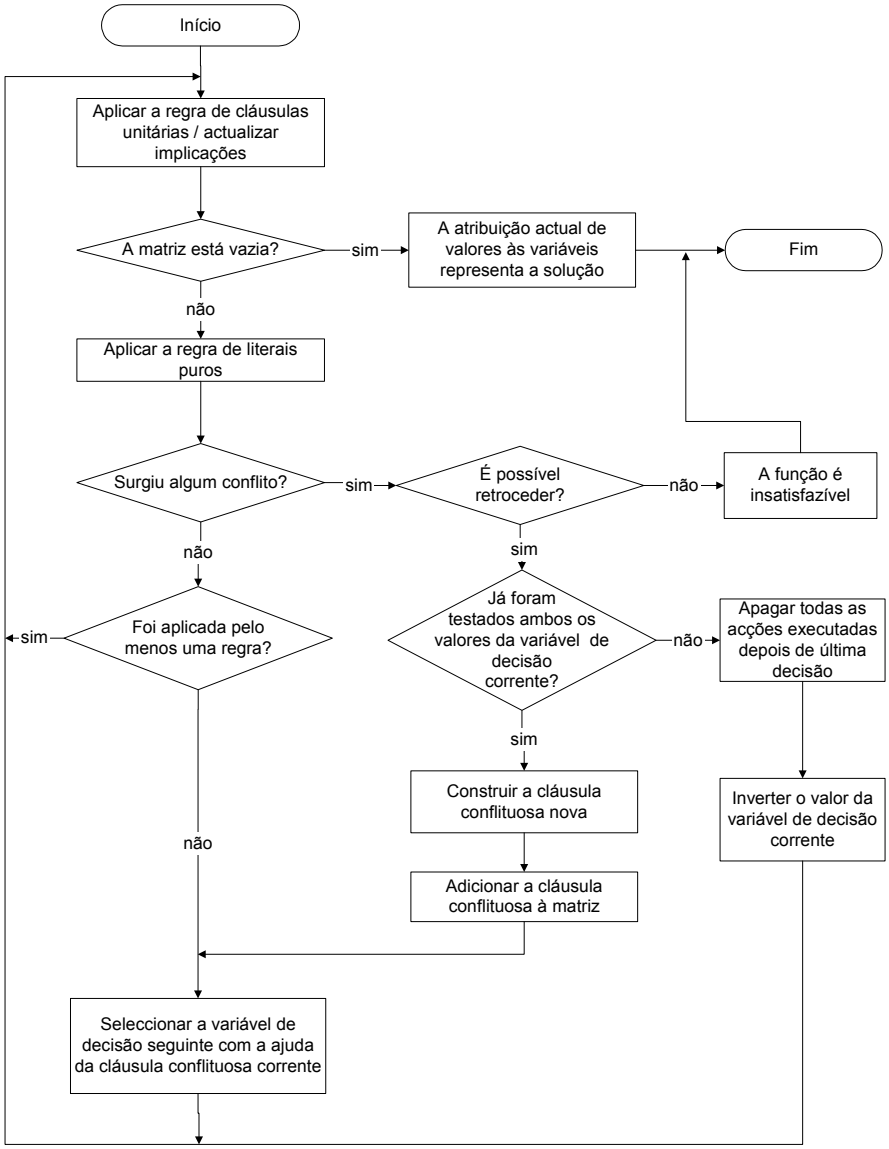


Fig. 6.15. Algoritmo que implementa retrocesso não cronológico.

6.7 Comparação com outras implementações baseadas em hardware reconfigurável

Descrevemos algumas das implementações mais conhecidas (as secções subsequentes são nomeadas de acordo com o nome do primeiro autor da ideia respectiva). As características principais de todas as arquitecturas consideradas estão sumariadas na tabela 6.7.

6.7.1 *Suyama et al.*

Suyama et al. [Suyama01, Yokoo96] propuseram um algoritmo que é capaz de encontrar todas as soluções (ou um número fixo destas) de uma instância. O algoritmo aplicado caracteriza-se pelo facto de em cada instante ser avaliada a atribuição *completa* de valores às variáveis.

Para escolher a variável de decisão seguinte os autores aplicaram a selecção estática [Yokoo96] e duas técnicas de selecção dinâmica. A primeira destas é baseada na atribuição experimental de valores *0* e *1* a cada variável livre (i.e. com valor indeterminado) [Suyama99]. A seguir é seleccionada a variável que produz o número máximo de cláusulas unitárias. A segunda técnica escolhe a variável que aparece no número máximo de cláusulas binárias [Suyama98].

A implementação utiliza a abordagem orientada para instância do problema (i.e. para cada instância é gerado um circuito específico). Cada função é analisada por um programa desenvolvido em C que gera a descrição comportamental respectiva num HDL. De seguida o circuito correspondente é sintetizado e implementado a partir do código HDL, empregando para tal as ferramentas de CAD disponíveis comercialmente. Aplicando esta estratégia, alguns circuitos foram realizados em FPGA FLEX10K250 da Altera [Altera] e funcionaram à frequência de 10 MHz, e os outros só foram simulados [Suyama01]. Sendo assim foi possível resolver algumas instâncias de SAT disponíveis do DIMACS [DIMACS] e conseguir alguma aceleração em comparação com o algoritmo POSIT (*PrOpositional SatIsfiability Testbed*) [Freeman95] realizado em software e executado num UltraSPARC-II/296MHz. Contudo, é de notar que o tempo de geração do circuito não foi considerado e não foi proposta nenhuma solução eficiente para o caso da função exceder significativamente os recursos de hardware disponíveis.

6.7.2 *Zhong et al.*

Zhong et al. [Zhong98b, Zhong99b] implementaram a versão do algoritmo de Davis-Putnam [Davis62]. Para cada variável na função foi construído um circuito de implicação e uma máquina de estados finitos (FSM). A seguir, todas as FSM foram ligadas numa cadeia série. Em cada instante de tempo só uma FSM está activa. Logo que esta acabe o processamento, o controlo é transferido quer à FSM à sua direita (pesquisa para diante) quer à sua esquerda (retrocesso), i.e. para atribuir valores às variáveis o circuito usa o controlo distribuído. Cada FSM guarda o valor corrente da sua variável (que pode ser *0*, *1*, ou indeterminado) e conhece se este valor foi atribuído ou implicado. O facto da última FSM tentar activar a FSM seguinte indica que a solução foi encontrada. A solução não existe quando a primeira FSM pretende passar o controlo à esquerda. Inicialmente, todas as variáveis são ordenadas de acordo com o número das suas ocorrências em cláusulas. Esta ordem estática é utilizada para arranjar as variáveis na cadeia série. Vários circuitos foram implementados com base no emulador IKOS e funcionaram a frequências de 700 KHz a 2 MHz [Zhong98b, Zhong99b]. A aceleração média em comparação com o GRASP [Silva99] executado em Sun5/110MHz/64MB (em modo restringido) foi de 64x para um subconjunto de instâncias do DIMACS [DIMACS] (não incluindo o tempo de geração do circuito e o tempo de configuração).

No trabalho mais recente [Zhong98a, Zhong99a] foi proposta uma nova arquitectura do circuito orientada às cláusulas. Múltiplas cláusulas foram organizadas em elementos de processamento os quais foram interligados num anel. Este projecto regular e modular permitiu reduzir significativamente o tempo de compilação de hardware (até à ordem de segundos) e aumentar a frequência do relógio (até 20-30 MHz) [Zhong99a]. Foi também proposta a técnica de adição dinâmica de cláusulas [Zhong98a] e a implementação do mecanismo de retrocesso não cronológico [Zhong99a].

6.7.3 *Platzner et al.*

A implementação proposta por *Platzner et al*. [Platzner98, Mencer99, Platzner00, Mencer01] é semelhante à de Zhong [Zhong99b]. O circuito contém uma coluna de FSMs, a lógica de dedução e uma unidade de controlo central. A lógica de dedução calcula o resultado da função com base em valores correntes das variáveis. As variáveis de decisão e os seus valores são seleccionadas da maneira estática.

Os autores implementaram alguns circuitos com base na placa Pamette que contém 4 FPGAs XC4028 da Xilinx e a frequência de relógio foi de 20 MHz. A aceleração conseguida para instâncias *hole6-hole10* do DIMACS [DIMACS] em comparação com o GRASP [Silva99] (executado em PII/300MHz/128MB) é de 0.003 a 7.408 vezes (aqui o tempo de compilação e configuração do hardware está incluído).

Em [Platzner98] foram propostas duas extensões à arquitectura básica. Na primeira extensão são identificadas as variáveis que não influenciam a solução. Isto permite evitar decisões sobre estas variáveis (a estratégia seguida é semelhante ao retrocesso pela sequência de cláusulas). Na segunda extensão são exploradas cláusulas unitárias (da mesma maneira como no algoritmo de Davis-Putnam [Davis62]). Os autores acreditam que o hardware reconfigurável só deve ser utilizado para instâncias difíceis que requerem muito tempo quando resolvidas em software, enquanto para problemas simples uma aplicação de software pode ser mais eficiente. Se a aplicação de software não conseguir resolver uma função durante um período de tempo predeterminado, então a tarefa pode ser transferida para a FPGA.

6.7.4 *Abramovici et al.*

Abramovici et al. [Abramovici97] aplicaram a técnica de modelação da função por um circuito arbitrário. A implementação é baseada no algoritmo PODEM (*Path-Oriented DEcision Making*) [Goel81] que é normalmente utilizado para resolver problemas de geração de padrões de teste. No trabalho desenvolvido posteriormente [Abramovici99b, Abramovici00] as funções foram modeladas por um circuito com dois níveis. Aí foi empregue um algoritmo derivado do de DP que implementa uma estratégia optimizada de selecção das variáveis de decisão.

Para a implementação em FPGA os autores sugerem criar um biblioteca de módulos básicos que podem ser utilizados para qualquer função. Os módulos têm o encaminhamento e a colocação internos predefinidos. Sendo assim, o circuito respectivo para cada instância do problema é construído dos módulos, o que permite reduzir significativamente o tempo de compilação de hardware (até à ordem de alguns minutos). Os autores implementaram vários circuitos simples em FPGA XC6264 e simularam os mais complexos. Para um circuito que ocupa toda a área da XC6264 a frequência de relógio é de 3.5 MHz. A aceleração conseguida em comparação com o GRASP [Silva99] executado em Sun UltraSparc/248MHz/1026MB foi de 0.01 a 7000 vezes (após ajuste da frequência de relógio) para um subconjunto de instâncias do DIMACS [DIMACS]. Contudo, o tempo de compilação e configuração de hardware não foi considerado.

Em [Abramovici99a] foi proposto o sistema de lógica virtual que permite construir circuitos capazes de resolver funções que excedem os recursos de hardware disponíveis. Isto é atingido

através da decomposição da função em sub-funções independentes que podem ser processadas em FPGAs separadas quer em paralelo, quer sequencialmente. Deve-se notar contudo, que a eficiência da decomposição depende muito da função em causa. Em vista disso, a decomposição de algumas funções pode tornar-se pouco prática pois aumenta o tempo de compilação até níveis inaceitáveis.

O trabalho mais recente realizado nesta direcção é orientado para a eliminação da compilação de hardware específica para a instância. Assim, *Boyd et al*. [Boyd00] propuseram uma arquitectura do PLD orientado para a solução do problema de SAT que exclui a colocação e encaminhamento específicos à instância. O projecto consome recursos de hardware polinomiais (de acordo com o número de variáveis e cláusulas) e requer tempo polinomial para a configuração. Os autores implementaram uma pequena versão do seu circuito para um problema composto por 7 variáveis e 8 cláusulas numa FPGA XC4005XL que executa a 12 MHz. Contudo, não foram reportados resultados sobre instâncias de teste de maiores dimensões.

6.7.5 *Dandalis et al.*

Dandalis et al. [Dandalis02, Dandalis01, Redekopp00] propuseram uma arquitectura capaz de avaliar cláusulas em paralelo na fase de dedução de implicações. A característica distintiva da sua abordagem é que o circuito evolui dinamicamente durante a execução a fim de atingir o nível de paralelismo mais adequado e resultar assim no melhor desempenho. A arquitectura consiste num conjunto de grupos que executam em paralelo. A estrutura dum grupo é semelhante à proposta em [Zhong98c] e inclui um *array* linear de módulos interligados que correspondem às cláusulas, sendo todas as variáveis espalhadas entre os blocos de memória locais associados aos módulos. Os módulos de cláusulas recebem variáveis nas suas entradas e geram implicações ou conflitos. Os resultados provenientes de todos os grupos de módulos devem ser reunidos a fim de actualizar os valores das variáveis. A estrutura dos módulos é genérica e para fazer com que um módulo corresponda a uma cláusula, é necessário programar o conteúdo do bloco de memória adequado.

É de salientar que o encontro da partição óptima de cláusulas em grupos e o seu ordenamento dentro do grupo para uma dada instância do problema requer a solução desta mesma instância. Portanto, são necessários métodos heurísticos capazes de assegurar a partição de cláusulas adequada [Redekopp00]. A evolução da arquitectura consiste em encontrar o número de grupos óptimo para uma dada instância. Para tal os autores propuseram desenvolver uma série de *templates* com vários níveis de paralelismo e, durante a execução, aplicar uma heurística que permita decidir que *template* utilizar num dado momento. Os *templates* são comutados via reconfiguração dinâmica parcial ou global. Os resultados apresentados em [Dandalis02, Dandalis01, Redekopp00] baseiam-se na simulação em software, tornando difícil a avaliação das vantagens da arquitectura face às reconfigurações dinâmicas necessárias.

6.7.6 *Leong et al.*

Leong et al. investigaram a possibilidade e eficiência de implementação de algoritmos completos [Chung99] e incompletos [Yung99, Leong01] em hardware reconfigurável. Os algoritmos *incompletos* são algoritmos aproximados (normalmente os de pesquisa local) que não são capazes de provar a inexistência da solução para uma função.

Em [Leong99] apresenta-se a arquitectura dum componente que avalia as cláusulas da função. A arquitectura baseia-se num *hardware template* que inclui blocos de memória (construídos a partir das LUTs) que podem ser personalizados para qualquer instância do problema que satisfaz as restrições no número de cláusulas e variáveis. A personalização efectua-se modificando o *bitstream* do *hardware template* que, de seguida, é carregado para a FPGA. Deste modo a reconfiguração é realizada bastante rapidamente.

Na investigação posterior foi proposto o sistema resolvente de SAT [Leong01] que implementa o algoritmo incompleto WSAT [Gu97]. A arquitectura baseia-se também num HT eliminando deste modo a necessidade de compilação de hardware sendo só necessário modificar o *bitstream* (para o personalizar para uma dada instância). Os resultados reproduzidos em [Leong01], obtidos com as instâncias de teste do DIMACS [DIMACS] mostram uma aceleração de 0.1 a 3.3 em comparação com a realização do mesmo algoritmo em software (numa Sun SparcStation 20). A implementação em hardware baseia-se numa FPGA XCV300 da família Virtex da Xilinx. O circuito executa a 33 MHz e pode acomodar funções com o máximo de 50 variáveis e 170 cláusulas.

Para além disso foi também proposta a implementação do algoritmo incompleto GSAT [Gu97] com base em FPGA da família XC6200 da Xilinx [Yung99]. Contudo os resultados atingidos são piores que a solução em software (aplicando o mesmo algoritmo).

6.7.7 *Sousa et al.*

Sousa et al. [Sousa01] implementaram um algoritmo de pesquisa derivado do de Davis-Putnam composto por três fases que são a decisão, a dedução e o diagnóstico. Na primeira fase, a variável de decisão e o seu valor são seleccionados de maneira dinâmica. Na dedução todas as implicações unitárias são calculadas. A fase de diagnóstico só é activada quando surge um conflito. Esta consegue identificar as variáveis de decisão que o causaram, possibilitando deste modo o retrocesso não cronológico. A análise de conflitos permite também construir e adicionar à função novas cláusulas que ajudam a acelerar o processo da pesquisa.

Sousa et al. propuseram dividir o trabalho entre software e hardware reconfigurável sendo a parte mais intensiva em termos computacionais (esta inclui o cálculo das implicações e a selecção da variável de decisão seguinte) atribuída ao hardware, enquanto as tarefas de controlo (tais como análise de conflitos, gestão do retrocesso, etc.) são executadas em software. Neste caso o computador hospedeiro recebe e envia à FPGA as variáveis que mudaram de valor, e adiciona ou retira algumas cláusulas. A arquitectura proposta é orientada para a aplicação e aproveita os registos de configuração para instanciar uma função [Reis02]. Para lidar com as instâncias que excedem os recursos de hardware disponíveis, foi proposto um esquema de hardware virtual com comutação de contexto. Isto é efectuado através da divisão do circuito original em páginas de hardware que são executadas sequencialmente, sendo os resultados intermédios guardados numa memória externa. Os resultados publicados em [Sousa01, Ripado01] são baseados numa simulação em software sob a frequência estimada de 80 MHz e assumindo que é possível substituir páginas de hardware num ciclo de relógio. Para as instâncias *hole9-hole10* do DIMACS [DIMACS] a aceleração conseguida em comparação com o GRASP (a plataforma de software não foi publicada) foi de 3 vezes, não tomando em conta a comunicação entre a FPGA e o processador hospedeiro.

6.7.8 Análise

Nesta secção são analisadas várias implementações propostas de acordo com os seguintes critérios: algoritmos utilizados, modelo de programação, modelo de execução, modo de reconfiguração, capacidade lógica e desempenho, i.e. com base em critérios considerados no capítulo 3. A tabela 6.7 sumaria as características respectivas das arquitecturas descritas acima.

Tabela 6.7. Comparação de várias implementações baseadas em hardware reconfigurável.

Implementação	**Algoritmo**	**Modelo de programação**	**Modelo de execução**	**Modo de reconfiguração**	**Capacidade lógica**
Suyama et al. [Suyama01]	completo parecido com o DP com a selecção dinâmica	específico à instância	só hardware	estático	sistema composto por múltiplas FPGAs
Zhong et al. [Zhong99a]	baseado em DP com a selecção estática, retrocesso não cronológico, análise de conflitos	específico à instância	só hardware	dinâmico (global)	sistema composto por múltiplas FPGAs
Platzner et al.	baseado em DP com a selecção estática	específico à instância	só hardware	estático	utilização de um dispositivo com a capacidade maior
Abramovici et al. [Abramovici00]	baseado em DP com a selecção estática optimizada	específico à instância	só hardware	dinâmico (global)	partição lógica em sub-funções
Dandalis et al. [Dandalis02]	baseado em DP com a selecção estática	específico à aplicação	implicações são processadas em hardware optimizável durante a execução, o resto é realizado em software	dinâmico (parcial e global)	sistema composto por múltiplas FPGAs
Leong et al. [Leong01, Yung99]	algoritmos incompletos: WSAT, GSAT	específico à aplicação	ciclo interno em hardware, atribuição inicial de valores às variáveis em software	dinâmico (global)	nada proposto
Sousa et al.	baseado em DP com a selecção dinâmica e análise de conflitos (em software)	específico à aplicação	partição entre software e hardware de acordo com complexidade computacional	dinâmico (parcial)	esquema de hardware virtual
Nossa implementação	baseado em DP com a selecção dinâmica	específico à aplicação	partição entre software e hardware de acordo com capacidade lógica	dinâmico (parcial)	partição entre software e hardware

6.7.8.1 Algoritmos

A maioria das implementações baseadas em hardware reconfigurável recorrem às variantes do algoritmo de Davis-Putnam [Davis62]. As excepções a esta regra são as arquitecturas de *Leong et al*. [Leong01, Yung99], *Yap et al*. [Yap03] e de *Hamadi et al*. [Hamadi97] que implementam algoritmos incompletos e as de *Abramovici et al*. [Abramovici97] e de *Rashid et al*. [Rashid98] que implementam algoritmos derivados do PODEM [Goel81]. Embora a selecção dinâmica da variável de decisão seguinte tenha sido considerada uma tarefa difícil para a realização em hardware, esta foi incorporada numa série de arquitecturas [Suyama01, Sousa01, Skliarova02b].

Hoje em dia, todas as implementações baseadas em software aplicam inúmeras técnicas avançadas (tais como retrocesso não cronológico, adição dinâmica de cláusulas, exploração da simetria do problema, etc. [Bayardo97, Silva99, Moskewicz01, Goldberg02a, Goldberg02b]) que permitem identificar e desviar daquelas regiões no espaço de pesquisa que não contêm nenhuma solução. Contudo, até agora, estas técnicas têm sido bastante ignoradas em arquitecturas baseadas em hardware reconfigurável. Algumas excepções a esta regra são as implementações de *Zhong et al*. [Zhong99a] e *Sousa et al*. [Sousa01] (a última atribui estas tarefas ao software).

6.7.8.2 Modelo de programação

Existem duas abordagens principais para mapear uma função num sistema reconfigurável: *específica à instância* e *específica à aplicação*. A primeira abordagem foi explorada intensivamente pela comunidade de investigação de SAT [Yokoo96, Suyama01, Zhong99a, Abramovici00, Platzner00] e pressupõe a geração duma configuração individual de hardware para cada instância de problema. Neste caso, utiliza-se o fluxo de projecto típico para descrever e implementar ou todo o circuito específico à instância ou um conjunto de módulos primários que serão posteriormente interligados e personalizados (durante a compilação) pelas ferramentas de software desenvolvidas para o efeito fazendo com que os módulos correspondam às características da função respectiva.

Na *abordagem orientada para a aplicação* o circuito é projectado e optimizado apenas uma vez podendo a seguir ser utilizado para várias instâncias do problema [Boyd00, Sousa01, Dandalis01, Skliarova02b]. Isto pode ser atingido com a ajuda de *hardware templates* que são circuitos desenvolvidos utilizando o fluxo de projecto típico, mas que são personalizados em *run-time* (em vez de isto acontecer durante a compilação). É de notar que neste caso o passo de compilação de hardware é completamente eliminado.

6.7.8.3 Modelo de execução

Uma função pode ser completamente mapeada para o hardware reconfigurável (deixando para o processador hospedeiro apenas as tarefas de pré-processamento e de inicialização) [Yokoo96, Suyama01, Zhong99a, Abramovici00, Platzner00, Boyd00] ou partilhada entre hardware e software [Sousa01, Skliarova02b].

Existem várias maneiras de partilhar uma aplicação entre software e hardware. No domínio de SAT, as duas seguintes são normalmente empregues: *partição de acordo com a complexidade computacional* e *partição respeitando a capacidade lógica*. O primeiro método atribui porções computacionalmente intensivas da aplicação ao hardware enquanto as porções restantes são geridas

pelo processador hospedeiro [Sousa01, Reis02, Dandalis01]. Por exemplo, a parte de determinação das implicações transitivas é a que pode beneficiar mais da implementação em hardware. O segundo método efectua a partição de acordo com a capacidade lógica do hardware utilizado [Skliarova02a, Skliarova02b]. Neste caso se a instância não "cabe" no dispositivo seleccionado, esta será primeiro processada em software conforme descrito na secção 5.7.

6.7.8.4 Modos de reconfiguração

Todas as implementações propostas na fase inicial da investigação nesta área foram orientadas para a instância do problema e por isso exigiram apenas a reconfiguração estática [Yokoo96, Suyama01, Zhong99b, Zhong98b, Platzner98, Platzner00]. Quase todas as arquitecturas mais recentes suportam a reconfiguração dinâmica global [Abramovici00, Leong01] ou parcial [Sousa01, Skliarova02a, Dandalis02].

6.7.8.5 Capacidade lógica

A capacidade lógica do hardware empregue é sempre limitada. Por conseguinte são necessários métodos eficientes capazes de lidar com a situação quando uma instância de problema excede os recursos de hardware disponíveis. A resposta a esta questão varia de acordo com os modelos de programação e execução adoptados. No essencial, foram exploradas as quatro possibilidades seguintes.

A primeira é a expansão da capacidade lógica através de interligação de um número de dispositivos programáveis e de partição do circuito entre estes. É de salientar que a partição e o encaminhamento de um sistema composto por múltiplos dispositivos é uma tarefa bastante difícil (obviamente os estilos de projecto modulares [Zhong99a] podem aliviá-la). Para além disso, a frequência de funcionamento de tais sistemas é normalmente limitada.

O segundo método consiste em transformar uma instância numa série de configurações que serão executadas quer sequencialmente quer em paralelo. A partição é realizada através de decomposição da função inicial num conjunto de sub-funções independentes [Abramovici00]. Cada sub-função deve satisfazer as restrições de hardware impostas. A maior limitação deste método é que a eficiência da decomposição depende fortemente das características da função em causa.

O terceiro método, proposto neste trabalho, baseia-se na partição da aplicação entre software e hardware de acordo com a capacidade lógica disponível (ver secção 6.7.8.3). Neste caso apenas os subproblemas que aparecem em vários níveis da árvore de pesquisa e respeitam as restrições de capacidade serão atribuídos ao hardware, ficando a parte restante do problema a cargo da aplicação de software.

O quarto método recorre ao esquema de hardware virtual proposto em [Sousa01, Reis02] que requer a divisão do circuito numa série de páginas de hardware que são executadas sucessivamente, ficando os resultados intermédios guardados em blocos de memória externos. Dado que todas as páginas de hardware possuem estrutura semelhante, contendo apenas um conjunto de registos que precisam de ser reconfigurados, a comutação de páginas é efectuada muito rapidamente. Esta técnica é semelhante ao mecanismo de memória virtual empregue em computadores convencionais.

6.7.8.6 Desempenho

O tempo total gasto por um sistema reconfigurável para resolver uma instância de problema, inclui quatro componentes: o tempo de compilação de hardware, o tempo de configuração de hardware, o tempo necessário para as comunicações entre software e hardware e o tempo actual de execução. Se a solução do problema está partilhada entre software e hardware, então o tempo de execução é composto por tempo de execução em software e tempo de execução em hardware. É de notar que os valores dos componentes mencionados dependem dos modelos de programação e de execução empregues e alguns podem ser iguais a zero. Por exemplo, se a instância do problema está inteiramente mapeada para o hardware, então normalmente não há nenhuma comunicação (à excepção da notificação do resultado) entre o processador hospedeiro e o dispositivo reconfigurável. Igualmente, na abordagem orientada para a aplicação o tempo de compilação de hardware é nulo. É de realçar que o tempo de compilação de hardware pode constituir uma porção significativa do tempo de solução total. Para as instâncias simples este pode até dominar cancelando assim todas as vantagens da execução rápida em hardware [Yokoo96, Suyama01, Zhong98b, Platzner00]. Por esta razão foi proposta uma série de técnicas destinadas a reduzir o tempo de compilação de hardware. Estas são baseadas na exploração de estilos de projecto modulares e no desenvolvimento de ferramentas de software personalizadas capazes de substituir eficientemente as disponíveis comercialmente [Zhong98c, Abramovici00, Dandalis01].

Uma característica inerente a implementações de algoritmos de solução de SAT em hardware reconfigurável é que é difícil analisar e comparar o seu desempenho de um modo preciso. Normalmente, os investigadores representam os seus resultados à luz do programa resolutor GRASP [Silva99]. Contudo, o GRASP é executado sobre plataformas diferentes utilizando parâmetros dissemelhantes o que influencia fortemente o seu desempenho. Para além disso, os parâmetros impostos nem sempre são publicados. Muitos sistemas reconfiguráveis necessitam dum passo de compilação de hardware que é às vezes ignorado (ou escondido) quanto à apresentação dos resultados. Este facto também contribui para tornar difícil a avaliação do impacto do tempo de compilação no tempo total por causa da variedade de plataformas de software utilizadas. Todavia, todas as arquitecturas recentes demonstram uma clara intenção de reduzir e até evitar o passo de compilação de hardware [Zhong99a, Abramovici00, Dandalis01, Boyd00, Sousa01, Skliarova02b].

Como mostraram os resultados das duas últimas competições de programas resolutores de SAT decorridas em 2002 e 2003 [Simon02, SAT03], o GRASP foi ultrapassado por programas mais recentes tais como zChaff [Moskewicz01] e BerkMin [Goldberg02b]. Consequentemente, é necessário explorar técnicas algorítmicas e arquitecturais inovadoras para pôr os sistemas reconfiguráveis à luz mais favorável perante as implementações em software.

6.8 Conclusões

Neste trabalho investigamos a possibilidade de utilização do hardware reconfigurável para a solução de problemas de optimização combinatória. Como um estudo de caso apresentámos várias arquitecturas de circuitos capazes de resolver o problema de *satisfação* booleana. A técnica proposta elimina o passo de compilação de hardware e permite processar problemas cujas dimensões excedem os recursos de hardware disponíveis. Como resultado conseguimos obter uma

aceleração bastante significativa para algumas instâncias de SAT em comparação com a aplicação de software GRASP. Deve ser notado que estas instâncias são difíceis enquanto a utilidade de mapeamento das instâncias simples (que são facilmente resolúveis em software) para o hardware reconfigurável é questionável. A abordagem proposta foi também comparada com outras implementações. A análise cuidadosa de várias arquitecturas reconfiguráveis destinadas a acelerar a solução do problema de SAT permite chegar às conclusões seguintes:

- A maioria dos investigadores implementam algoritmos de pesquisa completos derivados do algoritmo de Davis-Putnam. A análise de conflitos normalmente não é realizada (com poucas excepções).
- Inicialmente, todas as arquitecturas se basearam na abordagem orientada para a instância. Contudo o tempo de compilação do circuito de hardware limitava significativamente a gama de problemas para os quais a utilização de sistemas reconfiguráveis poderia ser considerada razoável em comparação com a solução em software. Em vista disso, em todos os trabalhos recentes tenta-se evitar a colocação e o encaminhamento específicos à instância.
- Todos os sistemas reconfiguráveis propostos para a solução de SAT são de fraca integração, sendo o dispositivo programável (normalmente, uma FPGA disponível comercialmente) ligada ao processador hospedeiro via uma interface externa.
- É bastante difícil comparar os resultados atingidos. Primeiro, os tempos de compilação e de configuração de hardware nem sempre estão claramente expostos nas publicações respectivas. Segundo, os investigadores comparam os seus próprios resultados com a aplicação de software GRASP, mas executam o GRASP em plataformas diferentes e usando opções diferentes. Como resultado, os tempos de solução da mesma instância em software podem diferir em vários casos até ao nível de ordens de grandeza.
- Muitos autores evitam enfrentar a situação quando os recursos de hardware não são suficientes para realizar uma função. Contudo, as funções encontradas em aplicações práticas são normalmente de grandes dimensões. Ultimamente tem sido proposta e realizada a ideia de dividir de algum modo o problema entre software e hardware, o que parece ser uma solução promissora e eficiente. É de notar que devido à evolução rápida das capacidades de FPGAs, muitos problemas interessantes podem actualmente ser mapeados numa única FPGA ou pelo menos ser partilhados de uma maneira mais eficiente.
- As acelerações conseguidas com os sistemas reconfiguráveis comparando-as com a solução baseada em software só são significativas para instâncias difíceis de SAT, para as quais as técnicas de optimização propostas e implementadas em aplicações de software são pouco eficientes. Por isso, embora já tenham sido propostas muitas arquitecturas interessantes, é necessário explorar abordagens inovadoras no domínio da computação reconfigurável.
- A arquitectura aqui proposta compara favoravelmente com as outras soluções documentadas na literatura.

7 Aceleração de algoritmos evolutivos

Sumário

Os algoritmos evolutivos (EA – *Evolutionary Algorithms*) revelaram-se uma abordagem efectiva no encontro de soluções sub-óptimas para problemas de optimização combinatória. Este capítulo analisa a possibilidade de aceleração de EA para o caso do problema do caixeiro viajante (TSP – *Traveling Salesman Problem*) com a ajuda de hardware reconfigurável.

Começamos com a descrição do problema do caixeiro viajante que possui bastantes aplicações práticas em áreas diversas. A seguir fazemos uma introdução à computação evolutiva, mencionando os seus ramos mais conhecidos e demonstrando a estrutura típica de um algoritmo evolutivo. Analisamos as vantagens principais dos algoritmos evolutivos tais como simplicidade, aplicabilidade, possibilidade de se adaptar às alterações dinâmicas na especificação e resolver problemas que não possuem soluções conhecidas.

De seguida, descrevemos um algoritmo evolutivo que empregámos para a solução do problema do caixeiro viajante. Para tal definimos a representação de uma solução, a função de adequabilidade, os operadores de mutação e de cruzamento e os critérios de selecção. O algoritmo foi implementado numa aplicação de software desenvolvida em C++. Após realizar uma série de experiências com as instâncias de teste disponíveis da TSPLIB (*Library of Traveling Salesman and related Problems*) e analisar os resultados, descobrimos que a parte mais crítica do algoritmo considerado é a operação de cruzamento. Sendo assim, o aumento da eficiência desta operação pode influenciar significativamente o desempenho de todo o algoritmo. Por isso sugerimos o mapeamento da operação de cruzamento para o hardware reconfigurável. A arquitectura proposta foi implementada na FPGA XCV812E da Xilinx instalada na placa ADM-XRC que comunica com o processador hospedeiro via barramento PCI. Concluímos o capítulo com a avaliação e comparação dos resultados atingidos com ambas as abordagens, i.e. com as implementações do operador de cruzamento baseadas em software e em hardware reconfigurável.

7.1 Introdução

O TSP é o problema dum caixeiro que começando da sua casa, quer viajar por um conjunto específico de cidades e voltar para casa no fim [Golden00]. Cada cidade deve ser visitada uma só vez e, obviamente, o caixeiro precisa de encontrar o caminho mais curto. Em termos mais formais, o problema pode ser representado por um grafo interpretado $G=(V,E,W)$, onde $V=\{0,\ldots,n-1\}$ é um conjunto de vértices que correspondem às cidades, E é um conjunto de arcos que identificam as estradas existentes entre as cidades, e W é um conjunto de pesos atribuídos aos arcos. Cada peso $w_{ij} \in Z^+$ especifica a distância entre as cidades i e j, $i,j=0,...,n-1$. Portanto o problema do caixeiro viajante consiste em encontrar o ciclo hamiltoniano mais curto num grafo interpretado. Neste trabalho só abordamos o TSP simétrico para o qual $w_{ij}=w_{ji}$ para cada par de cidades.

TSP é um dos problemas de optimização combinatória mais conhecidos que possui muitas aplicações práticas em áreas diversas tais como cristalografia de raios X [Jünger95], planeamento de tarefas [Golden00], perfuração de placas de circuito impresso [Litke84], alinhamento de sequências de ADN [Ahuja95], etc. Embora seja bastante fácil formular o problema, é extremamente difícil resolvê-lo (TSP pertence à classe de problemas *NP-hard* [Garey79]). É por isso que se efectuam várias tentativas de acelerar de alguma maneira a solução de TSP.

Como demonstrámos no capítulo 4, os problemas de optimização combinatória podem ser resolvidos com uma ampla gama de algoritmos que variam desde os métodos exactos até às abordagens heurísticas aproximadas. Os métodos exactos são muito atraentes dado que retornam o resultado óptimo, mas podem falhar quando aplicados a problemas muito grandes. No mesmo tempo, os métodos heurísticos são frequentemente capazes de encontrar soluções de qualidade aceitável para muitos problemas práticos. Os resultados bons são atingidos com a ajuda de heurísticas específicas para um problema particular. Para além disso são aplicadas heurísticas gerais (i.e. as que podem ser usadas para uma variedade de problemas), tais como *simulated annealing* e *algoritmos evolutivos*. Estes algoritmos são bastante lentos quando aplicados a problemas grandes. Em vista disso, há todo o interesse em suportar a sua execução com sistemas computacionais de desempenho elevado, assim como sistemas reconfiguráveis orientados para a aplicação [Abramson97].

O TSP serve bastante bem para os algoritmos evolutivos pois é um problema *NP-hard*, possui um espaço de soluções grande e a sua função de adequabilidade é facilmente calculável. Neste capítulo analisamos várias possibilidades de implementação de EA para o TSP.

7.2 Computação evolutiva

As técnicas de computação evolutiva são métodos de optimização probabilísticos que manipulam uma população de indivíduos (i.e. soluções potenciais) e baseiam-se na teoria de Darwin de selecção natural e evolução. As técnicas evolutivas são normalmente utilizadas para problemas de optimização e de pesquisa [Gen00]. Na área da computação evolutiva destacam-se vários paradigmas tais como *algoritmos genéticos*, *estratégias evolutivas*, *programação evolutiva* e a *programação genética*. Os *algoritmos genéticos* tradicionais [Goldberg89] manipulam *strings*

binárias de comprimento fixo sobre as quais são definidos operadores de mutação e cruzamento (estes operam sem ter conhecimento algum sobre a interpretação das *strings*). No caso das *estratégias evolutivas*, a representação é um vector de valores reais que é manipulado pelos operadores de mutação [Jong99]. Na *programação evolutiva* os indivíduos são representados por máquinas de estados finitos capazes de responder aos estímulos ambientais [Jong99]. Os operadores de variação (essencialmente, os de mutação) afectam a estrutura e o comportamento dos indivíduos. A *programação genética* é a técnica de computação evolutiva mais recente que estende o modelo genético ao espaço de programas [Koza99]. Neste caso os indivíduos não são *strings* codificadas mas sim programas de computador. Portanto, a programação genética permite desenvolver (ou, evoluir) automaticamente programas que resolvem (exacta ou aproximadamente) um dado problema.

É de notar que com o decorrer do tempo, observa-se uma troca intensiva de ideias entre os investigadores que trabalham em cada uma das áreas mencionadas, o que resultou na redução de diferenças entre estes paradigmas. Por isso descrevemos aqui características inerentes a todos os *algoritmos evolutivos*.

O diagrama básico de um algoritmo evolutivo está apresentado na fig. 7.1. Primeiro, deve-se construir uma população inicial de indivíduos. Normalmente, a população inicial é criada pela amostragem aleatória de soluções possíveis. Contudo, é também possível gerar a população obtida com a ajuda de qualquer outro algoritmo ou pelos peritos humanos. A seguir cada solução é avaliada a fim de medir a sua adequabilidade. A partir deste momento o algoritmo entra num ciclo composto pelas operações de variação, avaliação e selecção. No fim de cada iteração (denominada geração) cria-se uma nova população de indivíduos.

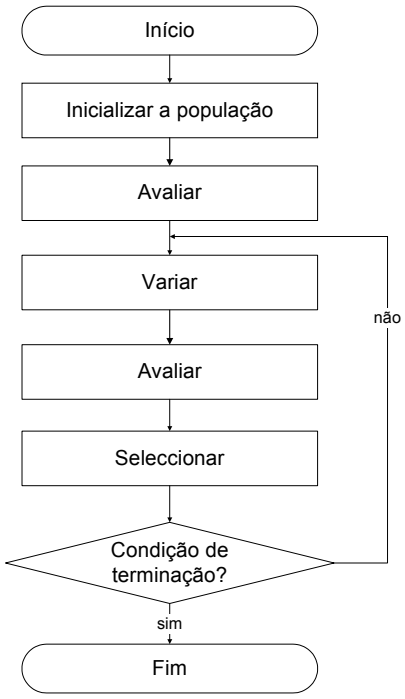


Fig. 7.1. Esquema básico de um EA.

Os operadores de variação (tais como os de mutação e cruzamento) são utilizados para gerar um conjunto novo de indivíduos. Um operador de mutação cria indivíduos novos ao efectuar certas modificações num só indivíduo, enquanto um operador de cruzamento cria indivíduos novos (descendentes) através da combinação de várias partes de dois ou mais indivíduos existentes (pais) [Michalewicz00]. A operação de selecção faz com que os indivíduos mais adequados sobrevivam e formem a geração seguinte. Este processo continua até que alguma condição de terminação seja atingida, tal como a obtenção de uma solução satisfatória, o esgotar do tempo disponível ou do número máximo de gerações permitidas.

Na maioria dos EA o tamanho da população é um valor fixo que é especificado como um parâmetro controlado pelo utilizador. No caso geral, o processo de selecção dos indivíduos que passam para a geração seguinte, é executado uma vez por cada ciclo de algoritmo, permitindo deste modo primeiro criar todos os descendentes necessários e só depois realizar a selecção. Por outro lado, os algoritmos *steady state* invocam o processo de selecção cada vez que seja produzido um descendente a fim de manter o tamanho de população sempre constante.

As vantagens principais dos algoritmos evolutivos são as seguintes [Fogel99]:

- *Simplicidade conceptual*. Como descrevemos no início desta secção, o esquema básico dos algoritmos evolutivos é bastante simples e só é composto pelas fases de inicialização, avaliação, variação e selecção que são aplicadas em ciclo até que a população converge para a solução óptima (caso isto seja possível);

- *Aplicabilidade ampla*. Os algoritmos evolutivos são aplicáveis a qualquer problema que pode ser formulado como o de optimização. Para tal é necessário definir a estrutura de dados para representar as soluções, a função para avaliar a sua adequabilidade, os operadores de variação que permitem gerar soluções novas com base em soluções já existentes, e a função de selecção;

  EA são aplicados com sucesso em áreas práticas diversas tais como síntese lógica [Sklyarov02, Jóźwiak02], criptografia [Nedjah02], pilotagem automática de automóveis [Laumanns02], engenharia mecânica [Alander98], engenharia aeronáutica, indústria nuclear e química [Bäck98], robótica [Nyakoe02], medicina [Fogel99], etc. Uma área de investigação bastante recente é o *hardware evolutivo*. Este termo descreve abordagens diferentes utilizadas para desenvolver circuitos electrónicos com a ajuda de técnicas evolutivas o que permite uma exploração mais alargada do espaço de projecto, conforme representado na fig. 7.2 [Miller98, Thompson99];

- *Os métodos evolutivos podem superar os clássicos em problemas reais*. Muitos problemas de optimização reais possuem características que não se coadunam com os requisitos dos métodos clássicos. Exemplos destas características são restrições não lineares, condições não estacionárias, etc. [Fogel99]. As superfícies de resposta em problemas reais são frequentemente multimodais e os métodos baseados em gradiente convergem rapidamente para um óptimo local. Ao contrário disso, com as técnicas evolutivas é possível levar em conta todas estas características o que resulta em desempenho mais elevado;

- *Potencialidade de hibridação com outros métodos*. Os algoritmos evolutivos possuem uma estrutura que facilita a incorporação relativamente natural de conhecimentos específicos ao problema em causa. Por exemplo, pode-se utilizar operadores de variação específicos, bem como uma função de adequabilidade específica. Para além disso, é possível e razoável combinar

algoritmos evolutivos com as técnicas de optimização tradicionais. Existem várias possibilidades de fazer isto: correr os métodos baseados em gradiente a seguir à realização de pesquisa inicial com um EA; aplicar algoritmos diferentes em simultâneo (por exemplo, incorporar os métodos de pesquisa local no EA [Reeves99]); utilizar métodos rápidos (por exemplo, os algoritmos *greedy*) a fim de gerar a população inicial de indivíduos para o EA, etc.;

- *Possibilidade de implementações paralelas e distribuídas*. É frequentemente possível realizar algumas partes de algoritmos evolutivos (por exemplo, a avaliação de indivíduos, a implementação de operadores de variação) em paralelo. Para além disso, pode-se explorar o paralelismo a nível de população. Neste caso criam-se grupos de indivíduos que se desenvolvem de uma maneira semi-independente, observando-se uma migração lenta de indivíduos entre os grupos [Tomassini99]. É de notar também que os EA possuem paralelismo implícito pois são capazes de encontrar várias soluções de qualidade igual. Isto permite seleccionar a solução mais apropriada para o problema em causa;

- *Robustez às alterações dinâmicas*. Os métodos tradicionais de optimização não são robustos às alterações dinâmicas das condições do problema. Caso ocorra alguma alteração na especificação da tarefa, é frequentemente necessário executar de novo o algoritmo a fim de obter a solução. Ao contrário disso, os algoritmos evolutivos são capazes de adaptar soluções às circunstâncias correntes porque a população de indivíduos já obtidos pode servir de base ao aperfeiçoamento futuro;

- *Capacidade de auto-optimização*. Todas as técnicas de optimização requerem a definição adequada dos valores de variáveis exogéneas (tais como o tamanho de passo, etc.). Contudo, nos algoritmos evolutivos é possível optimizar estes parâmetros durante o processo de pesquisa enquanto os métodos tradicionais utilizam parâmetros definidos *a priori* pelos peritos humanos;

- *Possibilidade de resolver problemas que não possuem soluções conhecidas*. Esta é uma das maiores vantagens dos EA pois estes podem enfrentar problemas para os quais não se conhece solução.

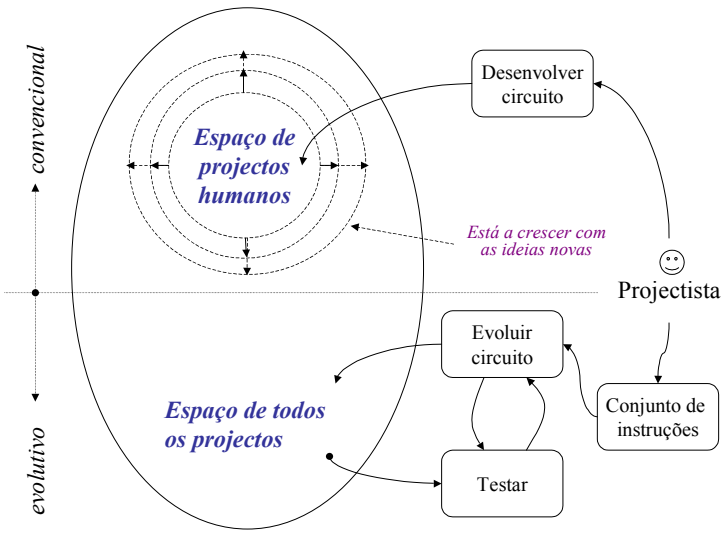


Fig. 7.2. Projecto convencional *vs*. projecto evolutivo.

7.3 Algoritmo evolutivo para o problema do caixeiro viajante

Descrevemos o algoritmo evolutivo que aplicámos para resolver o problema do caixeiro viajante. Para tal é necessário definir a representação de uma solução, a função de adequabilidade, os operadores de mutação e de cruzamento e os critérios de selecção.

7.3.1 Representação

No caso do TSP uma solução potencial é um caminho que começando numa cidade inicial percorre todas as cidades restantes numa determinada ordem. Representámos um caminho possível como um vector de números inteiros em que a cidade na posição i, i=1,...,n-2, é visitada depois da cidade na posição i-1 e antes da cidade na posição i+1. Por exemplo, para o grafo mostrado na fig. 7.3 uma das soluções possíveis está marcada pelos arcos mais grossos e pode ser representada com o vector [0 1 4 2 3].

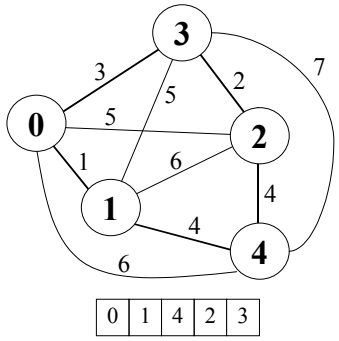


Fig. 7.3. Representação dum caminho possível.

7.3.2 Avaliação

Para o problema do caixeiro viajante a parte de avaliação do algoritmo é muito natural dado que a função de adequabilidade de um caminho corresponde ao seu comprimento. Para o exemplo da fig. 7.3 a adequabilidade da solução [0 1 4 2 3] é 14, e como se pode verificar esta é a solução óptima.

7.3.3 Operadores de variação

Analisemos em detalhe os operadores de variação que aplicámos.

7.3.3.1 Mutação

O operador de mutação selecciona aleatoriamente duas cidades numa solução, e inverte a ordem de todas as cidades que ficam entre as escolhidas. Como resultado, o operador de mutação tenta reparar o caminho que se intersecta a si próprio. O exemplo desta situação está ilustrado na fig. 7.4. Do lado esquerdo mostramos a sequência inicial de cidades. Caso seleccionemos as cidades número *0* e *4* e apliquemos o operador de mutação, obteremos o caminho apresentado do lado direito da fig. 7.4. Neste caso o comprimento do caminho resultante é mais curto do que o do caminho inicial, contudo nem sempre isto acontece.

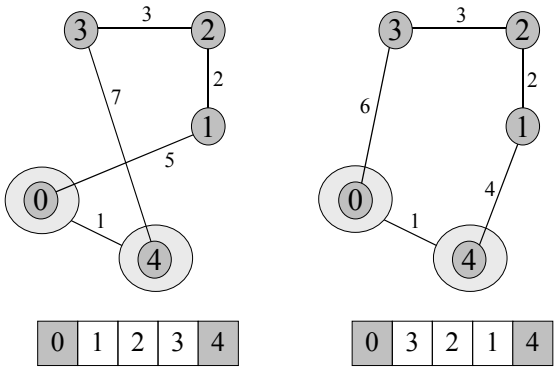


Fig. 7.4. Aplicação do operador de mutação.

7.3.3.2 Cruzamento

Aplicámos o operador de cruzamento PMX (*Partially Matched crossover*) proposto em [Goldberg85], que produz descendentes através da selecção de uma subsequência do caminho de um pai, preservando a ordem e a posição de um número máximo de cidades doutro pai. Este operador envolve dois pais (p_1 e p_2) que produzem dois descendentes (o_1 e o_2). A subsequência de caminho que passa do pai ao filho é definida com a selecção de dois pontos de corte. Inicialmente, os segmentos que se encontram entre os pontos de corte são copiados do pai p_1 para o descendente o_2, e do pai p_2 para o descendente o_1. Estes segmentos definem também uma série de mapeamentos. A seguir todas as cidades que se encontram antes do primeiro ponto de corte e depois do segundo ponto de corte são copiados do pai p_1 para o descendente o_1, e do pai p_2 para o descendente o_2. É de notar que esta operação pode resultar num caminho inválido; por exemplo um descendente pode incluir cidades duplicadas. A fim de ultrapassar esta situação, utiliza-se a série de mapeamentos definida previamente que indica como permutar as cidades que se encontram em conflito.

Por exemplo, para os pais p_1 e p_2 com os pontos de corte marcados pelas linhas verticais:

$$p_1 = [3 \mid 0\ 1\ 2 \mid 4]$$
$$p_2 = [1 \mid 0\ 2\ 4 \mid 3]$$

o operador PMX define a seguinte série de mapeamentos: ($0 \leftrightarrow 0$, $1 \leftrightarrow 2$, $2 \leftrightarrow 4$)

e preenche os segmentos dos descendentes entre os pontos de corte de maneira seguinte:

$$o_1 = [- \mid 0\ 2\ 4 \mid -]$$
$$o_2 = [- \mid 0\ 1\ 2 \mid -]$$

A seguir todas as cidades que ficam antes do primeiro e depois do segundo ponto de corte têm de ser transferidas dos pais para os filhos:

$$o_1 = [3 \mid 0\ 2\ 4 \mid \mathbf{4}]$$
$$o_2 = [\mathbf{1} \mid 0\ 1\ 2 \mid 3]$$

Como se pode ver entramos numa situação de conflito pois ambos os descendentes incluem cidades duplicadas (marcadas acima com os caracteres em negrito), o que está proibido pela definição do problema.

Consideremos o primeiro descendente o_1. Neste caso é preciso substituir a cidade número *4* por qualquer outra cidade válida. Para tal utiliza-se a série de mapeamentos que indica que à cidade *4* corresponde a cidade *2* (2↔4). Contudo, a cidade *2* já está incluída no caminho o_1. Portanto, recorre-se novamente à série de mapeamentos que mostra que à cidade *2* corresponde a *1* (1↔2). Em consequência, incluímos a cidade *1* no descendente o_1.

O mesmo procedimento executa-se para o caso do segundo descendente. Finalmente, obtemos o resultado seguinte:

$$o_1 = [3 \mid 0\ 2\ 4 \mid 1]$$
$$o_2 = [4 \mid 0\ 1\ 2 \mid 3]$$

A aplicação do operador de cruzamento PMX está ilustrada na fig. 7.5. As figuras 7.5a e 7.5b representam os dois pais e as figuras 7.5c e 7.5d mostram os dois descendentes resultantes. Como se pode observar, a adequabilidade de um dos filhos é bastante menor que a de qualquer dos pais, enquanto a do descendente o_2 é significativamente maior.

É de notar que o problema do caixeiro viajante possui a estrutura do "vale grande" do relevo de adequabilidade do espaço de pesquisa [Darwen02]. Isto é devido a uma forte correlação entre a adequabilidade de uma solução e a diferença existente entre esta solução e o óptimo global. Como resultado, todas as soluções boas possuem partes semelhantes. Sendo assim, para este problema o cruzamento é muito eficiente dado que permuta várias partes de soluções entre si.

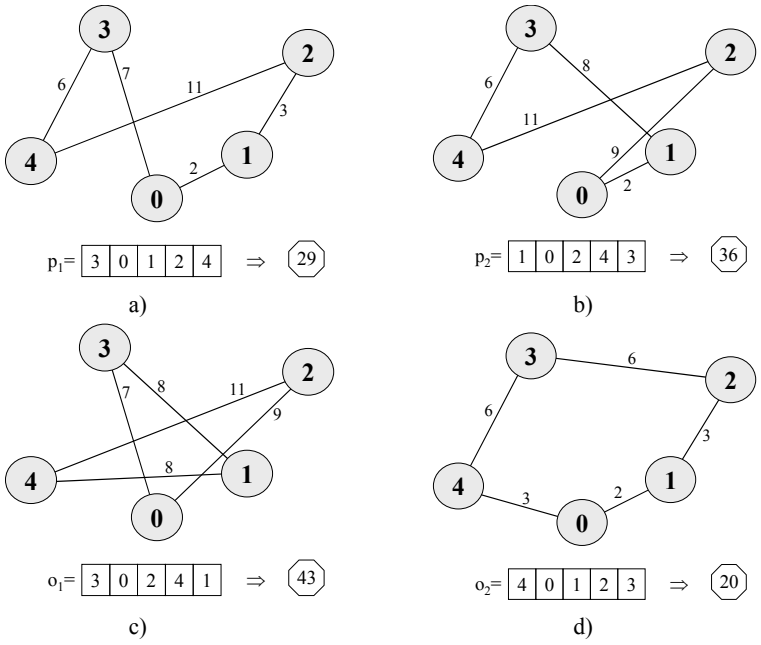


Fig. 7.5. Aplicação do operador de cruzamento PMX.

7.3.4 Selecção

A fim de escolher os pais para a produção de descendentes aplicámos a selecção proporcional à adequabilidade de indivíduos. Para tal utilizamos a abordagem da roda de roleta em que a cada indivíduo se atribui um sector cuja largura é proporcional à adequabilidade deste indivíduo. A roda é "activada" cada vez que se precisa de um pai.

Todos os descendentes criados formam a geração seguinte. Para além disso aplicámos a selecção *elitist* que garante a sobrevivência do melhor indivíduo encontrado durante a pesquisa.

7.4 Implementação do EA em software

O algoritmo descrito foi implementado numa aplicação de software desenvolvida para plataforma Windows em Microsoft Visual C++. A hierarquia de classes que correspondem a todos os elementos principais do EA está ilustrada na fig. 7.6.

A classe *CEvAlg* descreve o algoritmo evolutivo e possui atributos inerentes a qualquer EA, tais como o número de gerações a processar, as probabilidades de mutação e de cruzamento, etc. A classe *CEvAlg* contém dois objectos de classes *CParentPopulation* e *COffspringPopulation*, o primeiro dos quais representa a população paterna e o segundo corresponde à população descendente. Ambas as classes *CParentPopulation* e *COffspringPopulation* são derivadas da classe *CPopulation* que possui atributos e funções comuns a todos os tipos de populações entre os quais há um conjunto de soluções potenciais representadas pelos objectos da classe *CSolution*. A classe *CSolution* inclui a codificação duma solução possível bem como a sua adequabilidade. As distâncias entre as cidades são guardadas num objecto da classe *CMatrix* que é utilizado pelas soluções-candidatas para avaliar a sua adequabilidade.

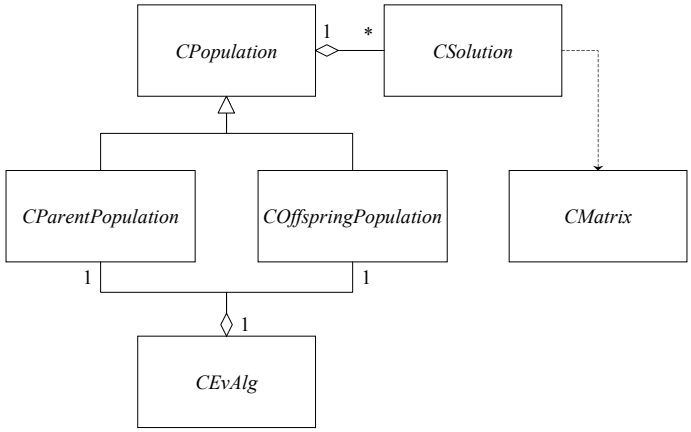


Fig. 7.6. Diagrama de classes.

A ferramenta de software desenvolvida aceita seis parâmetros que podem ser especificados na linha de comando:

```
tsp_pmx.exe problema.tsp <número_gerações tamanho_população prob_cruz prob_mut perc_el>
```

O parâmetro *problema.tsp* é obrigatório e corresponde ao nome de um ficheiro com os dados do problema em formato TSP [TSPLIB]. Este formato é composto por parte de especificação e parte de dados. Um exemplo de um ficheiro em formato TSP está representado na fig. 7.7. A versão corrente da aplicação só suporta o tipo de dados *TSP* (dado que apenas abordamos o TSP simétrico). Na parte de especificação das coordenadas de cidades só suportamos o formato *EUC_2D*. A distância euclidiana entre duas cidades com as coordenadas (x_i,y_i) e (x_j,y_j) é calculada de forma seguinte:

$$(int)(\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} + 0.5)$$

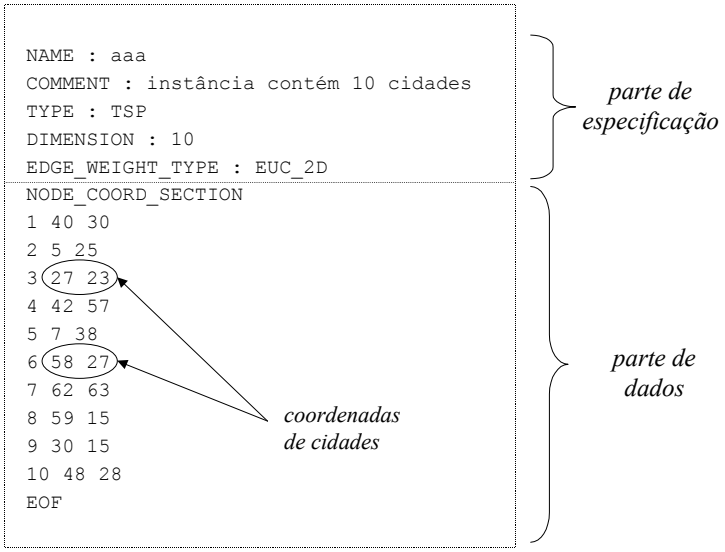


Fig. 7.7. Exemplo de um ficheiro em formato TSP.

Os parâmetros restantes da linha de comando são opcionais. Caso não sejam especificados serão utilizados os valores por defeito. O significado de cada um destes parâmetros está explicado na tabela 7.1. Depois de iniciada a aplicação de software executa os passos ilustrados na fig. 7.8.

Tabela 7.1. Significado dos parâmetros da linha de comando.

Parâmetro	Significado	Valor por defeito
número_gerações	número de gerações a processar	100
tamanho_população	número de indivíduos numa população	20
prob_cruz	probabilidade de execução da operação de cruzamento	50%
prob_mut	probabilidade de execução da operação de mutação	20%
perc_el	percentagem de aplicação da regra *elitist*	10%

Com o programa desenvolvido foi realizada uma série de experiências com instâncias de teste disponíveis na TSPLIB [TSPLIB]. As experiências foram efectuadas com diferentes probabilidades de cruzamento (nomeadamente, 10%, 25%, 50% e 100%). Para todas as instâncias o tamanho da população foi de 20 indivíduos e foram realizadas 1000 gerações. A probabilidade de mutação foi de 10% (no caso da probabilidade de cruzamento de 100% o operador de mutação não foi aplicado).

Os resultados são apresentados na tabela 7.2. A primeira coluna contém os nomes de várias instâncias. Os números incluídos em nomes especificam a quantidade de cidades na instância respectiva. As colunas t_{total} guardam o tempo total (em segundos) de resolução do problema num PIII/800MHz/256MB com o sistema operativo Windows2000. As colunas t_{cros} indicam o tempo (em segundos) despendido a efectuar as operações de cruzamento. E finalmente, as colunas $\%_{cros}$ contêm a percentagem do tempo de realização do cruzamento comparando-o com o tempo total de execução do algoritmo.

- Ler ficheiro *problema.tsp* e construir um objecto da classe *CMatrix* que contém as distâncias entre todas as cidades;
- Criar um objecto da classe *CEvAlg*;
- Criar um objecto da classe *CParentPopulation*;
- Gerar *tamanho_população* de soluções aleatórias (mas correctas), avaliá-las e inicializar o objecto da classe *CParentPopulation*.
- Criar um objecto da classe *COffspringPopulation*;
- for (unsigned *i* = 0; *i* < *número_gerações*; *i*++)

  {
  - Criar (*tamanho_população* * *prob_cruz*) de descendentes aplicando o operador de cruzamento;
  - Criar (*tamanho_população* * *prob_mut*) de descendentes aplicando o operador de mutação;
  - Preencher os descendentes restantes copiando-os da população paterna;
  - Avaliar os descendentes;
  - Substituir os (*tamanho_população* * *perc_el*) de descendentes piores pelo mesmo número de pais melhores;
  - Copiar a população descendente para a população paterna para que os descendentes correntes sirvam de pais para os descendentes futuros).

  }
- Gravar os resultados num ficheiro;
- Destruir todos os objectos.

Fig. 7.8. Passos executados pela aplicação de software para resolver o problema de TSP.

Tabela 7.2. Resultados das experiências em software.

Instância	10%			25%			50%			100%		
	t_{total}	t_{cros}	$\%_{cros}$	t_{total}	t_{cros}	$\%_{cros}$	t_{total}	t_{cros}	$\%_{cros}$	t_{total}	t_{cros}	$\%_{cros}$
a280	4.99	0.72	14.4	5.88	1.49	25.3	**8.39**	**3.66**	**43.6**	12.51	7.22	57.7
berlin52	**1.08**	**0.71**	**65.7**	1.27	0.32	25.2	1.79	0.75	41.9	2.64	1.42	53.8
bier127	2.36	0.36	15.2	2.75	0.68	24.7	**3.92**	**1.68**	**42.9**	5.84	3.29	56.3
d657	12.82	2.45	19.1	**15.18**	**4.62**	**30.4**	23.33	11.92	51.1	35.66	23.21	65.1
eil51	1.06	0.16	15.1	**1.22**	**0.31**	**25.4**	1.76	0.74	39.2	2.58	1.38	53.5
fl417	7.77	1.34	17.2	**9.30**	**2.62**	**28.2**	13.56	6.36	46.9	20.50	12.59	61.4
rat575	10.88	1.89	17.4	12.97	3.78	29.1	**19.46**	**9.53**	**48.9**	30.74	19.39	63.1
u724	**14.27**	**2.69**	**18.9**	17.02	5.36	31.5	25.60	13.19	51.5	40.55	26.74	65.9
vm1084	22.83	5.01	21.9	**28.09**	**9.69**	**34.5**	43.63	24.13	55.3	71.12	49.20	69.2

Os melhores resultados em termos do desempenho do algoritmo foram atingidos com as probabilidades de cruzamento de 25% e 50%. As linhas que correspondem à melhor solução estão destacadas na tabela 7.2 com os caracteres em negrito. Por exemplo, para a instância *a280* o melhor resultado foi obtido com a probabilidade de cruzamento de 50%, para a instância *berlin52* – com a probabilidade de 10%, etc. Como se pode ver da tabela 7.2, uma percentagem significativa do tempo do processador é despendida a efectuar a operação de cruzamento. Obviamente, o valor $\%_{cros}$ é maior para a probabilidade de cruzamento de 100% (contudo, em algoritmos práticos a probabilidade de cruzamento é sempre bastante menor do que 100%). Mas mesmo para as probabilidades de cruzamento "melhores" (em termos dos resultados obtidos) o valor $\%_{cros}$ varia de 19% a 65%.

7.5 Implementação parcial em FPGA

Os algoritmos evolutivos possuem um certo grau de paralelismo em termos de pesquisa no espaço de soluções. Contudo, quando aplicados a problemas complexos, os EA requerem que a população seja grande e necessitam de um grande número de gerações, o que os torna bastante lentos. Têm sido efectuadas várias tentativas de acelerar algoritmos evolutivos com a ajuda de hardware reconfigurável [Salami02a, Graham96, Aporntewan01, Shackleford01, Graham95]. Os autores respectivos comunicam a obtenção de resultados impressionantes em comparação com as implementações em software. Contudo, as realizações existentes baseadas em hardware dedicado são muito complexas e requerem bastantes recursos, ou, ao contrário, muito simplificadas o que não corresponde às necessidades de aplicações práticas. É por isso que decidimos analisar que parte do EA para o problema do caixeiro viajante é a mais crítica em termos do tempo que consome, e implementar só esta parte em FPGA.

Os resultados apresentados na secção 7.4 mostraram que a parte mais crítica do algoritmo considerado é a operação de cruzamento. Portanto, o aumento da eficiência desta operação vai influenciar significativamente o desempenho de todo o algoritmo. Por isso sugerimos a implementação da operação de cruzamento em FPGA [Skliarova02c].

A arquitectura do circuito respectivo está representada na fig. 7.9. Esta inclui uma unidade de controlo central que activa, na ordem necessária todos os passos da operação de cruzamento. A versão da arquitectura implementada é capaz de processar caminhos compostos por 1024 cidades no máximo. Por isso existem quatro blocos de memória do tamanho $2^{10}\times10$ bits que servem para guardar os dois caminhos-pais (*Pai 1* e *Pai 2*) e os dois caminhos-descendentes resultantes (*Filho 1* e *Filho 2*). Um caminho é representado de maneira seguinte. Em cada endereço de memória está escrito o número duma cidade. A cidade na posição de memória *i* é visitada depois da cidade que se encontra no endereço *i-1* e antes da cidade que está na posição *i+1*. Uma das aplicações práticas possíveis do circuito proposto está descrita em [Skliarova02c] para a qual umas 64-128 cidades são suficientes, possibilitando diminuir significativamente o tamanho dos blocos de memória.

Os pontos de corte utilizados no cruzamento são escolhidos aleatoriamente pela aplicação de software e escritos em dois registos especiais de 10 bits (*Primeiro ponto de corte* e *Segundo ponto de corte* na fig. 7.9). O registo *Max* guarda o comprimento actual do caminho. De acordo com este valor a unidade de controlo só vai forçar o processamento da área adequada dos blocos de memória

que contêm os dois pais e os dois descendentes. Isto permite acelerar o cruzamento de caminhos curtos.

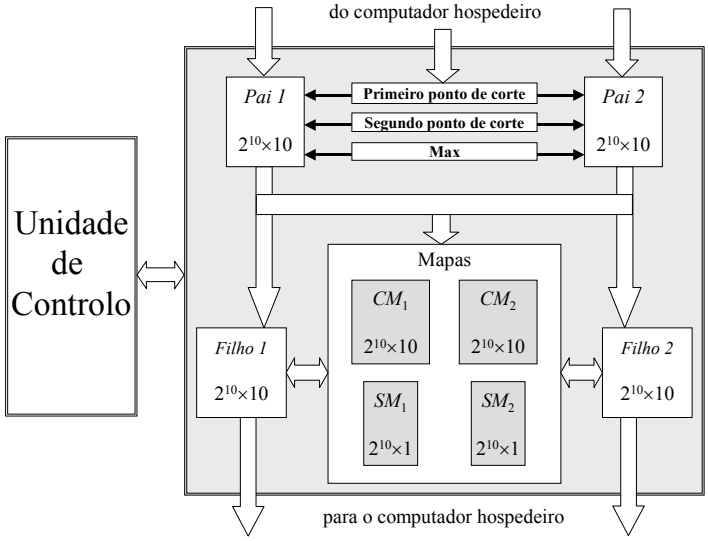


Fig. 7.9. Arquitectura proposta para a realização da operação de cruzamento PMX.

Para além disso há quatro blocos de memória adicionais que ajudam na realização do cruzamento. Estes blocos são CM_1 e CM_2 do tamanho de $2^{10}\times10$ bits (vamos referenciá-los por *mapas complexos*), e SM_1 e SM_2 do tamanho de $2^{10}\times1$ bit (referenciados por *mapas simples*).

A fim de executar o cruzamento PMX é necessário realizar a seguinte sequência de operações. Primeiro, os valores dos pontos de corte e do número de cidades para uma dada instância do problema devem ser carregados na FPGA. Depois, os caminhos-pais são transferidos do computador hospedeiro para os blocos de memória *Pai 1* e *Pai 2*. Cada vez que o número duma cidade se escreve num bloco de memória dos pais, a posição do mapa simples respectivo com o mesmo endereço inicializa-se com o valor *0* (o que permite efectuar o *reset* dos mapas simples).

A seguir, os segmentos entre os pontos de corte devem ser transferidos do *Pai 1* para o *Filho 2* e do *Pai 2* para o *Filho 1*. Cada vez que se transfere a cidade c_1 do bloco *Pai 1* para o bloco *Filho 2*, o valor *1* deve ser escrito no endereço c_1 do mapa simples SM_2. A mesma coisa acontece com o outro pai, i.e. quando copiamos a cidade c_2 do bloco *Pai 2* para o bloco *Filho 1*, o valor *1* deve também ser escrito no endereço c_2 do mapa simples SM_1. Ao mesmo tempo o valor c_1 é guardado no endereço c_2 no mapa complexo CM_1, e o valor c_2 é guardado no endereço c_1 no mapa complexo CM_2. Para o exemplo considerado na secção 7.3.3.2, todos os blocos devem ser preenchidos da maneira apresentada na fig. 7.10.

O passo seguinte da operação de cruzamento pressupõe que todas as cidades que se encontram antes do primeiro ponto de corte e depois do segundo ponto de corte devem ser copiadas do bloco *Pai 1* para o *Filho 1* e do bloco *Pai 2* para o *Filho 2*. Para além disso todos os conflitos têm de ser resolvidos.

Para isso aplicámos a estratégia apresentada na fig. 7.11. Analisemos aqui o caso do preenchimento do bloco *Filho 1* antes do primeiro ponto de corte, pois para os casos restantes os passos a realizar são semelhantes. Primeiro, o número da cidade *c* é lido do bloco *Pai 1*. Caso no endereço *c* do

bloco SM_1 esteja escrito o valor *0*, isto indica que a cidade *c* pode seguramente ser copiada para o bloco *Filho 1*. Caso contrário, se no endereço *c* do bloco SM_1 estiver escrito o valor *1*, isto significa que a cidade *c* já foi incluída no caminho *Filho 1*. Isso significa que esta deve ser substituída por outra cidade. Para tal utiliza-se o mapa complexo CM_1 da maneira ilustrada na fig. 7.11.

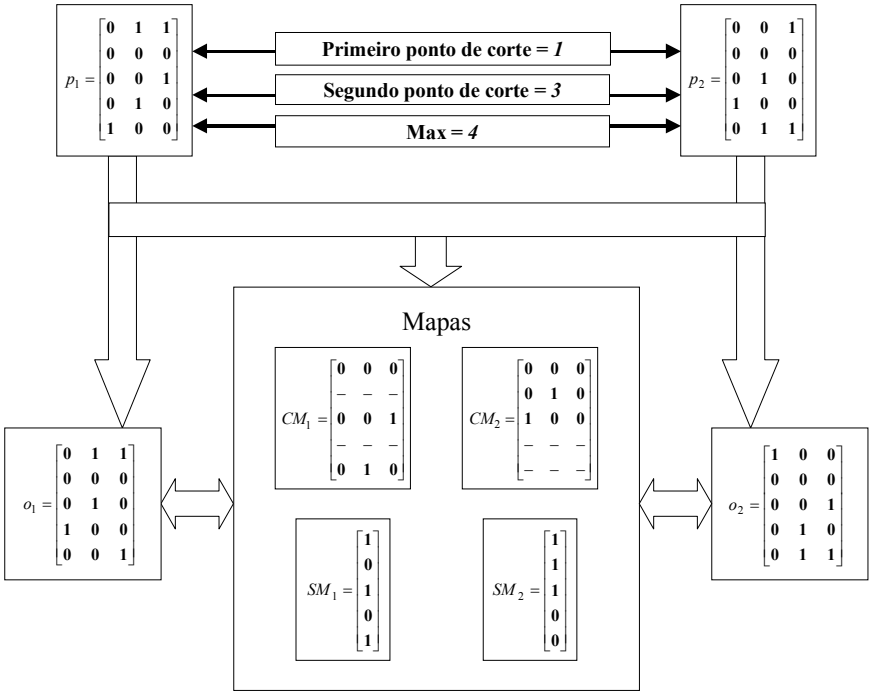


Fig. 7.10. Conteúdo dos blocos de memória e dos registos para o exemplo considerado na secção 7.3.3.2.

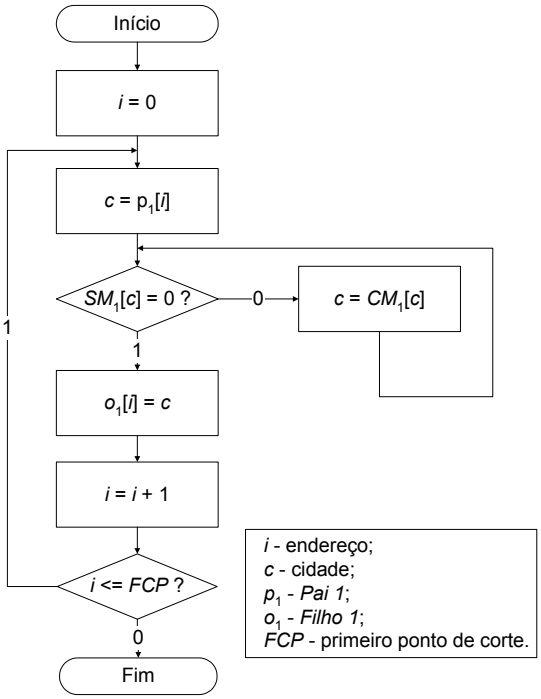


Fig. 7.11. Algoritmo que controla o preenchimento do bloco *Filho 1* antes do primeiro ponto de corte.

Para compor o segundo caminho-descendente são executadas as mesmas operações com a única diferença de que são utilizados os mapas SM_2 e CM_2. Por fim, os caminhos-descendentes resultantes são transferidos para o computador hospedeiro.

7.6 Avaliação dos resultados

Para as experiências foi utilizada a placa com interface PCI ADM-XRC da Alpha Data [Alpha] que contém uma FPGA da família Virtex-EM da Xilinx (nomeadamente, a XCV812E) [Xilinx_025]. Esta FPGA incorpora uma memória adicional de blocos *SelectRAM* organizados em colunas e inseridos em cada quarta coluna de CLBs. No total há mais de um milhão de bits disponíveis que complementam os 294 Kbits de RAM distribuída. Por conseguinte esta FPGA serve muito bem para a arquitectura proposta porque podemos aproveitar a grande quantidade da memória disponível para guardar os caminhos-pais e os descendentes bem como os mapas necessários. A interacção com a FPGA é feita com a ajuda da biblioteca de interface com a placa ADM-XRC que suporta a inicialização e configuração da FPGA, transferência de dados, processamento de interrupções e erros, gestão de relógio, etc.

A tabela 7.3 contém informação acerca da área ocupada pelo circuito implementado e da sua frequência de relógio. A área está expressa em número de *slices* da Virtex, sendo cada CLB composto por dois *slices*. Entre parêntesis está indicada a percentagem dos recursos da XCV812E que o circuito consome.

Tabela 7.3. Parâmetros do circuito implementado.

Área (em *slices*)	**Número dos blocos de RAM**	**Frequência de relógio máxima (MHz)**
149 (1%)	20 (7%)	102.659

A comparação entre as implementações baseadas em software e em hardware do operador de cruzamento PMX está apresentada na tabela 7.4. A primeira coluna contém o número de cidades no problema respectivo. A coluna t_{soft} guarda o tempo de realização do cruzamento em software. A versão de software baseia-se na linguagem C++ e foi executada num PIII/800MHz/256MB com o sistema operativo Windows2000. A coluna t_{hard} contém o tempo de realização do cruzamento em hardware. A versão de hardware foi executada em FPGA XCV812E sob a frequência de relógio de 40 MHz. Para facilitar a comparação, os pontos de corte foram seleccionados iguais em ambas as implementações. A aceleração conseguida é dada pela expressão t_{soft}/t_{hard}. Os resultados das experiências mostraram que o cruzamento PMX é executado em FPGA 10-50 vezes mais rapidamente do que em software.

É de notar que o tempo do cruzamento realizado em FPGA depende fortemente dos pontos de corte seleccionados. Isto explica-se pelo facto da primeira parte do cruzamento PMX (i.e. a troca dos segmentos entre os pontos de corte) ser executada em paralelo para ambos os pares pai-descendente. Para além disso, esta parte só envolve a leitura simples do bloco de memória que guarda o caminho-pai, e a escrita sequencial para o bloco de memória correspondente ao caminho-

descendente, sem executar qualquer outra operação. Como resultado, esta fase é muito rápida. Por outro lado, a segunda parte do cruzamento (i.e. o preenchimento dos descendentes antes do primeiro e depois do segundo pontos de corte) é executada sequencialmente para cada par pai-descendente. Para além da realização de operações de leitura/escrita na memória, devem ser efectuadas algumas verificações que garantem a construção de caminhos válidos. Em consequência, quanto maior for a distância entre os pontos de corte, mais rápido será concluído o cruzamento em FPGA. A mesma tendência observa-se na versão em software embora a primeira parte do cruzamento seja realizada de modo sequencial.

A aceleração conseguida em FPGA em comparação com a implementação em software explica-se pelas razões seguintes. Primeiro, foi aplicada a técnica de processamento paralelo, i.e. a parte da operação de cruzamento PMX é executada em paralelo em FPGA. Segundo, a organização dos blocos de memória é optimizada para os tamanhos de dados necessários. Achamos que é possível conseguir uma aceleração maior através da realização da segunda parte do cruzamento também em paralelo. Para tal será necessário introduzir na unidade de controlo mais um bloco que vai executar o algoritmo da fig. 7.11 ligeiramente modificado (os índices dos blocos de memória serão diferentes), preenchendo deste modo o segundo descendente em paralelo com o primeiro.

Tabela 7.4. Comparação das implementações baseadas em software e em hardware do operador de cruzamento PMX.

número de cidades	t_{soft} (*ms*)	t_{hard} (*ms*)	**aceleração**
280	1.0896	0.08443	12.9
52	0.4293	0.01872	22.9
127	0.6203	0.04040	15.4
657	3.838	0.19486	19.7
51	0.2151	0.01806	11.9
417	2.0605	0.12524	16.5
575	2.7703	0.17199	16.1
724	10.2145	0.20443	49.9

É de notar que uma outra parte de EA que, para muitas aplicações, é o entrave, é a tarefa de avaliação de indivíduos. Para aliviar este efeito usam-se métodos que estimam o valor de adequabilidade de uma solução em vez de o calcular exactamente [Salami02b].

7.7 Trabalho relacionado

A eficiência de implementação de algoritmos heurísticos em hardware reconfigurável foi explorada por vários investigadores [Graham95, Graham96, Abramson97, Abramson98a, Abramson98b, Aporntewan01, Shackleford01, Salami02a]. Descrevemos a seguir dois trabalhos que lidaram com o problema do caixeiro viajante.

7.7.1 *Graham et al.*

Graham et al. mapearam um algoritmo genético de solução de TSP simétrico para o sistema Splash-2 (ver secção 3.4.1.2) [Graham95, Graham96]. A representação de um caminho potencial e o operador de mutação aplicado são semelhantes aos nossos mas o operador de cruzamento empregue é diferente. Para gerar dois caminhos-descendentes seleccionam-se dois caminhos-pais e nestes escolhe-se aleatoriamente um ponto de corte. De seguida, o cabeçalho do pai *1* serve de cabeçalho para o descendente *1* e o cabeçalho do pai *2* transforma-se no cabeçalho do descendente *2*. A parte restante do descendente *1* é formada extraindo (pela sua ordem relativa) as cidades do pai *2* que não se encontram no cabeçalho do pai *1*. De maneira semelhante é completada a construção do descendente *2*. A probabilidade de cruzamento varia de 10% a 60%, a probabilidade da mutação é de 10%.

O sistema Splash-2 é programado utilizando VHDL. Para a síntese e implementação do circuito foram empregues as ferramentas comerciais. A população inicial é gerada e fornecida pelo processador hospedeiro. Os blocos de memória associados com as FPGAs guardam ambas as gerações (corrente e nova), os comprimentos de todos os caminhos e a matriz de distâncias entre as cidades. Os candidatos a reprodução são seleccionados com a ajuda do método da roda de roleta (ver secção 7.3.4).

Para implementar este algoritmo foram utilizadas 4 FPGAs, a primeira das quais executa a selecção de soluções para cruzamento, a segunda efectua o cruzamento com a probabilidade indicada, a terceira calcula a adequabilidade dos caminhos recém-gerados e realiza a mutação, a quarta grava as soluções novas na sua memória e determina as soluções melhores e piores. O ciclo que envolve todas as quatro FPGAs é repetido até que seja completada a formação da população nova. De seguida, esta é copiada para as memórias das duas primeiras FPGAs. A execução termina logo que se atinja o número de gerações desejado. A frequência máxima de funcionamento é de 11 MHz.

A implementação em hardware reconfigurável foi comparada com uma aplicação de software executada numa HP PA-RISC a 125 MHz. Os resultados das experiências mostraram que a aceleração atingida varia de 7 a 10 vezes (não tomando em conta o tempo de configuração do sistema). Em [Graham95] foi também proposto um modelo de computação paralela que consiste em criar várias unidades de processamento e efectuar a pesquisa simultaneamente em todas as unidades com a migração periódica de soluções entre estas. Desta maneira é possível encontrar soluções boas mais rapidamente.

7.7.2 *Abramson et al.*

Abramson et al. trabalharam também na aceleração de algoritmos de solução de TSP com a ajuda de sistemas reconfiguráveis específicos à aplicação. Assim em [Abramson97] descreve-se uma arquitectura que implementa um algoritmo baseado em *simulated annealing* (ver secção 4.4.3.2). Foi adoptada uma representação baseada em listas ligadas dinâmicas aninhadas, onde cada lista aninhada contém um número variável de elementos. Os autores [Abramson97] declaram que com a estrutura de dados proposta é possível representar uma ampla gama de problemas de optimização combinatória.

Os vários algoritmos de pesquisa local distinguem-se pela maneira de gerar soluções potenciais na vizinhança da solução corrente e pelas regras de aceitação/rejeição destas soluções novas. Na sequência de uma análise de muitos problemas foram identificados quatro operadores-geradores de soluções novas que são aplicados às estruturas de dados adoptadas. Estes operadores são os seguintes:

- mover um elemento de uma lista para o fim de outra lista;
- trocar posições de dois elementos numa ou várias listas;
- inverter a sequência entre dois elementos numa lista;
- reposicionar um elemento numa lista.

De seguida foi criado um *template* arquitectural composto por blocos capazes de guardar a solução corrente; aplicar nesta uma das transformações mencionadas acima a fim de gerar a solução nova; calcular a diferença em custo entre ambas as soluções e decidir se é necessário actualizar a solução corrente. Esta arquitectura optimizada para o problema de TSP foi implementada na placa AP4 da Aptix que contém uma série de FPGAs XC4010 da Xilinx. O circuito foi partilhado entre duas FPGAs. Para acelerar as computações é necessário guardar múltiplas cópias da matriz de distâncias entre as cidades e da solução, o que requer consumos significativos de memória. Por isso as FPGAs foram interligadas com um conjunto de módulos de memória estática de 5×128k×8 bits cada. Sendo assim, o sistema é capaz de resolver problemas que incluem no máximo 256 cidades.

Os resultados apresentados em [Abramson97, Abramson98a] são baseados na simulação em software (a frequência de funcionamento do circuito assumida foi de cerca de 3 MHz). O sistema atinge uma aceleração de 37 vezes em comparação com a implementação do mesmo algoritmo em software (RS6000/590/66MHz). Contudo, não fica claro como o sistema é configurado e qual a percentagem do tempo de configuração e de comunicação (sempre que a temperatura é decrementada, a tabela exponencial guardada num bloco de memória deve ser reprogramada pelo computador hospedeiro) no tempo de execução total.

7.8 Conclusões

Neste capítulo apresentámos uma descrição breve de algoritmos evolutivos e mostrámos que estes podem ser eficientemente utilizados para a resolução de problemas de optimização combinatória. Como um estudo de caso, analisámos o problema do caixeiro viajante.

Para além disso apresentámos a implementação de um dos operadores genéticos (nomeadamente, o de cruzamento) com base em hardware reconfigurável. O circuito proposto é bastante simples, consome poucos recursos da FPGA e é capaz de funcionar com a frequência de 100 MHz. Os resultados das experiências mostraram que a implementação do operador de cruzamento em FPGA pode aumentar o desempenho global do algoritmo.

Existem várias ideias interessantes para a investigação futura. Por exemplo, em [Merz97, Bhatia94] foi mostrado que a combinação de algoritmos evolutivos e de pesquisa local permite obter soluções de melhor qualidade, especialmente para problemas de grande dimensão. Contudo, a pesquisa local consome muito tempo de computação, o que torna a ideia da sua implementação em hardware reconfigurável bastante promissora.

8 Conclusões

Sumário

Este capítulo conclui a tese com a apresentação de um sumário das contribuições principais resultantes do trabalho desenvolvido. São também discutidas algumas ideias para a investigação futura.

A computação reconfigurável é uma tecnologia muito promissora. Ao personalizar o hardware de acordo com as características de uma dada aplicação é possível atingir um desempenho elevado e uma utilização eficiente de recursos. Nesta tese, descrevemos a implementação em hardware reconfigurável de uma série de arquitecturas destinadas a resolver problemas de optimização combinatória, estendendo deste modo o domínio de aplicação da computação reconfigurável. O trabalho realizado é também útil como um estudo de caso progressivo que ilustra como as aplicações são projectadas e implementadas em hardware reconfigurável. Essencialmente, foram exploradas as duas metodologias de projecto seguintes:

1) *Arquitecturas reconfiguráveis orientadas para o domínio*. Esta parte englobou actividades tais como identificação de um núcleo de problemas combinatórios e concepção de arquitecturas capazes de os resolver. Para tal foi proposta a técnica de projecto baseada em *hardware templates* e foram desenvolvidas ferramentas de apoio à realização de reconfiguração dinâmica parcial dos mesmos. Uma das questões importantes a resolver foi a de como processar instâncias de problemas que excedem os recursos de hardware disponíveis.
2) *Arquitecturas reconfiguráveis orientadas para a aplicação*. Nesta parte foi investigada a adequabilidade de algoritmos relativamente complexos, tais como os de retrocesso e algoritmos genéticos, para implementações em hardware reconfigurável. Para tal foram escolhidos dois problemas combinatórios particulares que são a satisfação booleana e o problema do caixeiro viajante. Na concepção das arquitecturas relevantes, o desafio principal era evitar completamente a compilação de hardware específica à instância.

8.1 Contribuições

As principais contribuições deste trabalho são as seguintes:

- *Arquitectura de um processador combinatório reconfigurável.* Esta tese está focada no desenvolvimento de arquitecturas reconfiguráveis para problemas de optimização combinatória. Estes problemas possuem inúmeras aplicações práticas, portanto a sua aceleração com a ajuda de hardware reconfigurável é capaz de melhorar o desempenho destas aplicações. Propusemos a arquitectura de um processador combinatório reconfigurável destinado a resolver *vários* problemas combinatórios formulados sobre matrizes discretas. A abordagem adoptada recorre ao uso de modelos (*hardware templates*) que permitem que as mesmas unidades de execução e de controlo sejam utilizadas para uma variedade de problemas e que a sua arquitectura não precise de ser modificada de tarefa a tarefa. Neste caso, é apenas necessário configurar as operações computacionais básicas e os algoritmos de controlo respectivos para cada problema combinatório.
- *Reconfiguração dinâmica parcial de dispositivos programáveis estaticamente*. Para suportar a capacidade de reprogramação do processador combinatório foi proposta a técnica de desenvolvimento baseada em *hardware templates*. A ideia principal dum HT consiste em construir uma unidade computacional parametrizável que inclua componentes modificáveis e fixos com as ligações fixas entre eles. A personalização da unidade faz-se ao configurar os

componentes com a funcionalidade modificável sendo o número destes componentes minimizado. Isto permite utilizar a reconfiguração parcial e reduz substancialmente o *overhead* de reconfiguração. Os HT empregues são baseados em RAM de tal modo que a personalização da funcionalidade do processador combinatório é atingida através da reprogramação dos blocos de RAM relevantes. Consequentemente, o processador pode ser construído com base em FPGAs reprogramáveis estaticamente que incluem células de memória distribuídas (LUTs) ou blocos de memória embutidos.

- *Síntese da unidade de controlo reprogramável*. A unidade de controlo do processador combinatório foi modelada por uma máquina de estados finitos com o comportamento modificável baseada em RAM. Para realizar a reconfiguração da unidade de controlo foi desenvolvida uma ferramenta de software que sintetiza automaticamente o conteúdo dos blocos de memória relevantes a partir duma especificação comportamental. É importante notar que a estrutura da unidade de controlo proposta permite a implementação de *qualquer* algoritmo de controlo que satisfaça as restrições impostas (número máximo de estados, condições lógicas, etc.).

- *Modelo de computação.* A fim de atenuar o problema de capacidade lógica limitada do dispositivo reconfigurável foi sugerido um modelo de computação que permite que a solução de um problema seja partilhada entre uma aplicação de software e o hardware reconfigurável. Este modelo conta com o facto de que a técnica comum de resolução de problemas combinatórios se baseia na construção de uma árvore de pesquisa. No processo de construção da árvore de pesquisa aplicam-se vários métodos de redução e de decomposição que fazem com que as dimensões iniciais da matriz sejam gradualmente reduzidas. Cada problema é inicialmente processado por uma aplicação de software que realiza o mesmo algoritmo que o processador combinatório reconfigurável. Logo que as dimensões da matriz intermédia satisfaçam as restrições de hardware impostas esta é transferida para a FPGA. Se o hardware reconfigurável encontrar a solução, o problema está resolvido e o resultado será despachado para o computador hospedeiro. Se o ramo considerado da árvore de pesquisa não permitir encontrar a solução, o controlo regressará à aplicação de software que vai continuar a percorrer a árvore de pesquisa até atingir um outro ponto em que a matriz intermédia satisfaz as restrições impostas.

- *Aceleração da satisfação booleana.* Como um estudo de caso, foi desenvolvido um circuito capaz de resolver o problema de satisfação booleana. Para tal foi construído um *hardware template*, que possui uma estrutura predefinida que pode ser reutilizada para várias instâncias do problema. Implementámos duas versões do circuito, a primeira das quais executa um algoritmo de retrocesso (derivado do de Davis-Putnam) e a segunda corresponde a um algoritmo híbrido (que combina o primeiro método com o algoritmo baseado em intervalos). A técnica proposta elimina totalmente o passo de compilação de hardware e permite processar problemas cujas dimensões excedem os recursos de hardware disponíveis. Como resultado conseguimos obter uma aceleração bastante significativa para algumas instâncias de SAT em comparação com a aplicação de software GRASP. A solução proposta compara favoravelmente com outras arquitecturas reconfiguráveis destinadas a acelerar a solução do problema de SAT documentadas na literatura.

- *Análise de sistemas reconfiguráveis existentes destinados a acelerar a solução do problema de SAT*. Efectuamos uma análise detalhada de vários sistemas desenvolvidos no âmbito da investigação académica que é bastante útil para a concepção de sistemas futuros. Uma das conclusões a que chegámos é que as acelerações conseguidas com as arquitecturas reconfiguráveis comparando-as com a solução baseada em software, só são significativas para instâncias difíceis de SAT, para as quais as técnicas de optimização propostas e implementadas em aplicações de software são pouco eficientes. Para além disso, o desempenho dos programas resolutores de SAT progride muito rapidamente (como foi demonstrado pelos resultados das duas últimas competições de programas resolutores de SAT decorridas em anos consecutivos). Estes factos implicam que é necessário explorar técnicas algorítmicas e arquitecturais inovadoras para manter os sistemas reconfiguráveis competitivos face às implementações em software.
- *Aceleração de algoritmos evolutivos.* Foi explorada a adequabilidade de uso de hardware reconfigurável para a solução do problema do caixeiro viajante com a ajuda de um algoritmo genético. Os algoritmos genéticos, quando aplicados a problemas complexos, requerem que a população seja grande e necessitam de um grande número de gerações, o que os torna bastante lentos. Têm sido efectuadas várias tentativas de acelerar estes algoritmos com a ajuda de hardware reconfigurável. Contudo, as realizações existentes baseadas em hardware dedicado são muito complexas e requerem bastantes recursos, ou, ao contrário, muito simplificadas o que não corresponde às necessidades de aplicações práticas. É por isso que decidimos analisar que parte do algoritmo considerado para o problema do caixeiro viajante é a mais crítica em termos do tempo que consome, e implementar só esta parte em FPGA. Ao revelar que a operação mais crítica é a de cruzamento propusemos uma arquitectura (baseada num *hardware template*) capaz de a realizar. Os resultados das experiências mostraram que a execução do cruzamento em FPGA pode contribuir para o aumento do desempenho de todo o algoritmo.
- *Extensão do domínio de aplicação da computação reconfigurável*. As aplicações típicas da computação reconfigurável envolvem algoritmos de controlo bastante simples. Neste trabalho, implementámos uma série de algoritmos complexos que suportam propriedades como recursividade e ordenação de variantes a explorar. Isto demonstra a viabilidade de fluxos de controlo complexos na computação reconfigurável.

8.2 Discussão e trabalho futuro

Apesar dos contributos descritos, o trabalho desenvolvido caracteriza-se também por uma série de questões pouco estudadas, deixando deste modo pistas para investigação futura. A seguir são sumariadas algumas ideias que seria interessante explorar.

- A eficiência do método proposto de partição de solução de um problema entre software e hardware reconfigurável depende das características da instância respectiva. Portanto, é necessário investigar outras vias de ultrapassagem do problema de capacidade lógica limitada. Uma solução interessante foi reportada em [Reis02] para o caso específico do problema de SAT que se baseia no esquema de hardware virtual.

- O desempenho dos circuitos de solução de SAT implementados só é considerável para instâncias muito difíceis. Contudo existem várias ideias de como lidar com esta situação. Uma das possibilidades é implementar o retrocesso não cronológico em hardware dado que este permitirá descartar grandes porções do espaço de pesquisa. Uma outra extensão óbvia à investigação consiste em analisar outros algoritmos completos de solução de SAT dado que alguns métodos superam os outros para certas classes de problemas. Na execução desta parte do trabalho restringimos a nossa atenção ao algoritmo de Davis-Putnam porque este é um dos métodos mais conhecidos e utilizados e possui muitos aperfeiçoamentos potenciais a explorar. Uma alternativa aos algoritmos completos são os incompletos. Assim, os métodos de pesquisa local são de uma implementação relativamente fácil, e embora não sejam capazes de provar a inexistência da solução, apresentam interesse para algumas classes de aplicações. Uma combinação de algoritmos exactos com os aproximados pode também ser útil. Por exemplo, ao gerar uma atribuição parcial de valores às variáveis com um algoritmo de retrocesso, é possível aplicar um método de pesquisa local para tentar encontrar uma solução baseada nesta atribuição inicial.
- O desenvolvimento de um acelerador combinatório é um processo bastante moroso que requer conhecimentos profundos da plataforma de hardware relevante. No âmbito deste trabalho utilizamos o fluxo de projecto tradicional. Contudo seria interessante explorar a utilidade das linguagens de alto nível, tais como Handel-C, na concepção de sistemas complexos. Actualmente, os circuitos sintetizados a partir das descrições de alto nível consomem mais recursos de hardware que os circuitos projectados manualmente. Mas as linguagens de alto nível facilitam substancialmente o desenvolvimento de sistemas complexos, possibilitando uma avaliação rápida de soluções algorítmicas alternativas. Para além disso, pode-se prever que com os avanços na tecnologia de compiladores, as linguagens de alto nível vão suplantar (pelo menos parcialmente) as linguagens tradicionais de descrição de hardware de maneira semelhante à que aconteceu com a linguagem *assembly* que quase desapareceu do repertório dos programadores.
- A partição da aplicação entre software e hardware reconfigurável continua a ser feita à mão exigindo deste modo experiência no desenvolvimento de hardware para a implementação de aceleradores. Um dos desafios importantes é ultrapassar os fluxos de projecto separados no desenvolvimento de software e de hardware reconfigurável. Um cenário ideal seria a compilação a partir da especificação numa linguagem de alto nível que incluísse a partição automática entre software e hardware reconfigurável, i.e. uma co-compilação. Cremos que para o sucesso da computação reconfigurável são necessários aperfeiçoamentos das ferramentas de apoio a projecto de sistemas complexos.
- O progresso constante que se verifica nas arquitecturas de FPGAs abre possibilidades adicionais a explorar. Assim, as FPGAs recentes passaram a incluir núcleos de processadores embutidos que podem contribuir para o desenvolvimento de novos sistemas reconfiguráveis destinados a acelerar problemas de optimização combinatória. Estas novas arquitecturas de FPGAs são particularmente interessantes quando a solução de um problema é partilhada entre software e hardware.

Referências

[Abramovici97] M. Abramovici, D. Saab, “Satisfiability on Reconfigurable Hardware”, Proc. of the 7th Int. Workshop on Field-Programmable Logic and Applications – FPL’97, Lecture Notes in Computer Science, vol. 1304, Springer, pp. 448-456.

[Abramovici99a] M. Abramovici, J.T. de Sousa, “A Virtual Logic Algorithm for Solving Satisfiability Problems Using Reconfigurable Hardware”, K.L. Pocek, J. Arnold (eds.), Proc. of the IEEE Symp. on FPGAs for Custom Computing Machines – FCCM’99, pp. 306-307.

[Abramovici99b] M. Abramovici, J.T. de Sousa, D. Saab, “A massively-parallel easily-scalable satisfiability solver using reconfigurable hardware”, Proc. of the Design Automation Conf. – DAC’99, pp. 684-690.

[Abramovici00] M. Abramovici, J.T. de Sousa, “A SAT solver using reconfigurable hardware and virtual logic”, Journal of Automated Reasoning, vol. 24, nos. 1-2, Feb. 2000, pp. 5-36.

[Abramson97] D. Abramson, P. Logothetis, A. Postula, M. Randall, “Application Specific Computers for Combinatorial Optimisation”, Proc. of the Australian Computer Architecture Workshop, Sydney, Feb. 1997, Springer-Verlag, pp. 29-43.

[Abramson98a] D. Abramson, P. Logothetis, A. Postula, M. Randall, “FPGA Based Custom Computing Machines for Irregular Problems”, Proc. of the 4th Int. Symp on High-Performance Computer Architecture – HPCA’98, Feb. 1998, Las Vegas, Nevada, USA.

[Abramson98b] D. Abramson, P. Logothetis, M. Randall, A. Postula, “A Tail of 2^N cities: Performing Combinatorial Optimisation using Linked Lists on Special Purpose Computers”, Proc. of the Int. Conf. on Computational Intelligence and Multimedia Applications – ICCIMA’98, Gippsland, Australia, Feb. 1998, pp. 17-26.

[Actel] [Online]: http://www.actel.com/products/devices.html.

[Aho83] A. Aho, J. Hopcroft, J. Ullman, "Data Structures and Algorithms", Addison-Wesley, 1983.

[Ahuja95] R.K. Ahuja, T.L. Magnanti, J.B. Orlin, M.R. Reddy, “Applications of network optimization”, Handbooks in Operations Research and Management Science, vol. 7, “Network Models”, M.O. Ball, T.L. Magnanti, C.L. Monma, G.L. Nemhauser (eds.), Elsevier, 1995, pp. 1-83.

[Alander98] J.T. Alander, J. Lampinen, "Cam Shape Optimization by Genetic Algorithm", "Genetic Algorithms and Evolution Strategies in Engineering and Computer Science", D. Quagliarella, J. Périaux, C. Poloni, G. Winter (eds.), 1998, pp. 153-174.

[Alpha] [Online]: http://www.alpha-data.com/overview.html.

[Altera] [Online]: http://www.altera.com/products/devices/dev-index.jsp.

[Aporntewan01] C. Aporntewan, P. Chongstitvatana, "A Hardware Implementation of the Compact Genetic Algorithm", Proc. of the IEEE Congress on Evolutionary Computation, Korea, May 2001, pp. 624-629.

[Athanas93] P.M. Athanas, H.F. Silverman, "Processor Reconfiguration through Instruction Set Metamorphosis", Computer, vol. 26, no. 3, Mar. 1993, pp. 11-18.

[Atmel] [Online]: http://www.atmel.com/products/FPGA/.

[Babb93] J. Babb, R. Tessier, A. Agarwal, "Virtual Wires: Overcoming Pin Limitations in FPGA-based Logic Emulators", Proc. of the IEEE Workshop on FPGA-based Custom Computing Machines – FCCM'93, Apr. 1993, pp. 142-151.

[Babb96] J.W. Babb, M.I. Frank, A. Agarwal, "Solving graph problems with dynamic computation structures", "High-Speed Computing, Digital Signal Processing, and Filtering Using reconfigurable Logic", J. Schewel (ed.), Proc. SPIE 2914, Bellingham, WA, Oct. 1996, pp. 225-236.

[Bäck98] T. Bäck, U. Hammel, R. Lewandowski, M. Mandischer, B. Naujoks, S. Rolf, M. Schütz, H.-P. Schwefel, J. Sprave, S. Theis, "Evolutionary Algorithms: Applications at The Informatik Center Dortmund", "Genetic Algorithms and Evolution Strategies in Engineering and Computer Science", D. Quagliarella, J. Périaux, C. Poloni, G. Winter (eds.), 1998, pp. 175-204.

[Baranov81] S. Baranov, L. Zhuravina, V. Sklyarov, "Computer Aided Design", Minsk, High School, 1981, (em russo).

[Baranov86] S. Baranov, V. Sklyarov, "Digital Devices Based on Programmable Matrix LSI", Moscow, Radio and Communications, 1986, (em russo).

[Baranov94] S. Baranov, "Logic Synthesis for Control Automata", Kluwer Academic Publishers, 1994.

[Bayardo97] R.J. Bayardo Jr., R.C. Schrag, "Using CSP Look-Back Techniques to Solve Real-World SAT Instances", Proc. of the 14th Int. Conf. on Artificial Intelligence, 1997, pp. 203-208.

[Betz98] V. Betz, J. Rose, "How Much Logic Should Go in an FPGA Logic Block?", IEEE Design & Test of Computers, vol. 15, no. 1, Jan.-Mar. 1998, pp. 10-15.

[Bhatia94] K. Bhatia, "Genetic Algorithms and the Traveling Salesman Problem", Computer Science and Engineering CSE292: New Age Algorithms,

University of California at San Diego, 1994.

[Bouridane99] A. Bouridane, D. Crookes, P. Donachy, K. Alotaibi, K. Benkrid, "A high level FPGA-based abstract machine for image processing", Journal of Systems Architecture, 45, 1999, pp. 809-824.

[Boyd00] M. Boyd, T. Larrabee, "ELVIS – a scalable, loadable custom programmable logic device for solving Boolean satisfiability problems", Proc. of the 8th IEEE Int. Symp. on Field-Programmable Custom Computing Machines – FCCM'2000.

[Brayton84] R. Brayton, G. Hachtel, C. McMullen, A.L. Sangiovanni-Vincentelli, "Logic Minimization Algorithms for VLSI Synthesis", Kluwer Academic Publishers, 1984.

[Breuer00] M.A. Breuer, M. Sarrafzadeh, F. Somenzi, "Fundamental CAD Algorithms", IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems, vol. 19, no. 12, Dec. 2000, pp. 1449-1475.

[Brown96] S. Brown, J. Rose, "FPGA and CPLD Architectures: A Tutorial", IEEE Design & Test of Computers, vol. 13, no. 2, 1996, pp. 42-57.

[Bryant86] R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation", IEEE Trans. on Computers, vol. C-35, no. 8, Aug. 1986, pp. 677-691.

[Buell96] D. A. Buell, J. M. Arnold, W. J. Kleinfelder, "Splash 2: FPGAs in a Custom Computing Machine", IEEE Computer Society Press, Los Alamitos, CA, 1996.

[Callahan00] T.J. Callahan, J.R. Hauser, J. Wawrzynek, "The Garp Architecture and C Compiler", Computer, Apr. 2000, pp. 62-69.

[Chang99] Y.-W. Chang, D.F. Wong, C.K. Wong, "Programmable Logic Devices", Encyclopedia of Electrical and Electronics Engineering, J.G. Webster (ed.), John Wiley & Sons, vol. 17, 1999, pp. 334-348.

[Chin86] R.T. Chin, C.R. Dyer, "Model-based Recognition in Robot Vision", ACM Computing Surveys, vol. 18, no. 1, Mar. 1986, pp. 67-108.

[Chung99] C.K. Chung, P.H.W. Leong, "An Architecture for solving boolean satisfiability using runtime configurable hardware", Proc. of the 1999 Int. Workshops on Parallel Processing, Fukushima, Japan, pp. 352-357.

[CoCentric] [Online]: http://www.synopsys.com/products/cocentric_systemC/cocentric_systemC_ds.pdf.

[Compton02] K. Compton, S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software", ACM Computing Surveys, vol. 34, no. 2, Jun. 2002, pp. 171-210.

[Cormen97] T.H. Cormen, C.E. Leiserson, R.L. Rivest, "Introduction to Algorithms", McGraw-Hill, 1997.

[Crawford94] J.M. Crawford, A.B. Baker, "Experimental result on the Application of Satisfiability Algorithms to Scheduling Problems", Proc. of the 12th National Conf. on Artificial Intelligence – AAAI'94, Washington, USA, 1994, pp. 1092-1097.

[Dandalis99] A. Dandalis, A. Mei, V.K. Prasanna, "Domain Specific Mapping for Solving Graph Problems on Reconfigurable Devices", Reconfigurable Architectures Workshop - RAW '99, Apr. 1999.

[Dandalis01] A. Dandalis, V.K. Prasanna, B. Thiruvengadam, "Run-time Performance Optimization of an FPGA-based Deduction Engine for SAT Solvers", Proc. of the FPL'01, Lecture Notes in Computer Science, vol. 2147, Springer, 2001, pp. 315-325.

[Dandalis02] A. Dandalis, V.K. Prasanna, "Run-time performance optimization of an FPGA-based deduction engine for SAT solvers", ACM Trans. on Design Automation of Electronic Systems, vol. 7, no. 4, Oct. 2002, pp. 547-562.

[Darwen02] P.J. Darwen, "Evolving a Schedule with Batching, Precedence Constraints, and Sequence-Dependent Setup Times: Crossover Needs Building Blocks", "Developments in Applied Artificial Intelligence", T. Hendtlass, M. Ali (eds.), 2002, pp. 525-535.

[Davis60] M. Davis, H. Putnam, "A Computing Procedure for Quantification Theory", Journal of ACM, vol. 7, 1960, pp. 201-215.

[Davis62] M. Davis, G. Logemann, D. Loveland, "A machine program for theorem proving", Communications of the ACM, n. 5, 1962, pp. 394-397.

[DeHon00] A. DeHon, "The Density Advantage of Configurable Computing", Computer, vol. 33, no. 4, Apr. 2000, pp. 41- 49.

[Devadas89] S. Devadas, "Optimal Layout Via Boolean Satisfiability", IEEE Int. Conf. on CAD, 1989, pp. 294-297.

[Dick99] C. Dick, F. Harris, "Virtual signal processors", Microprocessors and Microsystems, 22, 1999, pp. 135-148.

[Diestel00] R. Diestel, "Graph Theory", 2nd edn., Springer, 2000.

[DIMACS] DIMACS challenge benchmarks. [Online]: http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/benchm.html.

[Dubois98] M. Dubois, J. Jeong, Y.H. Song, A. Moga, "Rapid hardware prototyping on RPM-2", Design & Test of Computers, IEEE , vol. 15, no. 3, Jul.-Sep. 1998, pp. 112 –118.

[Estrin60] G. Estrin, "Organization of Computer Systems—The Fixed Plus Variable Structure Computer," Proc. Western Joint Computer Conf., New York, 1960, pp. 33-40.

[Estrin02] G. Estrin, "Reconfigurable Computer Origins: The UCLA Fixed-Plus-

Variable (F+V) Structure Computer", IEEE Annals of the History of Computing, Oct.-Dec. 2002, pp. 3-9.

[Feldman90] R. Feldman, M.C. Golumbic, "Optimization Algorithms for Student Scheduling via Constraint Satisfiability", Computer Journal, vol. 33, no. 4, 1990, pp. 356-364.

[Fogel99] D.B. Fogel, "An Introduction to Evolutionary Computation and Some Applications", "Evolutionary Algorithms in Engineering and Computer Science", K. Miettinen, M.M. Mäkelä, P. Neittaanmäki, J. Périaux (eds.), 1999, pp. 23-41.

[Freeman95] J.W. Freeman, "Improvements to Propositional Satisfiability Search Algorithms", Ph.D. dissertation, University of Pennsylvania, 1995.

[Garey79] M.R. Garey, D.S. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness", W.H. Freeman and Company, San Francisco, 1979.

[Gen00] M. Gen, R. Cheng, "Genetic Algorithms & Engineering Optimization", John Wiley & Sons, Inc., 2000.

[Glover89] F. Glover, "Tabu Search – Part I", ORSA Journal on Computing, vol. 1, no. 3, Summer 1989, pp. 190-206.

[Glover90] F. Glover, "Tabu Search – Part II", ORSA Journal on Computing, vol. 2, no. 1, Winter 1990, pp. 4-32.

[Goel81] P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinatorial Logic Circuits", IEEE Trans. on Computers, vol. C-30, no. 3, 1981, pp. 215-222.

[Gokhale98] M. B. Gokhale, J. M. Stone, "NAPA C: Compiling for a Hybrid RISC/FPGA Architecture", Proc. of the IEEE Symp. on FPGAs for Custom Computing Machines – FCCM'98, Apr. 1998, pp. 126-135.

[Goldberg85] D.E. Goldberg, R. Lingle, "Alleles, Loci, and the Traveling Salesman Problem", Proc. of the Int. Conf. on Genetic Algorithms, 1985, pp. 154-159.

[Goldberg89] D.E. Goldberg, "Genetic Algorithms in Search, Optimization & Machine Learning", Addison-Wesley, 1989.

[Goldberg02a] E. Goldberg, M.P. Prasad, R.K. Brayton, "Using Problem Symmetry in Search Based Satisfiability Algorithms", Proc. of the Design, Automation and Test in Europe Conf. – DATE'02, pp. 134-141.

[Goldberg02b] E. Goldberg, Y. Novikov, "BerkMin: a Fast and Robust SAT-solver", Proc. of the Design, Automation and Test in Europe Conf. – DATE'02, pp. 142-149.

[Golden00] B.L. Golden, B.K. Kaku, "Difficult Routing and Assignment Problems", "Handbook of Discrete and Combinatorial Mathematics", K.H. Rosen, J.G.

Michaels, J.L. Gross, J.W. Grossman, D.R. Shier (eds.), CRC Press, 2000, pp. 692-705.

[Golumbic80] M. Golumbic, "Algorithmic Graph Theory and Perfect Graphs", Academic Press, 1980.

[Graham95] P. Graham and B. Nelson, "A Hardware Genetic Algorithm for the Traveling Salesman Problem on Splash 2", Proc. of the 5th Int. Workshop on Field Programmable Logic and Applications – FPL'95, Oxford, England, Aug. 1995, pp. 352-361.

[Graham96] P. Graham, B. Nelson, "Genetic Algorithms in Software and in Hardware – A Performance Analysis of Workstation and Custom Computing Machine Implementation", Proc. of the IEEE Symp. on FPGAs for Custom Computing Machines – FCCM'96, pp. 216-225.

[Gschwind01] M. Gschwind, V. Salapura, D. Maurer, "FPGA prototyping of a RISC processor core for embedded applications", IEEE Trans. on Very Large Scale Integration (VLSI) Systems, vol. 9, no. 2, Apr. 2001, pp. 241-250.

[Gu95] J. Gu, R. Puri, "Asynchronous circuit synthesis with Boolean satisfiability", IEEE Trans. on CAD, vol. 14, no. 8, Aug. 1995, pp. 961-973.

[Gu96] J. Gu, R. Puri, B. Du, "Boolean Satisability Problems in VLSI Engineering", DIMACS Workshop on Satisfiability Problems: Theory and Applications, Mar. 1996.

[Gu97] J. Gu, P.W. Purdom, J. Franco, B.W. Wah, "Algorithms for the Satisfiability (SAT) Problem: A Survey", DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 35, 1997, pp. 19-151.

[Guccione01] S.A. Guccione, "Reconfigurable Computing at Xilinx", keynote talk, EUROMICRO Symp. on Digital System Design – DSD'2001, Warsaw, Poland, Sept. 2001.

[Gustin99] V. Gustin, "An FPGA extension to ALU functions", Microprocessors and Microsystems, 22, 1999, pp. 501-508.

[Hachtel96] G.D. Hachtel, F. Somenzi, "Logic Synthesis and Verification Algorithms", Kluwer Academic Publishers, 1996.

[Hamadi97] Y. Hamadi, D. Merceron, "Reconfigurable Architectures: A new vision for optimization problems", Proc. of the 3rd Int. Conf. on Principles and Practice of Constraint Programming, Austria, 1997, pp. 209-215.

[Handbook00] "Handbook of Discrete and Combinatorial Mathematics", K.H. Rosen, J.G. Michaels, J.L. Gross, J.W. Grossman, D.R. Shier (eds.), CRC Press, 2000.

[Handel-C] HandelC. [Online]: http://www.celoxica.com/.

[Hansen02] L. Hansen, "The New ISE 5.1i Software", Xcell Journal, Winter 2002, n. 44, pp. 10-11.

[Hartenstein01a] R. Hartenstein, “A decade of Reconfigurable Computer: A Visionary Retrospective”, Proc. of the Design, Automation and Test in Europe Conf. – DATE’01, pp. 642-649.

[Hartenstein01b] R. Hartenstein, “Reconfigurable Computing: A New Business Model – and its Impact on SoC Design”, Proc. of the EUROMICRO Symp. on Digital System Design – DSD’2001, Warsaw, Poland, Sept. 2001, pp. 103-110.

[Hauck97] S. Hauck, T.W. Fry, M.M. Hosler, J.P. Kao, “The Chimaera Reconfigurable Functional Unit”, IEEE Symp. on Field-Programmable Custom Computing Machines – FCCM’97, pp. 87-96.

[Hauser00] J.R. Hauser, “Augmenting a Microprocessor with Reconfigurable Hardware”, Ph.D. dissertation, University of California at Berkeley, 2000.

[Havener02] M. Havener, N. Hartl, “500+ Xilinx FPGAs Search for Elusive Higgs Boson at 1.5 Terabytes per Second”, Xcell Journal, Winter 2002, n. 44, pp. 68-73.

[Haynes00] S.D. Haynes, J. Stone, W. Luk, “Video Image Processing with the Sonic Architecture”, Computer, Apr. 2000, pp. 50-57.

[Hernandez02] A.M. Hernandez, “Deep Memory Yields Effective In-System Debugging”, Xcell Journal, Winter 2002, n. 44, pp. 26-28.

[Holland75] J.H. Holland, “Adaptation in Natural and Artificial Systems”, University of Michigan Press, 1975.

[Huang93] X. Huang, J. Gu, Y. Wu, “A Constrained Approach to Multifont Chinese Character Recognition”, IEEE Trans. on Pattern Analysis and Machine Intelligence, vol. 15, no. 8, Aug. 1993, pp. 838-843.

[Hutching95] B.L. Hutching, M.J. Wirthlin, “Implementation Approaches for Reconfigurable Logic Applications”, Proc. of the FPL’95, Lecture Notes in Computer Science, vol. 975, Springer, 1995, pp. 419-428.

[Iliopoulos00] M. Iliopoulos, T. Antonakopoulos, “Reconfigurable Network Processors Based on Field Programmable System Level Integrated Circuits”, Proc. of the FPL’00, Lecture Notes in Computer Science, vol. 1896, Springer, 2000, pp. 39-47.

[Jacob98] J.A. Jacob, “Memory Interfacing for the OneChip Reconfigurable Processor”, M.Sc. dissertation, University of Toronto, 1998.

[Jong99] K. De Jong, “Evolutionary Computation: Recent Developments and Open Issues”, “Evolutionary Algorithms in Engineering and Computer Science”, K. Miettinen, M.M. Mäkelä, P. Neittaanmäki, J. Périaux (eds.), 1999, pp. 43-54.

[Jóźwiak02] L. Jóźwiak, A. Postula, “Genetic engineering versus natural evolution. Genetic Algorithms with deterministic operators”, Journal of Systems Architecture, 48, 2002, pp. 99-112.

[Jünger95] M. Jünger, G. Reinelt, G. Rinaldi, "The Traveling Salesman Problem", Handbooks in Operations Research and Management Science, vol. 7, "Network Models", M.O. Ball, T.L. Magnanti, C.L. Monma, G.L. Nemhauser (eds.), Elsevier, 1995, pp. 225-330.

[Kean00] T. Kean, "It's FPL, Jim – But Not as We Know It! Opportunities for the New Commercial Architectures", Proc. of the FPL'00, Lecture Notes in Computer Science, vol. 1896, Springer, 2000, pp. 575-584.

[Kennedy03] I. Kennedy, "Exploiting Redundancy to Speedup Reconfiguration of an FPGA", Proc. of the 13th Int. Conf. on Field-Programmable Logic and Applications – FPL'2003, Lecture Notes in Computer Science, vol. 2778, Springer, pp. 262-271.

[Kirkpatrick83] S. Kirkpatrick, C.D. Gelatt, Jr., M.P. Vecchi, "Optimization by Simulated Annealing", Science, vol. 220, no. 4598, 13 May 1983, pp. 671-680.

[Kocan03] F. Kocan, "Reconfigurable Randomized K-way Graph Partitioning", Proc. of the EUROMICRO Symp. on Digital System Design – DSD'2003, Antalya, Turkey, Sept. 2003, pp. 272-278.

[Koza99] J.R. Koza, F.H. Bennet III, D. Andre, M.A. Keane, "Genetic Programming III", Morgan Kaufmann Publishers, 1999.

[Kreher99] D.L. Kreher, D.R. Stinson, "Combinatorial Algorithms: Generation, Enumeration, and Search", CRC Press, 1999.

[Krupnova00] H. Krupnova, G. Saucier, "FPGA-Based Emulation: Industrial and Custom Prototyping Solutions", Proc. of the FPL'00, Lecture Notes in Computer Science, vol. 1896, Springer, 2000, pp. 68-77.

[Larrabee92] T. Larrabee, "Test Pattern Generation Using Boolean Satisfiability", IEEE Trans. on CAD of Integrated Circuits and Systems, vol. 11, no. 1, 1992, pp. 4-15.

[Lattice] Lattice. [Online]: http://www.vantis.com/products/index.cfm.

[Laumanns02] N. Laumanns, M. Laumanns, H. Kitterer, "Evolutionary Multi-objective Integer Programming for the Design of Adaptive Cruise Control Systems", "Developments in Applied Artificial Intelligence", T. Hendtlass, M. Ali (eds.), 2002, pp. 200-210.

[Leong99] P.H.W. Leong, C.K. Chung, "FPGA Based Runtime Configurable Clause Evaluator for SAT Problems", Electronics Letters, vol. 35, no. 19, Sept. 1999, pp. 1618-1619.

[Leong01] P.H.W. Leong, C.W. Sham, W.C. Wong, H.Y. Wong, W.S. Yuen, M.P. Leong, "A Bitstream Reconfigurable FPGA Implementation of the WSAT algorithm", IEEE Trans. on VLSI Systems, vol. 9, no. 1, 2001, pp. 197-201.

[Litke84] J.D. Litke, "An Improved Solution to the Traveling Salesman Problem with

Thousands of Nodes", Communications of the ACM, vol. 27, no. 12, Dec. 1984, pp. 1227-1236.

[Mangione97] W.H. Mangione-Smith, B.L. Hutchings, "Configurable Computing: The Road Ahead", Proc. of the Reconfigurable Architectures Workshop – RAW'97, pp. 81-96.

[Mano00] M.M. Mano, C.R. Kime, "Logic and Computer Design Fundamentals", 2nd edn., Prentice Hall, 2000.

[Matula72] D.W. Matula, G. Marble, J.D. Isaacson, "Graph Coloring Algorithms", "Graph Theory and Computing", R.C. Read (ed.), Academic Press, New York, 1972, pp. 109-122.

[McMillan00] S. McMillan, S.A. Guccione, "Partial Run-Time Reconfiguration Using JRTR", Proc. of the FPL'00, Lecture Notes in Computer Science, vol. 1896, Springer, 2000, pp. 352-360.

[Mei03] B. Mei, S. Vernalde, D. Verkest, H. De Man, R. Lauwereins, "ADRES: An Architecture with Tightly Coupled VLIW Processor and Course-Grained Reconfigurable Matrix", Proc. of the 13th Int. Conf. on Field-Programmable Logic and Applications – FPL'2003, Lecture Notes in Computer Science, vol. 2778, Springer, pp. 61-70.

[Mencer99] O. Mencer, M. Platzner, "Dynamic Circuit Generation for Boolean Satisfiability in an Object-Oriented Design Environment", Proc. of the 32nd Hawaii Int. Conf. on System Sciences, 1999.

[Mencer01] O. Mencer, M. Platzner, M. Morf, M.J. Flynn, "Object-Oriented Domain Specific Compilers for Programming FPGAs", IEEE Trans. on Very Large Scale Integration (VLSI) Systems, vol. 9, no. 1, Feb. 2001, pp. 205-210.

[Merz97] P. Merz, B. Freisleben, "Genetic Local Search for the TSP: New Results", Proc. of the IEEE Int. Conf. on Evolutionary Computation - ICEC'97, pp. 159-164.

[Michalewicz00] Z. Michalewicz, D.B. Fogel, "How to Solve It: Modern Heuristics", Springer-Verlag, 2000.

[Micheli94] G. De Micheli, "Synthesis and Optimization of Digital Circuits", McGraw-Hill, Inc., 1994.

[Miller98] J. F. Miller, P. Thomson, T. Fogarty, "Designing Electronic Circuits Using Evolutionary Algorithms. Arithmetic Circuits: A Case Study", "Genetic Algorithms and Evolution Strategies in Engineering and Computer Science", D. Quagliarella, J. Périaux, C. Poloni, G. Winter (eds.), 1998, pp. 105-131.

[Miller00] J.F. Miller, D. Job, V.K. Vassilev, "Principles in the Evolutionary Design of Digital Circuits – Part I", Genetic Programming and Evolvable Machines 1, 2000, pp. 7-35.

[Mirsky96] E. Mirsky, A. DeHon, "MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Destribution and Deployable Resources", Proc. of the FCCM'96, Apr. 1996, pp. 157-166.

[Moskewicz01] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, S. Malik, "Chaff: Engineering an Efficient SAT Solver", Proc. of the 38th Design Automation Conf. – DAC'01, Jun. 2001, pp. 530-535.

[Murgai95] R. Murgai, R.K. Brayton, A. Sangiovanni-Vincentelli, "Logic Synthesis for Field-Programmable Gate Arrays", Kluwer Academic Publishers, 1995.

[Nedjah02] N. Nedjah, L. de Macedo Mourelle, "Minimal Addition Chain for Efficient Modular Exponentation Using Genetic Algorithms", "Developments in Applied Artificial Intelligence", T. Hendtlass, M. Ali (eds.), 2002, pp. 88-98.

[Nyakoe02] G.N. Nyakoe, M. Ohki, S. Tabuchi, M. Ohkita, "Optimization of Pulse Pattern for a Multi-robot Sonar System Using Genetic Algorithm", "Developments in Applied Artificial Intelligence", T. Hendtlass, M. Ali (eds.), 2002, pp. 179-189.

[Page96] I. Page, "Reconfigurable Processor Architectures", "Microprocessors and Micrisystems", vol. 20, no. 3, 1996, pp. 185-196.

[Perkowski97] M.A. Perkowski, L. Jozwiak, D. Foote, "Architecture of a Programmable FPGA Coprocessor for Constructive Induction Approach to Machine Learning and other Discrete Optimization Problems", "Reconfigurable Architectures: High Performance by Configware", R.W. Hartenstein, V.K. Prasanna (eds.), IT press, Chicago, USA, 1997, pp. 33-40.

[Perkowski02] M. Perkowski, D. Foote, Q. Chen, A. Al-Rabadi, L. Jozwiak, "Learning Hardware Using Multiple-Valued Logic- Part 2: Cube Calculus and Architecture", IEEE Micro, vol. 22, no. 3, May/Jun. 2002, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 52-61.

[Phillips81] D.T. Phillips, A. Garcia-Diaz, "Fundamentals of Network Analysis", Prentice-Hall, Inc., 1981.

[Platzner98] M. Platzner, G. Micheli, "Acceleration of Satisfiability Algorithms by Reconfigurable Hardware", Proc. of the 8th Int. Workshop on Field Programmable Logic and Applications – FPL'98, Tallin, Estonia, Springer-Verlag, 1998, pp. 69-78.

[Platzner00] M. Platzner, "Reconfigurable accelerators for combinatorial problems", IEEE Computer, Apr. 2000, pp. 58-60.

[Plessl01] C. Plessl, M. Platzner, "Instance-Specific Accelerators for Minimum Covering", Proc. of the 1st Int. Conf. on Engineering of Reconfigurable Systems and Algorithms, Las Vegas, USA, Jun. 2001, pp. 85-91.

[PLX] [Online]: http://www.plxtech.com/products/9080/.

[Przybus02] B. Przybus, "New ChipScope Pro Integrated Bus Analyzer", Xcell Journal, Winter 2002, n. 44, pp. 24-25.

[QuickLogic] [Online]: http://www.quicklogic.com/.

[Rabaey00] J.M. Rabaey, "Silicon Platforms for the Next Generation Wireless Systems – What Role Does Reconfigurable Hardware Play?", Proc. of the FPL'00, Lecture Notes in Computer Science, vol. 1896, Springer, 2000, pp. 277-285.

[Radunovic98] B. Radunovic, V. Milutinovic, "A Survey of Reconfigurable Computing Architectures", Proc. of the FPL'98, Lecture Notes in Computer Science, vol. 1482, Springer, 1998, pp. 376-385.

[Rao83] C.D.V.P . Rao, N.N. Biswas, "On the Minimization of Wordwidth in the Control Memory of a Microprogrammed Digital Computer", IEEE Trans. on Computers, vol. 32, no. 9, Sept. 1983, pp. 863-868.

[Rashid98] A. Rashid, J. Leonard, W.H. Mangione-Smith, "Dynamic Circuit Generation for Solving Specific Problem Instances of Boolean Satisfiability", 6th IEEE Symp. on FPGAs for Custom Computing Machines – FCCM'98, pp. 196-205.

[Razdan94] R. Razdan, M.D. Smith, "A High-Performance Microarchitecture with Hardware-Programmable Functional Units", Proc. of the 27th Annual IEEE/ACM Int. Symp. on Microarchitecture, Nov. 1994, pp. 172-180.

[RC_Bibliography] RC Bibliography, [Online]: http://www.ife.ee.ethz.ch/~enzler/rc/bib.html.

[Redekopp00] M. Redekopp, A. Dandalis, "A Parallel Pipelined SAT Solver for FPGA's", Proc. of the FPL'00, Lecture Notes in Computer Science, vol. 1896, Springer, 2000, pp. 462-468.

[Reeves99] C.R. Reeves, T. Yamada, "Embedded Path Tracing and Neighbourhood Search Techniques in Genetic Algorithms", "Evolutionary Algorithms in Engineering and Computer Science", K. Miettinen, M.M. Mäkelä, P. Neittaanmäki, J. Périaux (eds.), 1999, pp. 95-111.

[Reis02] N.A. Reis, J.T. de Sousa, "On Implementing a Configware/Software SAT Solver", Proc. of the 10th IEEE Int. Symp. on Field-Programmable Custom Computing Machines – FCCM'02, pp. 282-283.

[Ripado01] R.C. Ripado, J.T. de Sousa, "A Simulation Tool for a Pipelined SAT Solver", Proc. of the XVI Conf. on Design of Circuits and Integrated Systems – DCIS'2001, Porto, Portugal, Nov. 2001, pp. 498-503.

[Rupp98] C.R. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J.M. Arnold, M. Gokhale, "The NAPA Adaptive Processing Architecture", Proc. of the IEEE Symp. on FPGAs for Custom Computing Machines – FCCM'98, pp. 28-37.

[Salami02a] M. Salami, T. Hendtlass, "A Fast Evolutionary Algorithm for Image

Compression in Hardware", "Developments in Applied Artificial Intelligence", T. Hendtlass, M. Ali (eds.), 2002, pp. 241-252.

[Salami02b] M. Salami, T. Hendtlass, "A Fitness Estimation Strategy for Genetic Algorithms", "Developments in Applied Artificial Intelligence", T. Hendtlass, M. Ali (eds.), 2002, pp. 502-513.

[Sanchez96] E. Sanchez, M. Tomassini (eds.), "Towards Evolvable Hardware. The Evolutionary Engineering Approach", Lecture Notes in Computer Science, vol. 1062, 1996.

[Sanchez99] E.Sanchez, M. Sipper, J.O. Haenni, J.L. Beuchat, A. Stauffer, A. Perez-Uribe, "Static and Dynamic Configurable Systems", IEEE Trans. on Computers, vol. 48, no. 6, Jun. 1999, pp. 556-564.

[Sasao97] T. Sasao, J.T. Butler, "On bi-decomposition of logic functions," Int. Workshop on Logic Synthesis, Lake Tahoe, California, USA, May 18-21, 1997, vol. 2, pp. 1-6.

[SAT03] 2003 SAT Competition. [Online]: http://www.lri.fr/~simon/contest03/results/.

[Schaumont01] P. Schaumont, I. Verbauwhede, K. Keutzer, M. Sarrafzadeh, "A quick safari through the reconfiguration jungle", Proc. of the 38th Design Automation Conf. - DAC'01, pp. 172-177.

[Shackleford01] B. Shackleford, G. Snider, R. J. Carter, E. Okushi, M. Yasuda, K. Seo, H. Yasuura, "A High-Performance, Pipelined, FPGA-Based Genetic Algorithm Machine", Genetic Programming and Evolvable Machines, vol. 2, no. 1, Kluwer Academic Publishers, Mar. 2001, pp. 33-60.

[Sharma98] A.K. Sharma, "Programmable Logic Handbook. PLDs, CPLDs, and FPGAs", MCGraw-Hill, 1998.

[Silva99] J. M. Silva, K.A. Sakallah, "GRASP: a search algorithm for propositional satisfiability", IEEE Trans. on Computers, vol. 48, no. 5, May 1999, pp. 506-521.

[Simon02] L. Simon, D. Le Berre, E. Hirsch, "The SAT2002 Competition. Technical Report (preliminary draft)", [Online]: http://www.satlive.org/SATCompetition/onlinereport.pdf, 2002.

[Singh00] H. Singh, M.H. Lee, G. Lu, F.J. Kurdahi, N. Bagherzadeh, E.M.C. Filho, "MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications", IEEE Trans. on Computers, vol. 49, no. 5, May 2000, pp. 465-481.

[Sipper98] M.Sipper, D.Mange, A.Pérez-Uribe (eds.), "Evolvable Systems: From Biology to Hardware", Lecture Notes in Computer Science, vol. 1478, 1998.

[Skliarova00a] I. Skliarova, A.B. Ferrari, "Exploiting FPGA-based Architectures and Design Tools for Problems of Reconfigurable Computations", Proc. of the XIII

Symp. on Integrated Circuits and System Design – SBCCI'2000, Manaus, Brazil, Sept. 2000, pp. 347-352.

[Skliarova00b] I. Skliarova, A.B. Ferrari, "Development tools for problems of combinatorial optimization", Proc. of the 4th Portuguese Conf. on Automatic Control - CONTROLO'2000, Portugal, Oct. 2000, pp. 552-557.

[Skliarova01a] I. Skliarova, A.B. Ferrari, "Modelos matemáticos e problemas de optimização combinatória", Electrónica e Telecomunicações, v.3, Nº 3, Jan. 2001, pp. 202-208.

[Skliarova01b] I. Skliarova, A.B. Ferrari, "Design and Implementation of Reconfigurable Processor for Problems of Combinatorial Computations", Proc. of the EUROMICRO Symp. on Digital System Design – DSD'2001, Varsóvia, Poland, Sept. 2001, pp. 112-119.

[Skliarova02a] I. Skliarova, A.B. Ferrari, "A SAT Solver Using Software and Reconfigurable Hardware", Proc. of the Design, Automation and Test in Europe Conf. – DATE'02, Paris, France, Mar. 2002, p. 1094.

[Skliarova02b] I. Skliarova, A.B. Ferrari, "A hardware/software approach to accelerate Boolean satisfiability", Proc. of the IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop - DDECS'2002, Brno, Czech Republic, Apr. 2002, pp. 270-277.

[Skliarova02c] I. Skliarova, A.B. Ferrari, "FPGA-based Implementation of Genetic Algorithm for the Traveling Salesman Problem and its Industrial Application", "Developments in Applied Artificial Intelligence", T. Hendtlass, M. Ali (eds.), 2002, pp. 77-87.

[Sklyarov84a] V. Sklyarov, "Synthesis of Finite State Machines Based on Matrix LSI", Minsk: Science and Techniques, 1984, (em russo).

[Sklyarov84b] V. Sklyarov, "Regularly Structured Finite State Machines", Control Systems and Machines, Kiev, no. 2, 1984, pp. 23-28, (em russo).

[Sklyarov00] V. Sklyarov, "Synthesis and Implementation of RAM-based Finite State Machines in FPGAs", Proc. of the FPL'2000, Villach, Austria, Aug. 2000, pp. 718-728.

[Sklyarov01] V. Sklyarov, I. Skliarova, A.B. Ferrari, "Hierarchical Specification and Implementation of Combinatorial Algorithms Based on RHS Model", Proc. of the XVI Conf. on Design of Circuits and Integrated Systems - DCIS, 2001, pp. 486-491.

[Sklyarov02] V. Sklyarov, "An Evolutionary Algorithm for the Synthesis of RAM-Based FSMs", "Developments in Applied Artificial Intelligence", T. Hendtlass, M. Ali (eds.), 2002, pp. 108-118.

[Sklyarov03a] V. Sklyarov, I. Skliarova, "Architecture of Reconfigurable Processor for Implementing Search Algorithms over Discrete Matrices", Proc. of the Int.

Conf. on Engineering of Reconfigurable Systems and Algorithms - ERSA'2003, T.P. Plaks (ed.), Las Vegas, USA, Jun. 2003, pp. 127-133.

[Sklyarov03b] V. Sklyarov, I. Skliarova, "Design of Digital Circuits on the Basis of Hardware Templates", Proc. of the Int. Conf. on Embedded Systems and Applications - ESA'2003, H.R. Arabnia, L.T. Yang (eds.), Las Vegas, USA, Jun. 2003, pp. 56-62.

[Sousa01] J. de Sousa, J.P. Marques-Silva, M. Abramovici, "A configware/software approach to SAT solving", Proc. of the 9th IEEE Int. Symp. on Field-Programmable Custom Computing Machines – FCCM'01, May 2001.

[Souza03] L. de Souza, P. Ryan, J. Crawford, K. Wong, G. Zyner, T. McDermott, "Prototyping for the Concurrent Development of an IEEE 802-11 Wireless LAN Chipset", Proc. of the 13th Int. Conf. on Field-Programmable Logic and Applications – FPL'2003, Lecture Notes in Computer Science, vol. 2778, Springer, pp. 51-60.

[Stephan96] P. Stephan, R.K. Brayton, A.L. Sangiovanni-Vincentelli, "Combinational test generation using satisfiability", IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol. 15, no. 9, Sep. 1996, pp. 1167-1176.

[Suyama98] T. Suyama, M. Yokoo, H. Sawada, "Solving Satisfiability Problems Using Logic Synthesis and Reconfigurable Hardware", Proc. of the 31st Hawaii Int. Conf. on System Sciences – HICSS'98, vol. 7, 1998, pp. 179-186.

[Suyama99] T. Suyama, M. Yokoo, A. Nagoya, "Solving Satisfiability Problems on FPGAs using Experimental Unit Propagation", Proc. of the 5th Int. Conf. on Principles and Practice of Constraint Programming, Lecture Notes in Computer Science, vol. 1713, 1999.

[Suyama01] T. Suyama, M. Yokoo, H. Sawada, A. Nagoya, "Solving Satisfiability Problems Using Reconfigurable Computing", IEEE Trans. on VLSI Systems, vol. 9, no. 1, Feb. 2001, pp. 109-116.

[Synopsys] [Online]: http://www.synopsys.com.

[SystemC] SystemC. [Online]: http://www.systemc.org/.

[Tambouratzis98] T. Tambouratzis, "A consensus-function artificial neural network for map-coloring", IEEE Trans. on Systems, Man and Cybernetics, Part B, vol. 28, no. 5, Oct. 1998, pp. 721-728.

[Tang00] X. Tang, M. Aalsma, R. Jou, "A Compiler Directed Approach to Hiding Configuration Latency in Chameleon Processors", Proc. of the FPL'00, Lecture Notes in Computer Science, vol. 1896, Springer, 2000, pp. 27-38.

[Tessier01] R. Tessier, W. Burleson, "Reconfigurable Computing for Digital Signal Processing: A Survey", Journal of VLSI Signal Processing, vol. 28, nos. 1/2, May 2001, pp. 7-27.

[Thompson99] A. Thompson, P. Layzell, R.S. Zebulum, "Explorations in Design Space: Unconventional Electronics Design Through Artificial Evolution", IEEE Trans. on Evolutionary Computation, vol. 3, no. 3, Set. 1999, pp. 167 - 196.

[Togawa98] N. Togawa, M. Yanagisawa, T. Ohtsuki, "Maple-opt: a performance-oriented simultaneous technology mapping, placement, and global routing algorithm for FPGAs", IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol. 17, no. 9, Sep. 1998, pp. 803-818.

[Tomassini99] M. Tomassini, "Parallel and Distributed Evolutionary Algorithms: A Review", "Evolutionary Algorithms in Engineering and Computer Science", K. Miettinen, M.M. Mäkelä, P. Neittaanmäki, J. Périaux (eds.), 1999, pp. 113-133.

[Torresen00] J. Torresen, "Possibilities and Limitations of Applying Evolvable Hardware to Real-World Applications", Proc. of the FPL'00, Lecture Notes in Computer Science, vol. 1896, Springer, 2000, pp. 230-239.

[TSPLIB] Library of Traveling Salesman and related Problems. [Online]: http://www.informatik.uni-heidelberg.de/groups/comopt/software/TSPLIB95/index.html.

[Tredennick03] N. Tredennick, B. Shimamoto, "The Rise of Reconfigurable Systems", *keynote talk*, Proc. of the Int. Conf. on Engineering of Reconfigurable Systems and Algorithms - ERSA'2003, Las Vegas, USA, Jun. 2003, pp. 3-9.

[Vigander01] S. Vigander, "Evolutionary Fault Repair of Electronics in Space Applications", University of Sussex, [Online]: http://www.cogs.susx.ac.uk/users/adrianth/SVERRE_VIGANDER/sverre.html, 2001.

[Vuillemin96] J.E. Vuillemin, P. Bertin, D. Roncin, M. Shand, H.H. Touati, P. Boucard, "Programmable Active Memories: Reconfigurable Systems Come of Age", IEEE Trans. on Very Large Scale Integration (VLSI) Systems, vol. 4, no. 1, Mar. 1996, pp. 56-69.

[Waingold97] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, A. Agarwal, "Baring it all to Software: RAW machines", IEEE Computer, vol. 30, no. 9, Sept. 1997, pp. 86-93.

[Wang96] W. Wang, C.K. Rushforth, "An Adaptive Local-Search Algorithm for Channel-Assignment Problem (CAP)", IEEE Trans. on Vehicular Technology, vol. 45, no. 3, Aug. 1996, pp. 459-466.

[Wirthlin94] M.J. Wirthlin, B.L. Hutching, K.L. Gilson, "The Nano Processor: a Low Resource Reconfigurable Processor", Proc. of the IEEE Symp. on FPGAs for Custom Computing Machines – FCCM'94, Apr. 1994, pp. 23-30.

[Wood98] R.G. Wood, R. Rutenbar, "FPGA Routing and Routability Estimation via

Boolean Satisfiability", IEEE Trans. on VLSI Systems, vol. 6, no. 2, 1998, pp. 222-231.

[XESS] [Online]: http://www.xess.com.

[Xilinx] [Online]: http://www.xilinx.com/.

[Xilinx00] Xilinx, "The programmable Logic Data Book", San Jose, 2000.

[Xilinx_025] Xilinx, "Virtex™-E 1.8 V Extended Memory Field Programmable Gate Arrays", DS025-1 (v1.5), Product Specification, Jul. 2002, [Online]: http://direct.xilinx.com/bvdocs/publications/ds025.pdf.

[Yap03] R.H.C. Yap, S.Z.Q. Wang, M.J. Henz, "Hardware Implementations of Real-Time Reconfigurable WSAT Variants", Proc. of the 13th Int. Conf. on Field-Programmable Logic and Applications – FPL'2003, Lecture Notes in Computer Science, vol. 2778, Springer, pp. 488-496.

[Yokoo96] M. Yokoo, T. Suyama, H. Sawada, "Solving Satisfiability Problems Using Field Programmable Gate Arrays: First Results", Proc. of the 2nd Int. Conf. on Principles and Practice of Constraint Programming, 1996, pp. 497-509.

[Yung99] W.H. Yung, Y.W. Seung, K.H. Lee, P.H.W. Leong, "A Runtime Reconfigurable Implementation of the GSAT Algorithm", Proc. of the 9th Int. Workshop on Field Programmable Logic and Applications - FPL'99, Glasgow, UK, pp. 526-531.

[Zakrevski71] A.D. Zakrevski, "Algorithms of Synthesis of Discrete Automata", Moscow: Science, 1971, (em russo).

[Zakrevski81] A.D. Zakrevski, "Logical Synthesis of Cascade Networks", Moscow: Science, 1981, (em russo).

[Zakrevski90] A.D. Zakrevski, "Combinatorial Theory of Logical Design", Automatic and Computer Engineering, n. 2, 1990, pp. 68-79, (em russo).

[Zakrevski98] A.D. Zakrevski, "Combinatorial Problems over Logical Matrices in Logic Design and Artificial Intelligence", Electrónica e Telecomunicações, vol. 2, no. 2, 1998, pp. 261-268.

[Zakrevski00] A.D. Zakrevski, "Graph Coloring and Decomposition of Boolean Functions", Logical Design, n. 5, 2000, (em russo).

[Zhong98a] P. Zhong, M. Martonosi, P. Ashar, S. Malik, "Accelerating Boolean Satisfiability with Configurable Hardware", K.L. Pocek, J. Arnold (eds.), Proc. of the IEEE Symp. on FPGAs for Custom Computing Machines – FCCM'98, Apr. 1998, pp. 186-195.

[Zhong98b] P. Zhong, P. Ashar, S. Malik, M. Martonosi, "Using reconfigurable computing techniques to accelerate problems in the CAD domain: a case study with Boolean satisfiability", Proc. of the Design Automation Conf. – DAC'98, pp. 194-199.

[Zhong98c] P. Zhong, M. Martonosi, P. Ashar, S. Malik, “Solving Boolean satisfiability with dynamic hardware configurations”, R.W. Hartenstein, A. Keevallik, (eds), Field-Programmable Logic: From FPGAs to Computing Paradigm, 1998, Lecture Notes in Computer Science 1482, Springer, pp. 326-335.

[Zhong99a] P. Zhong, “Using Configurable Computing to Accelerate Boolean Satisfiability”, Ph.D. dissertation, Department of Electrical Engineering, Princeton University, Jun. 1999.

[Zhong99b] P. Zhong, M. Martonosi, P. Ashar, S. Malik, “Using Configurable Computing to Accelerate Boolean Satisfiability”, IEEE Trans. CAD of Integrated Circuits and Systems, vol. 18, no. 6, Jun. 1999, pp. 861-868.