Universidade do Porto
Faculdade de Engenharia

FEUP

Síntese Automática para Sistemas Digitais Reconfiguráveis Dinamicamente

João Paulo de Castro Canas Ferreira

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

Porto
Setembro 2001

Universidade do Porto
Faculdade de Engenharia

FEUP

Síntese Automática para Sistemas Digitais Reconfiguráveis Dinamicamente

João Paulo de Castro Canas Ferreira

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

Porto
Setembro 2001

Resumo

A presente dissertação descreve a concepção e desenvolvimento do sistema ReDiFlex, um ambiente integrado de apoio ao processo de criação de aplicações baseadas em reconfiguração dinâmica de circuitos FPGA, e que inclui um conjunto de ferramentas automáticas para apoiar todas as etapas do desenvolvimento de uma tal aplicação. O sistema ReDiFlex está orientado para o desenvolvimento interactivo de aplicações que usam um circuito FPGA como co-processador parcialmente reconfigurável.

O sistema ReDiFlex usa um modelo funcional de alto nível para especificar o comportamento da infra-estrutura reconfigurável. O modelo é baseado numa rede acíclica de operadores parametrizados com estado interno acessível e suporta a reconfiguração dinâmica parcial ao permitir alterar em tempo de execução os valores dos parâmetros; cada novo valor origina uma nova implementação. O sistema inclui ainda operadores especiais, os contentores, que podem ser reconfigurados para assumir a personalidade de um entre vários operadores em qualquer altura durante a execução da aplicação. As mutações da implementação são colocadas sob o controlo do utilizador, o que lhe permite fazer depender as reconfigurações de critérios complexos a determinar em tempo de execução e que podem envolver os resultados obtidos no decurso do processamento.

Para cada especificação de alto nível o sistema gera automaticamente a correspondente implementação física, efectuando qualquer decomposição necessária, seja do modelo funcional seja, em alguns casos, da implementação física. Este mapeamento automático entre os domínios funcional e físico preserva a estrutura do modelo funcional, e permite obter uma implementação física dotada de uma estrutura regular.

O sistema faz automaticamente a colocação dos componentes e o encaminhamento das respectivas ligações. A colocação é efectuado por um procedimento heurístico baseado em pesquisa local. O encaminhamento é efectuado por pesquisa exaustiva com *backtracking*. Dada a estrutura regular do modelo físico, as duas tarefas são executadas em tempo compatível com a utilização interactiva do sistema.

O sistema inclui funções para controlar a operação do co-processador durante a execução, tarefa que envolve sequenciar as operações de configuração, transferir os dados para o co-processador e retirar os resultados. É possível usar o co-processador em regime *pipelined*, minimizando o número de ciclos de relógio gastos e de reconfigurações efectuadas.

O sistema também assegura a reconfiguração dinâmica de componentes, coordenando a criação das novas configurações e a reconfiguração parcial do co-processador, garantindo que qualquer modificação de parâmetros de um operador é imediatamente reflectida na configuração do co-processador. O sistema inclui ainda meios de apoio à depuração e à inspecção do estado da infra-estrutura reconfigurável.

O sistema ReDiFlex constitui assim um ambiente interactivo com suporte totalmente automático e integrado para todas as tarefas associadas ao uso de reconfiguração dinâmica parcial.

Abstract

This dissertation describes the design and development of the ReDiFlex system, an integrated environment for the creation of applications based on the dynamic reconfiguration of field-programmable gate arrays (FPGA), that includes a set of automatic tools capable of supporting all the steps of the process. The system is targeted towards the interactive development of applications that use a dynamically reconfigurable FPGA as a co-processor.

The ReDiFlex system employs a high-level functional model for the specification of the reconfigurable co-processor's behavior. The model is based on an acyclic graph of parameterized operators with observable internal state and supports the specification of partial reconfiguration operations, by allowing runtime modification of parameter values: each new value specifies a different implementation. The system also includes special operators, called containers, that can be reconfigured to assume the personality of one of several operators anytime during the execution of the application. The mutation of operators is placed under the direct control of the programmer, allowing him to use complex criteria, which may involve runtime information, to determine when to proceed with the reconfiguration operations.

For each high-level specification, the system automatically generates a corresponding physical implementation, performing automatic partitioning either on the high-level model or, in some cases, directly on the physical implementation. This mapping from functional specification to physical implementation preserves the precedence structure specified by the functional model, and produces a physical implementation with a regular structure that makes placement and routing relatively fast.

The system automatically places all components and routes all the interconnections. Placement is done by an heuristic based on local search. Routing is performed by exhaustive search with backtracking. Due to the regular structure of the physical model, the time taken by both operations is compatible with the interactive use of the system.

The system is capable of automatically generating the control functions that manage the co-processor during runtime. The tasks involved include the scheduling of reconfiguration operations, the feeding of input data and the collection of the results. The implemented circuits can be operated in pipeline mode, thus minimizing both the number of clock cycles spent and the number of reconfigurations performed.

The system also manages the dynamic reconfiguration of the components by coordinating the generation of new configurations and the partial reconfiguration of the FPGA, ensuring that any modification of operator parameters is reflected immediately on the co-processor configuration.

The system includes support for tracing and debugging, including the ability to interactively inspect the state of the hardware.

The ReDiFlex system is an integrated, interactive environment with full automatic support for all tasks associated with the use of partial dynamic reconfiguration.

Résumé

Le travail présenté dans cette dissertation porte sur la conception et realisation du système ReDiFlex, un environnement intégré d'appui à la création d'applications fondées sur l'utilisation de la reconfiguration dynamique, et qui inclut un ensemble d'outils automatiques pour accomplir toutes les tâches associées au développement d'une telle application. Le système ReDiFlex est dédié au développement interactif d'applications faisant usage d'un co-processeur partiellement reconfigurable.

Le système ReDiFlex emploie un modèle fonctionnel de haut niveau pour spécifier les opérations à être exécutées par l'infrastructure reconfigurable. Le modéle est bâti autour d'un réseau d'opérateurs paramètrisés à l'état interne accessible et admet la reconfiguration dynamique partielle en permettant de modifier la valeur des paramètres en temps d'exécution; chaque nouvelle valeur implique la génération d'une nouvelle implantation. Le système inclut aussi des opérateus speciaux, les conteneurs, qui peuvent être reconfigurés pour assumer la personnalité d'un opérateur parmi plusieurs à n'importe quel point de l'exécution. Le changement des implantations est placé sous le contrôle de l' utilisateur, ce qui lui permet de faire dépendre da configuration de critères complexes à verifier en temps d'exécution.

Pour chaque spécification de haut niveau, le système crée automatiquement la correspondante implantation matérielle, réalisant, si nécessaire, la décomposition du modèle fonctionnel ou, en quelques cas, de l'implantation matérielle. La correspondance entre la domaine fonctionnelle et la domaine matérielle préserve la structure du modèle et permet d'obtenir une implantation regulière, rendant plus simples et rapides les tâches de placement et routage.

Le système effectue ces deux tâches automatiquement. Le placement est effectué par un procédé heuristique de recherche locale. Le routage est fait par une recherche exhaustive avec *backtracking*. À cause de la structure regulière de l'implantation matérielle, les deux tâches sont faites en temps compatible avec la nature interactive du système.

Le système fournit aussi des procédés pour contrôler l'opération du co-processeur pendant l'exécution de l'application, ce qui inclut la mise en séquence des opérations de reconfiguration, le transfert de donnés pour le co-processeur et la collection des résultats produits. Le co-processeur peut être contrôlé de façon à travailler en régime *pipeline*, minimisant le numéro de cycles d'horloge nécessaires et le número total d'opérations de reconfiguration.

Le système assure aussi automatiquement la reconfiguration dynamique des opérateurs, en combinant la génération de nouvelles configurations et la reconfiguration partielle du co-processeur, de façon que toutes les modifications des paramètres d'un opérateur se reflètent tout de suite dans la configuration du co-processeur. Le système inclut encore des moyens d'appui à la dépuration et à l'inspection de l'état de la couche reconfigurable.

Le système ReDiFlex constitue un environnement interactif qui supporte de façon completement automatique et articulée toutes les tâches associées à l'utilisation de la reconfiguration dynamique partielle.

Agradecimentos

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

Ao meu orientador, Professor José Silva Matos, agradeço os seus permanentes apoio, motivação e orientação, que foram decisivos na condução dos trabalhos realizados e agora apresentados nesta dissertação.

Aos meus colegas, antigos e actuais, do grupo de CAD e Microelectrónica do INESC Porto, desejo agradecer o bom ambiente de trabalho que sempre proporcionaram e discussões proveitosas que contribuíram para a realização do presente trabalho. Agradeço igualmente a todos os outros colegas da FEUP e do INESC Porto que de uma forma ou de outra me apoiaram durante este trabalho.

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

Conteúdo

Lista de Figuras

Lista de Tabelas

1 Introdução

Em meados dos anos 80 surgiram no mercados circuitos genéricos de elevado nível de integração compostos por blocos lógicos configuráveis e uma rede de interligações, igualmente configurável. Estes circuitos, designados genericamente por "matrizes de portas lógicas programáveis" (*Field Programmable Gate Array–FPGA*), permitem ao utilizador configurar os blocos lógicos e as interligações de maneira a implementar as mais variadas funções digitais. Uma subclasse destes circuitos é caracterizada por armazenar os dados de configuração em células de memória estática, e, consequentemente, permitir um número praticamente ilimitado de reconfigurações muito rápidas. O potencial destes circuitos para a implementação de sistemas computacionais dedicados capazes de serem adaptados à resolução de problemas particulares por simples modificação das configurações foi rapidamente reconhecido e deu origem a uma pujante área de investigação aplicada.

Uma evolução natural desta ideia consiste em modificar as configurações durante a execução de uma tarefa, possivelmente em função dos dados entretanto processados. Esta técnica, conhecida pela designação genérica de "reconfiguração dinâmica", apresenta facetas muito diversas e o seu aproveitamento requer a resolução de numerosos problemas técnicos. Embora surgissem repetidamente na literatura técnica reflexões e estudos de aplicação de várias técnicas de reconfiguração dinâmica, o seu impacto e visibilidade foram sempre relativamente reduzidos.

Contudo trata-se de um conceito com possibilidades sedutoras e em meados dos anos noventa surgiram os primeiros circuitos FPGA comerciais com suporte explícito para reconfiguração dinâmica. Na sua faceta mais comum, esse apoio consiste em permitir reconfigurar parcialmente um circuito FPGA sem alterar o funcionamento da parte não alterada (reconfiguração parcial); alternativamente podem usar-se circuitos com capacidade para múltiplas configurações, apenas uma das quais está activa num dado instante.

Embora a reconfiguração dinâmica seja, estritamente falando, apenas uma técnica de implementação de sistemas digitais, a sua aplicação efectiva requer frequentemente a formulação de uma abordagem específica ao problema concreto a que se deseja aplicá-la. Esta situação, bem como a falta de modelos conceptuais apropriados e a inexistência de ferramentas automáticas de apoio à concepção e realização das correspondentes implementações, contribuíram para a falta de aproveitamento desta técnica apesar das suas características promissoras.

O trabalho descrito nesta dissertação centrou-se na investigação das formas de abordar os

problemas conceptuais e de implementação associados à utilização de reconfiguração dinâmica. Em particular, foi estudada a utilização de circuitos FPGA parcialmente reconfiguráveis a funcionar como co-processador.

Os resultados do trabalho estão concretizados no estudo, concepção e implementação de um sistema automático de apoio ao processo de criação e desenvolvimento de aplicações baseadas em reconfiguração dinâmica, designado por ReDiFlex (REconfiguração DInâmica FLEXível). Foi dado especial valor à produção de uma implementação do sistema, porque apenas uma realização concreta permite avaliar e validar integradamente as numerosas opções que é necessário tomar e a relevância relativa dos diversos problemas a resolver.

O sistema ReDiFlex foi concebido também como um veículo flexível para investigação das aplicações de reconfiguração dinâmica, pelo que flexibilidade de utilização e adaptabilidade foram considerados aspectos importantes. Em particular, pretendeu-se que o sistema permitisse uma utilização interactiva, e suportasse uma actividade de concepção baseada na exploração rápida de diferentes alternativas. Este objectivo impõe restrições ao tempo admissível para a execução das ferramentas automáticas que produzem os dados de configuração.

Por outro lado, pretendeu-se ter um sistema que fosse utilizável por peritos das áreas de aplicação com experiência de programação, mas não necessariamente com prática de concepção de sistemas digitais. O interesse desta característica reside em alargar significativamente o universo de utilizadores potenciais. De facto é amplamente reconhecida na literatura técnica o facto de a exigência de conhecimentos no domínio dos sistemas digitais constituir um entrave significativo à divulgação dos sistemas reconfiguráveis em geral.

Para além da implementação de um protótipo operacional do sistema ReDiFlex, o trabalho aqui descrito apresenta os seguintes aspectos mais relevantes:

- Definição de um modelo funcional (baseado em diagramas de fluxo de dados) a ser usado na especificação do processamento a efectuar pelo co-processador. O modelo é baseado em elementos de processamento abstractos (os operadores), cuja funcionalidade pode ser alterada dinamicamente. O modelo está formulado de maneira a permitir o aproveitamento da capacidade de reconfiguração parcial do co-processador. O modelo funcional está definido de maneira a dar origem a implementações que possam tratar os dados em modo *pipelined*.

  A existência de um modelo de alto nível é também importante por permitir, se correctamente formulado, incorporar a utilização de reconfiguração dinâmica num conjunto de regras orientadoras. O desencadeamento das operações de reconfiguração dinâmica é deixado ao programa executado pelo processador principal, o que permite obter grande flexibilidade na geração de configurações e na sua activação. Em particular, o sistema ReDiFlex permite a geração de configurações durante a execução da aplicação (possivelmente dependendo dos dados já processados pela aplicação), bem como a sua activação sob condições arbitrariamente complexas. O modelo inclui ainda a possibilidade de

definir elementos de processamento capazes de assumirem diferentes "personalidades" durante a execução da aplicação.

- Disponibilização de ferramentas automáticas para tratamento de todas os aspectos de implementação sem intervenção do utilizador. Entre as tarefas consideradas encontram-se a decomposição do modelo funcional, o processamento da informação de nível físico (incluindo a relacionada com as fases de colocação e encaminhamento), a geração dos dados de configuração e a criação de funções de gestão de *hardware* durante a execução. Estas tarefas são geralmente executadas em tempos compatíveis com a natureza interactiva do sistema. A integração completa de todas as fases permite que cada uma se adapte precisamente às necessidades globais do sistema.

- Organização modular do sub-sistema de implementação física, baseado em geradores parametrizados de componentes. A definição de novos elementos básicos de processamento segue um procedimento simples e bem definido.

- Modelo de organização da implementação física adaptado ao modelo funcional. Ao restringir apropriadamente a organização da implementação física, a utilização deste modelo pelas ferramentas de processamento físico permite executar as tarefas de colocação e encaminhamento em tempo útil.

- Implementação de diversas políticas de gestão do co-processador. Em particular, o sistema ReDiFlex permite fazer o processamento em modo *pipeline*, obtendo significativas melhorias do desempenho global, tanto em número de ciclos de relógio gastos, como em número total de operações de reconfiguração efectuadas. A implementação das diversas políticas é quase transparente para o utilizador.

- Integração em ambiente de programação. O sistema ReDiFlex está integrado num ambiente de programação interactivo (baseado na linguagem Common Lisp). Normalmente, o utilizador invoca o co-processador como se invocasse uma qualquer função da linguagem. O sistema ReDiFlex pode mesmo ser visto como uma ferramenta que cria funções da linguagem hospedeira a partir de especificações de processamento em termos de um modelo funcional, com a respectiva implementação a ser executada pelo co-processador reconfigurável.

A forma como o sistema ReDiFlex integra todos estes aspectos de maneira coerente ilustra as mais-valias que se podem obter ao considerar o sistema no seu todo, e não apenas os seus elementos parcelares. Frequentemente, o valor das decisões tomadas em relação a um problema sectorial apenas pode ser apreciado ao considerar o sistema na sua totalidade.

O protótipo ReDiFlex pretende ser uma ferramenta de exploração do universo da reconfiguração dinâmica e da sua implementação. Para além de mostrar a viabilidade das abordagens escolhidas, permite tirar conclusões sobre diversos aspectos da utilização desta técnica, desde

a arquitectura dos dispositivos reconfiguráveis às características dos modelos de especificação.

Contribuições do trabalho

As principais contribuições do trabalho descrito nesta dissertação são:

- Concepção e implementação de um sistema automático completo de apoio ao processo de desenvolvimento de aplicações com reconfiguração dinâmica. O carácter interactivo do sistema permite explorar rapidamente diversas abordagens alternativas a um dado problema.
- Definição de um modelo de especificação comportamental de alto nível que permite a utilização de reconfiguração dinâmica de modo flexível, dentro de limites bem definidos.
- Implementação do tratamento de especificações segundo o modelo mencionado, incluindo a verificação de consistência e a decomposição funcional.
- Criação de um ambiente integrado que permite gerar configurações em tempo de execução (dependentes dos dados em tratamento), bem como fazer a sua activação em qualquer instante.
- Utilização de uma organização para a implementação física que permite o tratamento da informação física (colocação e encaminhamento) em tempos compatíveis com a natureza interactiva do sistema, bem como a concepção e implementação dos algoritmos associados.
- Concepção e implementação de decomposição automática da implementação a nível físico, de forma transparente para o utilizador.
- Disponibilização de funções para a gestão automática do circuito FPGA durante a execução da aplicação, e, em especial, a utilização do circuito FPGA em modo *pipeline*. Estas funções também fazem a sequenciação automática das diversas partes dos circuitos previamente decompostos.

Organização da dissertação

Os restantes capítulos deste trabalho estão organizados em duas partes. A primeira apresenta os circuitos e sistemas reconfiguráveis e faz uma resenha da actividade de investigação nesta área, com uma orientação particular para os problemas da reconfiguração dinâmica. Estes capítulos incluem os resultados de um esforço de sistematização na apresentação dos assuntos tratados, em particular daqueles mais directamente relacionados com reconfiguração dinâmica. Esta área tem sido objecto de actividades menos sistemáticas e os respectivos resultados estão dispersos pela literatura técnica, pelo que se entendeu importante um esforço unificador que permitisse a sua apresentação coerente.

Assim, o capítulo 2 apresenta as principais características tecnológicas e arquitecturais dos circuitos FPGA, bem como dos sistemas neles baseados, classificados segundo critérios funcionais e estruturais

O capítulo 3 trata dos vários aspectos relacionados com a reconfiguração de circuitos e sistemas *durante a execução das aplicações*, descrevendo as principais abordagens anteriores à reconfiguração dinâmica, tanto ao nível dos circuitos individuais como dos sistemas.

Os capítulos restantes constituem a segunda parte da tese e ocupam-se da concepção e implementação de um sistema de apoio ao desenvolvimento de aplicações com reconfiguração dinâmica—o sistema ReDiFlex. Numa situação em que as opções de implementação tomadas nas diversas etapas estão fortemente interrelacionadas, não é possível seguir uma ordem de exposição que evite referir tópicos cujos detalhes são discutidos em capítulos posteriores. Assim, optou-se por organizar a exposição de forma a evoluir dos aspectos mais gerais para os mais particulares, tratando em primeiro lugar dos aspectos mais visíveis ao utilizador e, portanto, de maior importância para estabelecer um modelo conceptual do sistema. A apresentação das estratégias de controlo de execução constitui uma excepção a esta organização: como esta tarefa usa os serviços de todos os outros módulos do sistema, a sua descrição foi deixada para o final.

A segunda parte começa com a descrição dos objectivos gerais que presidiram ao desenvolvimento do sistema ReDiFlex, os diferentes cenários previstos para a sua utilização, a sua organização geral e as principais opções de implementação tomadas (cap. 4). O objectivo é explicitar os principais critérios que presidiram à concepção e implementação do sistema.

Os capítulos seguintes descrevem os principais módulos do sistema. O capítulo 5 trata o modelo funcional usado para especificar a alto nível o comportamento dos circuitos reconfiguráveis dinamicamente. A implementação do modelo é apresentada em detalhe, incluindo a representação interna da especificação, as operações que esta suporta (e os respectivos algoritmos), bem como a estratégia de mapeamento para o domínio físico.

A representação do modelo físico e o seu tratamento são descritos no capítulo 6. Os principais assuntos abordados são o modelo físico do circuito a implementar em *hardware*, a geração de componentes e os algoritmos de colocação e encaminhamento.

O capítulo 7 descreve a geração de configurações a partir do modelo físico e a forma de efectuar o controlo da infra-estrutura reconfigurável (inicialização, controlo do sinal de relógio, transmissão de dados e configurações).

O capítulo 8 descreve o sistema de apoio à execução. Os principais tópicos abordados são: activação e execução (da implementação) do modelo funcional, interface com o sistema de programação hospedeiro, e apoio à depuração e traçagem da execução.

O capítulo final (cap. 9) apresenta algumas reflexões sobre o uso de reconfiguração dinâmica inspiradas pela experiência obtida durante a criação do sistema ReDiFlex. Segue-se ainda um resumo do trabalho realizado e a descrição de possíveis extensões futuras.

A dissertação inclui ainda dois apêndices. O primeiro resume os passos necessários para

proceder à instalação do sistema ReDiFlex num computador dotado do sistema operativo Linux. O segundo inclui uma descrição muito breve da linguagem Common Lisp para ajudar à mais fácil compreensão de alguns fragmentos de código incluídos na descrição do sistema ReDiFlex.

Notas ao leitor

Cada capítulo contém uma pequena introdução e um resumo final. Uma possível estratégia de leitura consiste, numa primeira abordagem, em consultar as introduções e resumos de cada capítulo, eventualmente as conclusões globais do capítulo 9. Desta forma é possível obter-se uma perspectiva geral da dissertação, antes de se passar à leitura detalhada.

Convém também chamar a atenção para o facto de que, embora a segunda parte possa ser lida independentemente da primeira, existem nesta última alguns dados importantes para a compreensão do trabalho posterior. Em particular, deve mencionar-se a descrição da arquitectura interna do circuito FPGA usado (págs. 25 e 39) e do respectivo suporte para utilização de reconfiguração dinâmica (pág. 65). A primeira parte inclui igualmente a descrição de um sistema em que o referido circuito FPGA é empregue, a placa H.O.T. Works (pág. 72) que constitui a infra-estrutura de *hardware* usada pelo sistema ReDiFlex.

Na escrita desta dissertação optou-se naturalmente por traduzir os termos técnicos de origem anglo-saxónica para a língua portuguesa. Contudo em alguns casos utilizam-se termos estrangeiros. Isso ocorre por o autor não conhecer uma tradução adequada, ou por o termo ser de uso tão corrente que a tradução para Português, em vez de ajudar à comunicação das ideias, a vem dificultar. Na primeira situação está, por exemplo, o termo *pipeline*, cujo significado no contexto dos sistemas digitais não é minimamente capturado pela tradução directa "oleoduto". Na segunda situação estão siglas como FPGA e PLD ou termos como *flip-flop*. O termo *tracing* como designação para o processo de obter uma descrição detalhada dos passos executados por um programa constitui uma excepção: nesta dissertação foi substituído por "traçagem", um termo que embora não adoptado pelos dicionários, parece traduzir adequadamente o conceito em causa.

Parte I

Circuitos e sistemas reconfiguráveis

Resumo

Esta parte está dividida em dois capítulos. O primeiro capítulo (cap. 2) apresenta as principais características tecnológicas e arquitecturais dos circuitos FPGA. O objectivo geral é permitir ao leitor avaliar posteriormente a sua influência, frequentemente determinante, nos sistemas em que são incluídos, seja através das possibilidades abertas seja pelas restrições impostas. O capítulo termina com uma classificação dos sistemas que empregam circuitos FPGA segundo critérios funcionais e estruturais, em que é dada atenção especial às formas de que se pode revestir o aproveitamento da capacidade de reconfiguração.

A reconfiguração de circuitos e sistemas *durante a execução das aplicações* suscita questões interessantes de concepção e implementação, que são objecto também do presente trabalho. O capítulo 3 descreve as principais abordagens anteriores à reconfiguração dinâmica, ao nível dos circuitos individuais, dos sistemas que os usam e das ferramentas de apoio. O principal objectivo é estabelecer o contexto para o trabalho descrito na segunda parte.

2 Circuitos e sistemas digitais reconfiguráveis

Os sistemas digitais reconfiguráveis surgem na última década e meia como forma promissora de combinar a eficiência de *hardware* com a flexibilidade de *software*. A reconfigurabilidade, característica definidora desta categoria de sistemas, é proporcionada pelo emprego de componentes do tipo *Field Programmable Gate Array* (FPGA), circuitos genéricos de elevado nível de integração contendo blocos lógicos programáveis. A função de cada bloco e a sua ligação aos demais são estabelecidos através de um procedimento de configuração, que pode ser reversível (circuitos *reconfiguráveis*) ou não (circuitos *configuráveis*).

As sucessivas gerações de circuitos reconfiguráveis vieram fomentar a investigação aplicada em sistemas digitais reconfiguráveis em geral e sistemas computacionais dedicados (*custom computing machines*) em particular.

A organização interna dos circuitos FPGA pode ser considerada uma extensão de soluções anteriores para o problema de aproveitar de forma económica os elevados níveis de integração microelectrónica. O presente capítulo começa por descrever os contornos gerais dessas extensões, estabelecendo assim o pano de fundo para a apreciação das arquitecturas actuais mais importantes.

De seguida apresenta-se uma análise breve das principais características tecnológicas e arquitecturais dos circuitos FPGA. O objectivo principal é permitir ao leitor avaliar a sua influência na organização dos sistemas em que estão incluídos.

As diferentes arquitecturas são comparadas segundo três critérios: tecnologia de configuração, estrutura dos blocos lógicos e organização da rede de interligações. Embora existam muitos outros factores arquitecturais relevantes para a utilização de circuitos FPGA, os três mencionados são certamente os mais determinantes. Numerosos exemplos retirados de circuitos comerciais são usados para ilustrar as diversas alternativas. Entre os exemplos analisados inclui-se a família XC6200 da companhia Xilinx, que constitui o núcleo do suporte físico do trabalho descrito na segunda parte da dissertação.

2.1 Evolução histórica dos circuitos reconfiguráveis

Os níveis crescentes de integração proporcionados pelo avanço da tecnologia microelectrónica colocam permanentemente a questão de como aproveitar de forma mais económica as novas capacidades para obter melhor funcionalidade ou maior nível de desempenho.

Nas fases de pequeno ou médio nível de integração, os sistemas digitais eram maioritariamente constituídos por componentes de catálogo (por exemplo, a família TTL) montados em placas de circuito impresso. Estes componentes realizam funções lógicas simples, mas de vasta aplicabilidade. Contudo, a utilização de componentes de catálogo não é atraente para elevados níveis de integração, porque componentes complexos tendem a exibir uma funcionalidade mais específica, que inviabiliza a sua utilização em grandes quantidades. As soluções encontradas para este problema podem ser classificadas em três categorias gerais:

Microprocessadores e memórias Estes circuitos, embora muito complexos, são componentes de aplicação geral em sistemas programáveis. A evolução exponencial deste sector do mercado mostra claramente o poder desta solução.

Circuitos integrados de aplicação específica Neste caso, opta-se pela concepção de circuitos integrados dedicados a uma aplicação particular. Para reduzir custos e tempo de desenvolvimento são impostas certas restrições à tecnologia usada.

Os circuitos do tipo *gate array* são talvez o melhor exemplo desta abordagem. Estes circuitos são constituídos por agregados genéricos de transístores pré-fabricados, geralmente agrupados em conjuntos com cerca de uma dezena de elementos. A personali-

zação dos circuitos genéricos é feita pela colocação de interligações de metal entre os transístores; assim, apenas as fases de fabrico correspondentes à metalização variam com a aplicação, o que se traduz em importantes reduções de tempo e custo (em comparação com o projecto de todos os dispositivos do circuito). Na sua organização mais simples, designada por *channeled gate array*, as ligações passam por canais livres entre linhas de transístores. Arquitecturas do tipo *sea of gates* são mais flexíveis, porque não usam canais pré-fabricados com dimensões fixadas *a priori*: os transístores cobrem toda a área do circuito, sendo as interligações estabelecidas por cima de transístores não usados.

Circuitos configuráveis Circuitos PROM (*programmable read-only memory*), PAL (*programmable array logic*) e PLA (*programmable logic array*) permitem a implementação de sistemas combinacionais e sequenciais relativamente simples. Trata-se de circuitos fabricados em quantidade, e que podem ser facilmente personalizados pelo utilizador[1]. Estes circuitos são compostos por dois níveis de portas lógicas simples que podem ser configurados através da interrupção ou estabelecimento de contactos eléctricos entre as diversas linhas, usando respectivamente fusíveis ou anti-fusíveis. A alteração da condução eléctrica é geralmente obtida de forma irreversível pela aplicação de uma tensão elevada aos terminais de um elemento apropriado. Posteriormente foram introduzidas versões baseadas em tecnologia EPROM (*Erasable PROM*) ou EEPROM (*Electrically Erasable PROM*) que permitem a reconfiguração repetida dum mesmo circuito (estas tecnologias são descritas na secção 2.2).

Neste tipo de circuitos, o aumento do número de entradas e saídas leva a um crescimento quadrático da área ocupada pela matriz de fusíveis ou anti-fusíveis, o que resulta num aproveitamento ineficiente da área e inviabiliza a utilização da mesma arquitectura em circuitos maiores.

Os circuitos FPGA são geralmente definidos como uma matriz bidimensional de elementos configuráveis, interligados de forma igualmente configurável, e com capacidade para implementar circuitos digitais com vários níveis [Oldfield e Dorf, 1995; Brown *et al.*, 1992; Skahill, 1995].

As principais arquitecturas internas adoptadas para circuitos FPGA podem ser consideradas como adaptações das três abordagens mencionadas [Oldfield e Dorf, 1995]. As categorias correspondentes são (fig. 2.1):

Matrizes computacionais homogéneas São matrizes homogéneas de elementos computacionais simples apropriadas para a implementação distribuída e *pipelined* de algoritmos, e que permitem ultrapassar as limitações do processamento sequencial característico

[1]Neste contexto, entende-se por "utilizador" o projectista ou fabricante do sistema que inclui o circuito configurável, não o utilizador do sistema final.

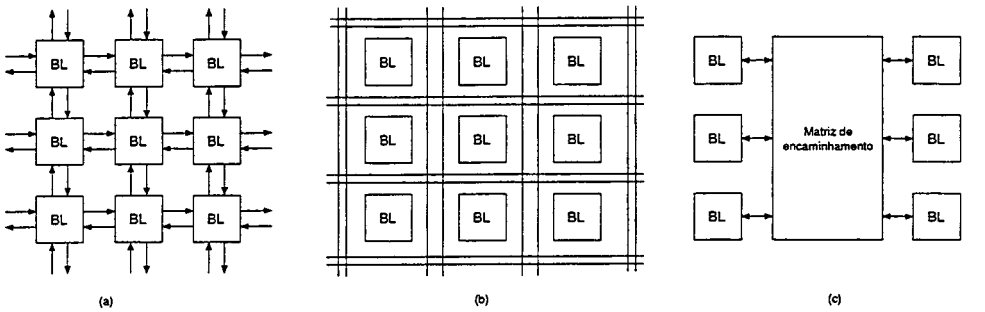


Figura 2.1: Alternativas para a organização interna de circuitos FPGA.

dos microprocessadores. Predominam as ligações directas entre blocos lógicos vizinhos, pelo que os elementos computacionais também desempenham papel importante no estabelecimento das ligações com elementos mais afastados. Embora os princípios organizacionais sejam antigos [Minnick, 1967], os primeiros circuitos FPGA desta categoria foram os da família CAL da companhia Algotronix.

Matrizes de blocos com canais A organização interna destes circuitos é semelhante à dos circuitos *gate array*: blocos lógicos configuráveis são interligados por segmentos metálicos agrupados em canais. As ligações entre os segmentos são igualmente configuráveis. Estes sistemas também são conhecidos como matrizes de estilo insular *(island-style)*, já que os blocos configuráveis constituem pequenas "ilhas" no "mar" dos recursos de interligação. Existe assim uma separação clara entre recursos lógicos e de interligação. A introdução deste tipo de circuitos foi feita pela companhia Xilinx. Trata-se da categoria de maior sucesso comercial.

Agregados lógicos estruturados Estes circuitos contêm vários circuitos PAL (num arranjo bidimensional) interligados por uma matriz de encaminhamento. Existe alguma sobreposição entre esta categoria de circuitos FPGA e os circuitos CPLD (*Complex Programmable Logic Devices*), com alguns autores a classificarem estes últimos também como circuitos FPGA [Brown *et al.*, 1992]. A companhia Altera, fabricante de várias famílias de circuitos CPLD (muitos dos quais classificados como circuitos FPGA por vários autores [Rose *et al.*, 1993; Oldfield e Dorf, 1995]), desempenhou um papel importante na divulgação deste tipo de circuitos, por exemplo com a sua família Flex 10K.

A classificação actualmente mais usada, tanto por companhias do sector como por analistas de mercado, reserva a designação de FPGA para os dispositivos com uma rede de interligações sofisticada, garantindo a possibilidade de interligar vários níveis de blocos lógicos.

Os elementos das matrizes computacionais homogéneas suportam um conjunto restrito de operações lógicas primitivas (i.e., não implementam todas as funções lógicas possíveis da entrada), já que a a filosofia de implementação própria não torna desnecessário suportar a migração de outras tecnologias, como acontece com as matrizes de blocos com canais. Os blocos incluem um elemento de memória, tipicamente um *flip-flop* tipo D. Assim, recursos lógicos, de comunicação e de memória estão distribuídos uniformemente por toda a área do circuito (fig. 2.1a).

É uma arquitectura inerentemente expansível, que equilibra de forma natural recursos lógicos e de interligação. A regularidade e simetria do arranjo espacial favorece a implementação de circuitos sistólicos e *pipelined*, além de facilitar a tarefa das ferramentas de apoio. A utilização de blocos lógicos simples permite a sua inclusão, para efeitos de configuração, em memórias RAM estáticas, o que permite acesso directo à memória de configuração e simplifica grandemente a reconfiguração parcial do circuito. A família Xilinx XC6200 explora estas possibilidades de modo particularmente eficaz. Entre os produtos comerciais que implementam esta abordagem contam-se igualmente as famílias ATMEL AT6000 e AT40K .

Na segunda classe de circuitos FPGA (matrizes de blocos com canais) existe uma clara separação entre recursos lógicos (blocos configuráveis) e de comunicação, que reflecte a separação análoga existente nos circuitos *gate array*. Como nestes, as interligações são constituídas por segmentos de comprimento variável que são ligados entre si de maneira a estabelecer os percursos desejados. Para reduzir a necessidade de recursos de interligação, particularmente em circuitos programáveis por memória estática, os blocos configuráveis são tendencialmente complexos e têm um número relativamente elevado de entradas. A existência de linhas internas de alta impedância facilita a migração de sistema implementados em placas de circuito impresso e em circuitos *gate array* para este tipo de circuitos FPGA. Pela mesma razão, a interface de configuração é geralmente concebida para usar poucos pinos externos. Por último, é de notar que a arquitectura interna destes circuitos se adapta bem às ferramentas de projecto assistido por computador usadas com circuitos integrados de aplicação específica (CIAE).

No terceiro grupo, os agregados lógicos estruturados são constituídos por circuitos PAL relativamente pequenos interligados por uma matriz de encaminhamento. Como o tempo de atraso de um circuito PAL aumenta significativamente com o número de entradas, é inviável manter bons níveis de desempenho para circuitos muito grandes. A organização em sub-circuitos PAL é, por isso, muito mais eficiente em termos do número de elementos de configuração do que um único circuito PAL com o mesmo número de entradas e saídas. A utilização de interligações não-segmentadas ajuda a preservar os tempos de atraso baixos e previsíveis das implementações em dois andares. Contudo, a interligação através de uma matriz de encaminhamento não é capaz de acompanhar proporcionalmente o crescimento do número de blocos lógicos [Brown *et al.*, 1992], o que impõe alguns limites à complexidade dos sistemas a implementar.

Os circuitos PAL estão particularmente vocacionados para unidades de controlo (tipo má-

quina de estados finita), mas, em contrapartida, não têm a distribuição de recursos mais adaptada para unidades de processamento complexas (*datapaths*), ou aplicações que requeiram um número elevado de elementos de memória (implementações sistólicas e *pipelined*). Para estes tipos de aplicações, os outros tipos de FPGA apresentam características mais favoráveis, embora a complexidade das interligações torne os tempos de atraso menos previsíveis.

É previsível que a evolução das arquitecturas de circuitos FPGA tenda a diluir estas diferenças, à medida que vão surgindo organizações híbridas, capazes de combinar as melhores características de cada uma. Uma outra tendência, que aliás já começa a desenhar-se, consiste na inclusão de blocos especializados, por exemplo RAM estática ou circuitos PLL (*phase-locked loop*). Esta tendência acentuar-se-á à medida que os fabricantes tentam tornar prática a implementação de sistemas completos num único circuito programável de grande capacidade (*system-on-a-chip*).

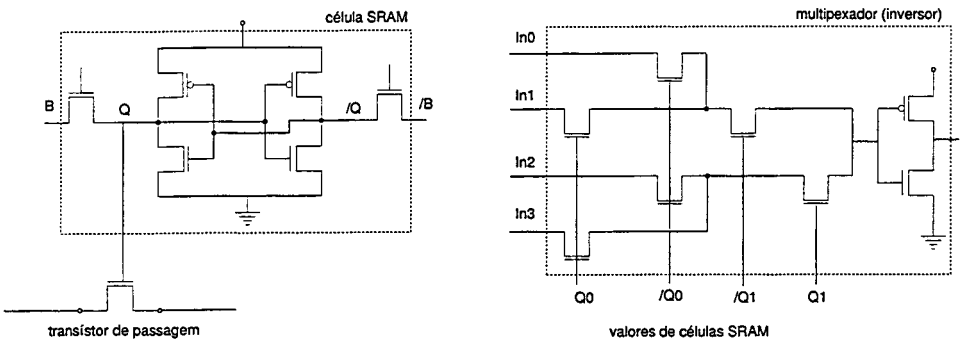
Se a exposição precedente permite compreender a razão de ser das actuais arquitecturas de circuitos FPGA, não permite, contudo, avaliar fundamentadamente os factores que entram no aproveitamento deste tipo de circuitos nem o seu impacto nos sistemas que integram. Para isso é preciso examinar mais detalhadamente as arquitecturas existentes. As secções seguintes são dedicadas a essa tarefa, ainda que de maneira necessariamente breve. São focados três aspectos: tecnologia de configuração, arquitectura dos blocos lógicos e a organização da rede de interligações.

2.2 Tecnologias de configuração

Actualmente utilizam-se três tecnologias diferentes para configuração de circuitos FPGA: células de memória estática, anti-fusíveis ou transístores com porta flutuante [Rose *et al.*, 1993; Brown *et al.*, 1992; Oldfield e Dorf, 1995]. Todas são implementadas em tecnologia CMOS. As características eléctricas, nomeadamente resistência de condução e capacidade parasita, as dimensões físicas e a reconfigurabilidade são os principais factores a ter em conta na avaliação dos diferentes métodos de configuração.

Existem duas classes de recursos a configurar, blocos lógicos e interligações. Para o mesmo circuito FPGA, ambas as funções devem ser realizadas pela mesma tecnologia. Interligações são tipicamente configuradas por dispositivos tipo interruptor, que estabelecem continuidade eléctrica entre dois segmentos. Alternativamente a configuração das interligações pode ser obtido pela utilização de multiplexadores para encaminhar sinais provenientes de origens diferentes (multiplexadores também podem ser usados para implementar funções lógicas).

Os blocos lógicos são personalizados através da configuração das interligações de elementos fixos (p.ex., portas NAND ou até mesmo transístores individuais [Marple, 1992]) e, dependendo da tecnologia, pela utilização de tabelas de memória (*look-up tables*) para a definição de funções lógicas .



(a) Célula de memória controla transístor de passagem

(b) Multiplexador controlado por células RAM

Figura 2.2: Célula de memória estática.

Células de memória estática

Uma possível tecnologia de configuração baseia-se na utilização de células de memória estática para controlar portas de passagem ou multiplexadores, ou para formar tabelas (fig. 2.2). Uma porta de passagem é implementada por um transístor MOS operando como um interruptor: o valor da célula RAM controla o estado do interruptor, que, quando fechado, estabelece a continuidade eléctrica entre dois segmentos de interligação. No caso dos multiplexadores, o valor da célula de memória determina qual das entradas fica ligada à saída. Alternativamente, os blocos de células podem ser usados para realizar pequenas tabelas (*lookup-tables*) capazes de implementar funções lógicas de algumas variáveis (normalmente três ou quatro): os valores de entrada seleccionam uma célula, cujo conteúdo representam o respectivo valor da função. Nesta tecnologia, a configuração é naturalmente volátil, mas pode ser feita muito rapidamente e um número praticamente ilimitado de vezes. Estas características fazem com que circuitos FPGA baseados em memória estática formam o núcleo de todos os sistemas digitais reconfiguráveis (SDR). Pode mesmo afirmar-se que foi o aparecimento de circuitos com estas características que fomentou o ressurgimento actual dos computadores dedicados.

Esta tecnologia tem a grande vantagem de usar processos CMOS convencionais, sem exigir etapas especiais durante o fabrico; pode assim aproveitar rapidamente os progressos feitos no fabrico de microprocessadores e memórias, geralmente os componentes que impulsionam os avanços da tecnologia de fabrico.

A maior desvantagem desta tecnologia reside no facto de a célula RAM ocupar uma área relativamente grande: cinco ou seis transístores sem contar com a porta de passagem. Em comparação com as células de seis transístores, as células de cinco são mais pequenas e permitem um arranjo físico mais denso. Contudo, o seu projecto é mais difícil e a imunidade ao

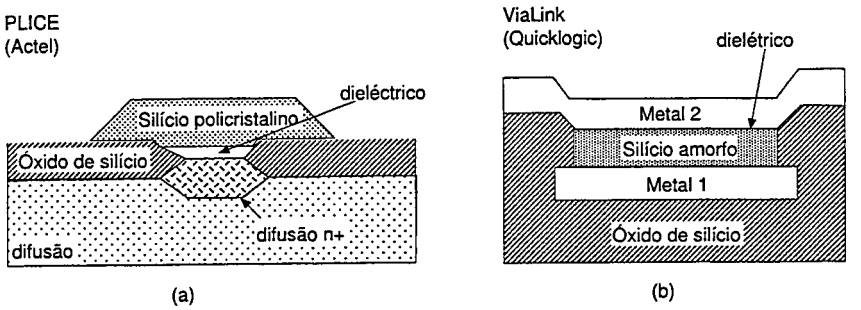


Figura 2.3: Anti-fusíveis: (a) tecnologia ONO; (b) silício amorfo.

ruído sensivelmente menor, pelo que são geralmente preteridas a favor da versão mais estável em processos CMOS submicrométricos e com tensões de alimentação iguais ou inferiores a 3 V.

Anti-fusíveis

O anti-fusível funciona como um interruptor normalmente aberto, apresentando uma grande resistência eléctrica entre os seus dois terminais. A aplicação de uma tensão elevada (10–20 V, dependendo da tecnologia) aos seus terminais força o estabelecimento permanente de um percurso de baixa resistência entre eles. Comporta-se assim de maneira oposta a um fusível, daí a sua designação. Existem dois tipos de anti-fusível em uso nos circuitos FPGA (fig. 2.3), qualquer um deles composto por uma camada de dieléctrico colocado entre duas camadas condutoras.

Nos anti-fusíveis ONO o dieléctrico é um composto multi-camada óxido/nitrido/óxido colocada entre uma camada de difusão N+ e uma camada de poli-silício (fig. 2.3a). Os dois terminais são ligados a condutores de metal.

A versão deste anti-fusível usada pela companhia ACTEL nas famílias ACT-1, ACT-2 e ACT-3 é designada por PLICE *(programmable low-impedance circuit element)*. Num processo CMOS 0.6 μm, a dimensão de um anti-fusível é de 0.6 μm^2 e a sua resistência de condução varia entre 100 MΩ (anti-fusível intacto) e 150–200 Ω. A tensão de programação é de 18 V.

Outro tipo de anti-fusível é composto por silício amorfo colocado entre dois níveis de metal ou entre poli-silício e o primeiro nível de metal (fig. 2.3b). A companhia Quicklogic usa este dispositivo, sob a designação de ViaLink, na sua família pASIC 3. A resistência de condução varia entre 1 GΩ e 80 Ω e a tensão de programação é de 10 V.

A maior vantagem dos anti-fusíveis é a sua pequena área, equivalente à área ocupada pelo contacto entre duas pistas de metal. Esta vantagem é algo atenuada pela área ocupada pelos circuitos de programação, cujos transístores assumem dimensões relativamente importantes devido à necessidade de operarem sob tensões e correntes elevadas (superior a 10 V e 5 mA,

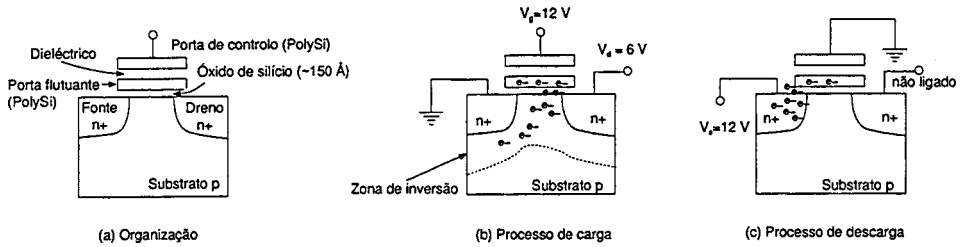


Figura 2.4: Transístor de porta flutuante.

respectivamente). Baixa resistência de condução e capacidades parasitas diminutas são outras importantes vantagens, porque se traduzem directamente em pequenos tempos de atraso nas interligações.

A irreversibilidade do processo de configuração determina fortemente as áreas de aplicação. Naturalmente, os circuitos FPGA usados em sistemas digitais reconfiguráveis não podem empregar esta tecnologia de configuração. As boas características eléctricas associadas a este tipo de tecnologia levam ao seu emprego em áreas até agora reservadas a circuitos *gate array*, com importantes benefícios de flexibilidade, tempo de projecto e mesmo de preço (para pequenas quantidades, por exemplo, menos de 1000 unidades por ano).

Para o fabrico de anti-fusíveis é necessário acrescentar três passos suplementares a um processo CMOS convencional, o que faz aumentar o preço final e, principalmente, atrasa o aproveitamento das melhorias do processo base em cerca de um ano [Oldfield e Dorf, 1995].

Muitos circuitos PLD de baixa densidade, implementados em tecnologia bipolar, usam fusíveis metálicos (normalmente fechados) como meio de configuração. Esta tecnologia não é usada em circuitos CMOS de elevada densidade por duas razões principais:

Testabilidade A utilização de dispositivos normalmente abertos permite o teste individual de blocos lógicos e segmentos de interligação antes da configuração.

Rapidez de programação Em geral é preciso alterar apenas um pequeno número de anti-fusíveis. Por exemplo, uma aplicação que use 85% dos recursos lógicos do maior elemento da família ACT-3 (com cerca de 1 250 000 anti-fusíveis) apenas necessita de alterar 2–3% dos dispositivos.

Transístores com porta flutuante

Transístores de porta flutuante (fig. 2.4) são usados como elementos de de memória em alguns circuitos FPGA. O princípio de funcionamento é similar ao das memórias EPROM, EEPROM ou FLASH. Transístores com porta flutuante têm uma estrutura similar a um transístor MOS com uma porta adicional colocada sob a porta convencional (porta de controlo) e electricamente isolada do restante circuito. A aplicação de uma tensão elevada à porta de controlo

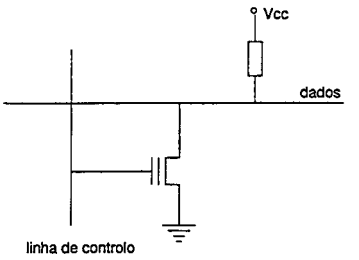


Figura 2.5: Modo de utilização de transístores de porta flutuante.

Tecnologia	Família de FPGA	Fabricante
Memória estática	AT6000, AT40K	Atmel
	XC6200, XC4000	Xilinx
	Orca	Lucent
	VF1	Vantis
	Flex 10K	Altera
Anti-fusível	ACT-3, 54MX	Actel
	pASIC 3	Quicklogic
Porta flutuante	ProASIC	Actel

Tabela 2.1: Tecnologia de configuração de diversas famílias de circuitos FPGA.

(em simultâneo com uma tensão de cerca de 6 V entre fonte e dreno) provoca uma injecção de carga na porta flutuante, onde fica capturada (fig. 2.4b). O aumento de carga eléctrica armazenada faz subir a tensão de limiar de condução do transístor, de forma a que este permanece no estado de não-condução na presença das tensões normais de operação. A carga adicional pode ser removida da porta flutuante por exposição à luz ultravioleta (tecnologia EPROM) ou por meios eléctricos (tecnologia EEPROM e FLASH), conforme ilustrado na fig. 2.4c. Em qualquer dos casos, o transístor funciona normalmente após a remoção da carga. As células de memória EPROM e FLASH ocupam áreas semelhantes; células EEPROM necessitam de cerca do dobro.

A forma de utilização mais comum de um transístor com porta flutuante também está ilustrada na figura 2.5. Com a porta flutuante descarregada, o transístor pode ser usado para levar a linha de dados ao nível baixo; no caso contrário a linha fica sempre ao nível alto por acção da resistência de *pull-up*. Este arranjo pode ser usado simplesmente para estabelecer a ligação entre as duas linhas ou para implementar funções lógicas no estilo *wired-AND*. A organização é similar à dos circuitos PAL, pelo que esta tecnologia é usada predominantemente em agregados lógicos estruturados, como a família MAX 9000 da companhia Altera. Células de memória FLASH também são empregues na família ProAsic da companhia Gatefield (mais

recentemente comercializada pela Actel), que tem uma organização muito semelhante aos circuitos tipo *sea-of-gates*. A principal vantagem desta tecnologia é permitir a reconfiguração, mas não ser volátil. Em termos de área, as células tipo EPROM ou FLASH ficam entre o anti-fusível e a célula RAM, e podem ser organizadas em matrizes densas. Células EEPROM levam naturalmente a arranjos menos densos. O processo de reconfiguração é mais demorado que no caso das células RAM, o que desencoraja a sua utilização em SDR e, particularmente, em aplicações de reconfiguração dinâmica. A utilização de resistências de *pull-up* resulta num consumo de energia superior ao das tecnologias alternativas. O fabrico de transístores com porta flutuante requer passos adicionais em relação ao um processo CMOS convencional: três no caso de circuitos EPROM ou FLASH, e cinco no caso de circuitos EEPROM. Como no caso dos anti-fusíveis, esta circunstância atrasa o aproveitamento dos avanços registados nos processos de fabrico convencionais. A tabela 2.1 resume quais as tecnologias de programação empregadas em diversos circuitos FPGA comerciais.

2.3 Arquitectura dos blocos lógicos

As arquitecturas de circuitos FPGA existentes são muito variadas, já que resultam de numerosas opções, cada uma influenciada por um elevado número de factores, desde a tecnologia de configuração até às áreas de aplicação previstas. Talvez a melhor forma de identificar os diversos factores consista na análise das arquitecturas mais representativas e na apreciação das opções tomadas por cada fabricante. Os exemplos escolhidos pretendem ser representativos das diversas abordagens; embora necessariamente sucinta, a descrição tenta ser abrangente sem ser exaustiva.

Embora os circuitos FPGA tenham apenas cerca de quinze anos (os primeiros foram anunciados em 1985), os fabricantes que estão neste sector desde o princípio já evoluíram através de quatro ou cinco gerações de circuitos, progressivamente mais complexas. No que se segue nem sempre se optou por descrever as versões mais recentes de cada arquitectura, já que frequentemente são as versões iniciais que ilustram melhor os factores em jogo.

2.3.1 Alternativas de organização

A dimensão dos blocos lógicos pode ser usada como um critério de classificação de circuitos FPGA. Blocos pequenos facilitam o aproveitamento dos recursos lógicos; com blocos complexos, é frequente parte dos seus recursos não ser utilizada numa dada configuração. Por outro lado, o número de blocos necessários para implementar uma dada funcionalidade é superior em arquitecturas de granularidade fina pelo que a existência de recursos de interligação mais abundantes é obrigatória neste caso [Rose *et al.*, 1993]. A tabela 2.2 mostra a classificação segunda a granularidade das principais arquitecturas mencionadas nesta secção.

Granularidade	Família de FPGA	Fabricante
Fina	CP20K	Crosspoint
	ACT-3	Actel
	AT6000	Atmel
	XC6200	Xilinx
Grossa	pAsic 3	Quicklogic
	AT40K	Atmel
	XC4000	Xilinx
	XC5200	Xilinx
	Flex 10K	Altera

Tabela 2.2: Granularidade de diversas famílias de circuitos FPGA.

Crosspoint CP20K

O primeiro exemplo é um circuito FPGA de granularidade muito fina com a estrutura praticamente idêntica à de um circuito *gate array*. A família CP20K da companhia Crosspoint Solutions tem uma organização baseada em pares de transístores, conforme ilustrado na figura 2.6a [Marple, 1992]. O circuito é configurado pela interligação apropriada de segmentos de metal por antifusíveis de silício amorfo. A maior parte dos segmentos agrupam-se em canais situados entre as linhas de pares de transitares. A dimensão pequena do anti-fusível e as boas características eléctricas (resistência de condução de 100 Ω e capacidade de 1.1 fF) ajustam-se bem à arquitectura escolhida.

Como se ilustra na figura 2.6b, o isolamento entre portas lógicas é conseguido por transístores colocados permanentemente no estado de não-condução por polarização adequada da respectiva porta (ligada a Vcc no caso de transístores P e a Gnd para transístores N). Os circuitos CP20K contêm ainda um outro tipo de bloco lógico, mais vocacionado para a implementação de células de memória estática e outros elementos bi-estáveis (fig. 2.6c).

Actel ACT

A companhia Actel produz diversas famílias de circuitos FPGA, todas de granularidade fina. Três gerações sucessivas da mesma arquitectura (ACT-1, ACT-2 e ACT-3) têm blocos lógicos simples, baseados em multiplexadores. O elemento de configuração é o antifusível PLICE. A figura 2.7 mostra os blocos das três séries. A família ACT-1 tem apenas um tipo de bloco lógico combinacional (fig. 2.7a); a realização de circuitos sequenciais é feita por realimentação. Já as famílias ACT-2/ACT-3 têm primitivas sequenciais (fig. 2.7b): cerca de metade dos blocos lógicos tem um *flip-flop* tipo D configurável.

As três famílias têm uma organização muito parecida com a dos circuitos *gate array*. A principal diferença está na maior complexidade dos blocos lógicos. O objectivo é optimizar o desempenho para as operações lógicas mais comuns, permitindo a sua implementação sem colocar anti-fusíveis no percurso crítico. Esta opção é justificada pela resistência de condução

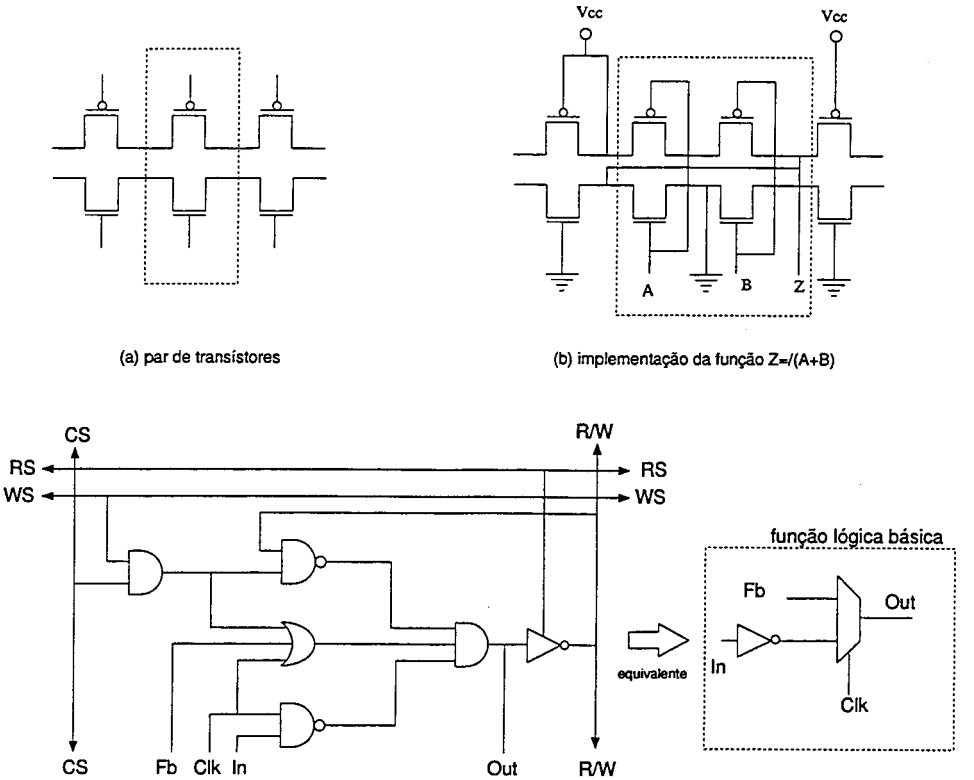
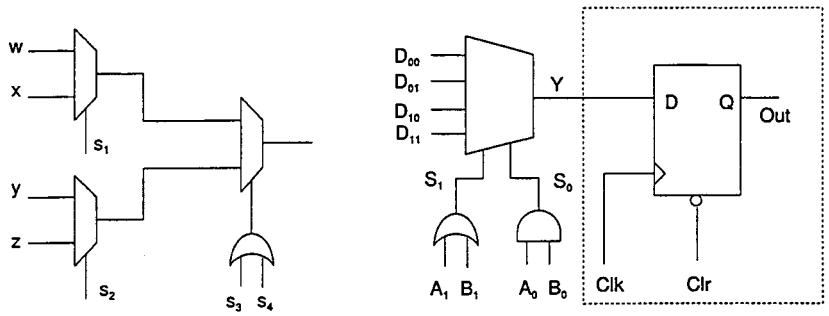


Figura 2.6: Bloco lógico da família CP20K.

dos anti-fusíveis, que, embora baixa, ainda é cerca de dez vezes superior à dos contactos metal-metal dos circuitos *gate array*. A presença de recursos de interligação abundantes é importante para aproveitar o elevado número de entradas [Skahill, 1995].

Atmel AT6000

Outro exemplo de granularidade fina é a família AT6000 da companhia Atmel. Programável por memória estática, esta família pertence à categoria das matrizes computacionais homogéneas. A arquitectura regular e simétrica está orientada para aplicações de cálculo *pipelined* como, por exemplo, em processamento digital de sinal, e para reconfiguração dinâmica. Os blocos lógicos estão dispostos uniformemente numa matriz quadrada. Cada um é composto por multiplexadores, portas lógicas e portas de transmissão, conforme está ilustrado na figura 2.8.



(a) bloco combinacional (b) bloco combinacional e sequencial

Figura 2.7: Blocos lógico das famílias ACT.

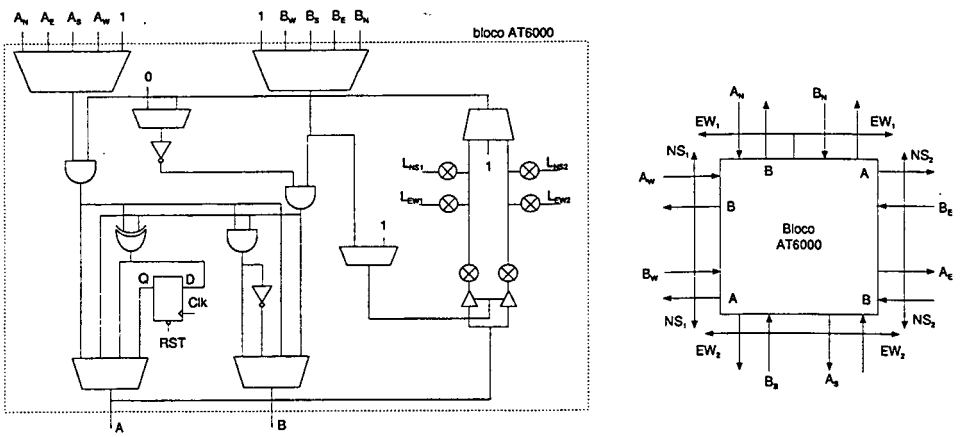


Figura 2.8: Bloco lógico da família AT6000.

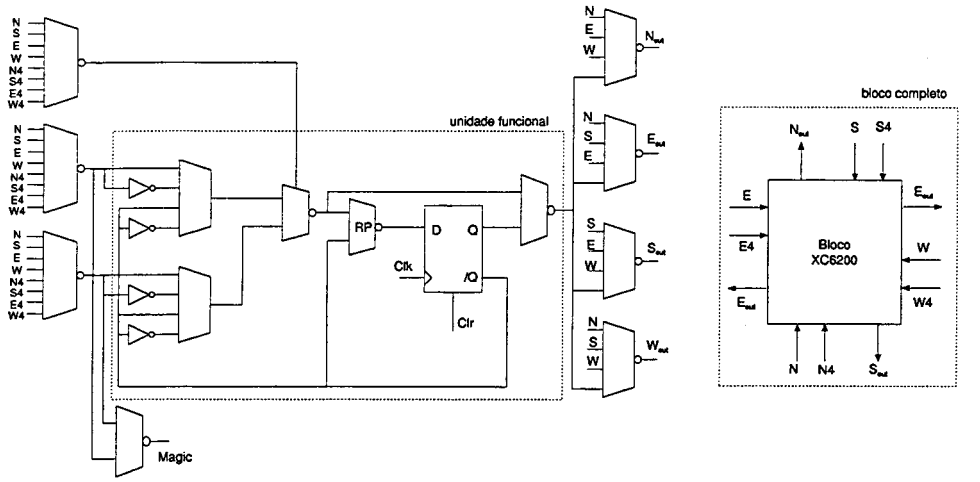


Figura 2.9: Bloco lógico da família XC6200.

Os multiplexadores são usados para encaminhamento de sinais, tanto de entrada (multiplexadores 5:1) como de saída (multiplexadores 4:1). O núcleo funcional da célula é constituído por portas lógicas fixas e um *flip-flop* tipo D. Entre as portas lógicas, ressalta uma porta EXOR de duas entradas, que facilita a implementação compacta de circuitos aritméticos.

O bloco de portas de transmissão permite a implementação de barramentos internos com linhas de alta impedância. Também permite o encaminhamento de sinais através do bloco sem afectar a implementação da função lógica. A combinação de funções lógicas e de encaminhamento num mesmo bloco é típica das matrizes computacionais homogéneas; já a presença de linhas de alta impedância é uma propriedade específica desta arquitectura.

A estrutura simples e regular favorece a utilização de geradores de componentes como ferramenta de apoio privilegiada; a uniformidade também é importante para aplicações de reconfiguração dinâmica (cf. secção 3.2.1).

Xilinx XC6200

A companhia Xilinx é outra pioneira do sector. A família XC6200 constitui uma implementação de matrizes computacionais homogéneas de granularidade fina. Esta arquitectura descende da família CAL da companhia Algotronix. Um elemento desta família, o circuito XC6216, constitui o núcleo do sistema de *hardware* que serve de veículo ao trabalho descrito na segunda parte da dissertação.

Este circuito está particularmente vocacionado para aplicações de reconfiguração dinâmica (secção 3.2.1): tem uma estrutura regular e totalmente simétrica nas duas dimensões, é configurado por células de memória estática e permite acesso directo aos dados de configuração e

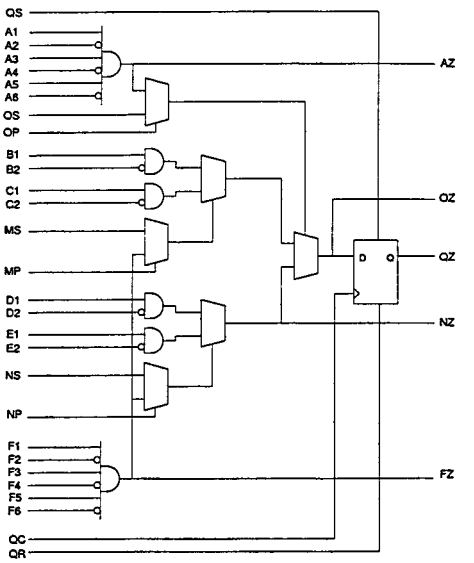


Figura 2.10: Bloco lógico da família Quicklogic.

ao conteúdo dos elementos de memória.

A arquitectura do bloco lógico XC6200 está representada na figura 2.9. Como no caso anterior, o bloco é composto por recursos de encaminhamento e por uma unidade funcional, que implementa qualquer função booleana de duas variáveis. O bloco é composto por multiplexadores, incluindo ainda um *flip-flop* tipo D. Os valores de entrada provêm das quatro células vizinhas (sinais N, S, E, W) ou de quatro segmentos partilhados por grupos de 4×4 blocos (sinais N4, S4, E4, W4). As saídas para os blocos vizinhos são independentes, ao contrário do que sucede na família AT6000. A utilização de multiplexadores nas entradas e saídas do bloco permite combinar o encaminhamento de sinais com a execução de operações lógicas. A saída da unidade funcional pode ser opcionalmente registada; a unidade inclui ainda um percurso de realimentação interno.

O multiplexador RP é usado para "proteger" o valor do *flip-flop*, ou seja, para impedir a sua alteração por influência do circuito implementado. Esse valor só pode ser acedido, para alteração, através da interface de acesso directo do circuito XC6200. É uma característica muito conveniente, que garante o acesso sem conflito aos valores de registos mesmo quando não existe sincronização entre o acesso e a operação do circuito.

Quicklogic pASIC3

O primeiro exemplo de circuitos FPGA de maior granularidade é a família pASIC3 da companhia Quicklogic. Esta família utiliza o antifusível ViaLink colocado entre o terceiro e o

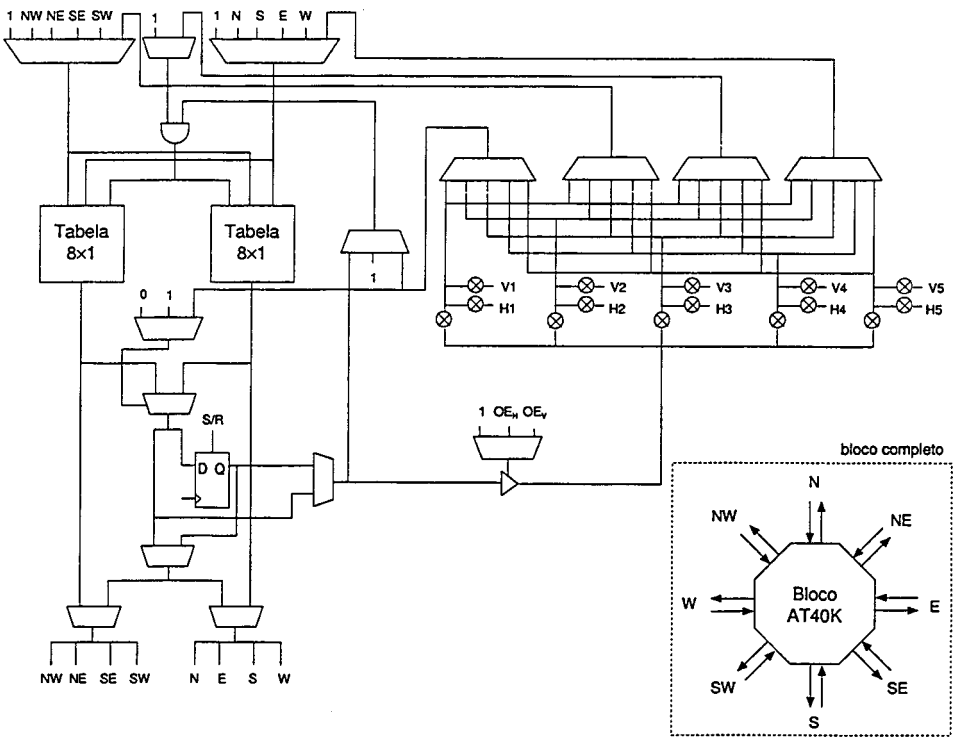


Figura 2.11: Bloco lógico da família AT40K.

quarto níveis de metal. A reduzida dimensão dos antifusíveis, bem como o facto de não ser necessário ter canais para interligações, porque os antifusíveis são colocados sobre as células, permitem uma grande abundância de recursos de interligação e, consequentemente, a utilização de blocos lógicos com elevado número de entradas. O bloco lógico pASIC 3 (figura 2.10) é constituído por portas lógicas AND (duas de 6 entradas e quatro de 2 entradas), cinco multiplexadores e um elemento de memória configurável como *flip-flop* tipo D, RS, JK ou T. Sem contar com os sinais de controlo do elemento de memória, o bloco lógico tem 26 entradas e 5 saídas. A funcionalidade do bloco lógico é fixa; a configuração é efectuada por ligação apropriada das entradas. Como cada bloco dispõe de várias saídas, pode implementar simultaneamente várias funções booleanas.

Atmel AT40K

A família Atmel AT40K representa uma tentativa de aplicar os princípios das matrizes computacionais homogéneas (tal como na família AT6000) a circuitos de maior granularidade. A configuração é feita por células de memória estática. O bloco lógico desta família (fig. 2.11)

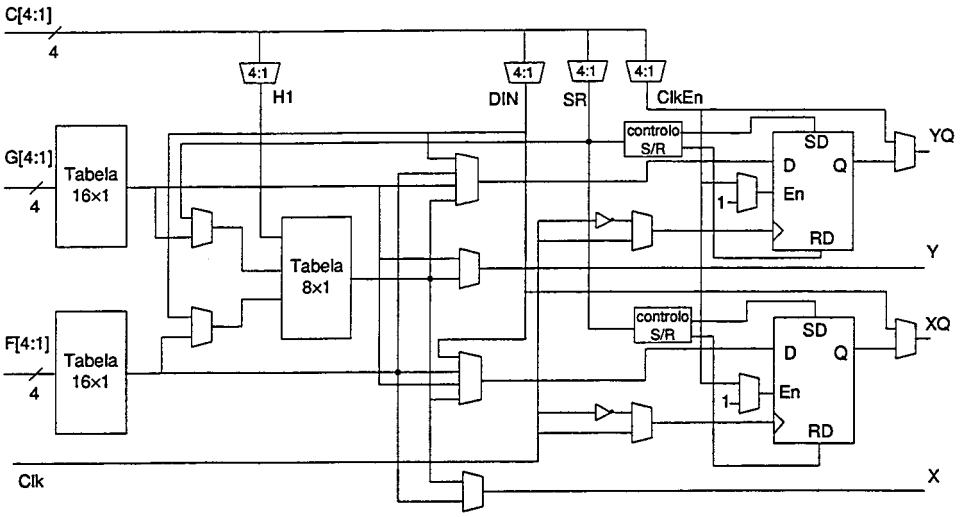


Figura 2.12: Bloco lógico simplificado da família XC4000.

tem ligações directas aos seus oito vizinhos. Tal como acontece com outras matrizes computacionais homogéneas, o bloco inclui recursos lógicos e de encaminhamento. Os principais recursos lógicos são duas tabelas de memória estática de três entradas e um *flip-flop* tipo D. Multiplexadores são usados para seleccionar os sinais de entrada. O bloco contém ainda uma linha interna de realimentação. Um bloco de portas de passagem e multiplexadores permite realizar funções de encaminhamento e linhas internas de alta impedância. Cada bloco tem ligação directa a dois barramentos de cinco linhas, um vertical (v1–V5) e um horizontal (H1-H5). Cada circuito FPGA inclui ainda blocos de RAM 32×4 bits com dois portos de acesso. A organização interna permite a realização de somadores e multiplicadores sem utilização de barramentos (apenas interligações locais). À semelhança da família AT6000, também estes circuitos incluem suporte para reconfiguração dinâmica.

Xilinx XC4000

A família XC4000 da Xilinx é possivelmente a mais divulgada e a de maior sucesso comercial; grande parte dos sistemas reconfiguráveis utilizam elementos desta família. Trata-se da terceira geração de uma arquitectura baseada em blocos lógicos de granularidade reduzida, rodeados por canais de interligação verticais e horizontais. Existem várias famílias de organização semelhante, mas com características eléctricas e capacidades sistematicamente melhoradas; a descrição seguinte aplica-se a todas elas. Cada bloco lógico (fig. 2.12) é composto por dois geradores de funções (tabelas) de quatro entradas ligados a um gerador de três entradas. Existem ainda dois elementos de memória e um conjunto de multiplexadores que

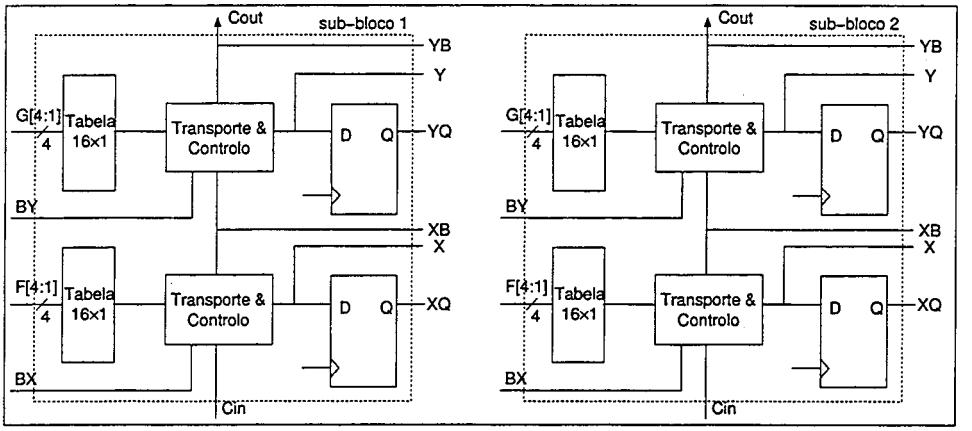


Figura 2.13: Bloco lógico simplificado da família Virtex.

fazem a interligação das saídas dos geradores aos elementos de memória e às saídas primárias. Cada elemento de memória pode operar como *latch* ou *flip-flop*.

Os geradores de funções (tabelas de memória estática— *look-up table*) produzem o valor booleano da função para cada combinação dos valores de entrada. Os tempos de atraso associados a cada gerador são independentes da função lógica implementada. Para além dos recursos ilustrados, cada bloco inclui circuitos dedicados para suporte de operações aritméticas (*carry logic*) e circuitos para usar os geradores de funções como pequenos blocos de memória. Vários blocos assim configurados podem ser combinados para construir blocos RAM de maior capacidade. Os blocos lógicos são interligados por uma rede complexa de segmentos de vários comprimentos cuja interligação é controlada por células de memória estática.

Os circuitos desta família são facilmente reconfiguráveis e permitem a realização de circuitos complexos envolvendo um número elevado de registos. Todas estas características são importantes para a realização de sistemas reconfiguráveis.

Esta arquitectura exemplifica bem a tendência de usar blocos complexos para compensar o tamanho dos elementos de programação: blocos complexos tendem a diminuir o número de segmentos de intercomunicação usados, o que por sua vez diminui a área ocupada pelas células de controlo dos respectivos transístores de passagem.

A família Virtex da mesma companhia representa um patamar superior ao longo da mesma tendência, bem como a tendência concomitante de implementar um bloco lógico como um agrupamento de vários sub-blocos, conforme se mostra na figua 2.13. A proximidade entre sub-bloco vizinhos favorece a respectiva interligação eficiente. O bloco lógico Virtex é composto por dois grupos de dois sub-blocos: cada grupo é de complexidade semelhante a um bloco lógico XC4000.

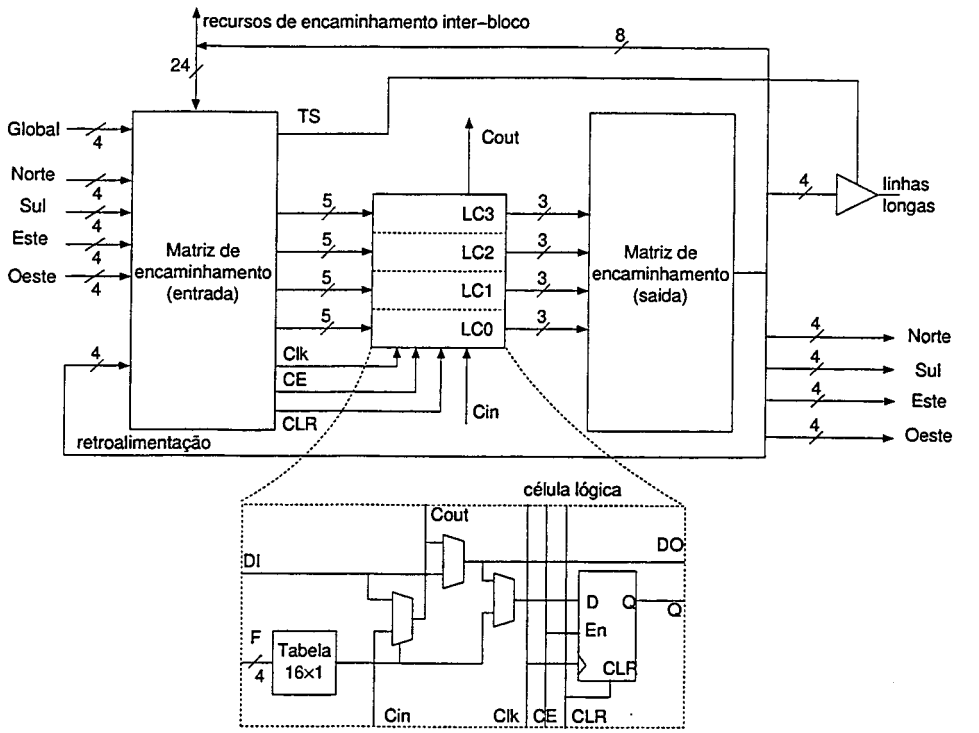


Figura 2.14: Bloco lógico da família XC5200.

Xilinx XC5200

A família Xilinx XC5200 pretende ser uma alternativa de custo inferior à família XC4000. Igualmente configurável por células de memória estática, a sua organização interna combina uma arquitectura tipo *gate array* com alguns princípios das matrizes homogéneas, nomeadamente ligações directas entre vizinhos. Conforme está ilustrado na figura 2.14, cada bloco lógico é constituído por quatro células, por sua vez compostas por uma tabela de quatro entradas e um *flip-flop* tipo D. A célula pode ainda operar em modo transparente (ligação directa da entrada para a saída). As quatro células de cada bloco partilham as mesmas linhas de controlo (não ilustradas na figura). As entradas das células provêm de uma matriz de encaminhamento local *(Local Interconnect Matrix–LIM)* que permite seleccionar seleccionar os sinais de entrada de entre dezasseis sinais de blocos vizinhos, quatro sinais globais fixos, quatro sinais realimentados e um conjunto de sinais bidireccionais provenientes das interligações de nível superior. Cada bloco tem oito saídas cujos sinais podem ser escolhidos de entre todos os doze sinais produzidos pelas células individuais. Quatro desses sinais estão ligados directamente aos blocos vizinhos; todos podem ser ligados aos recursos de interligação exteriores

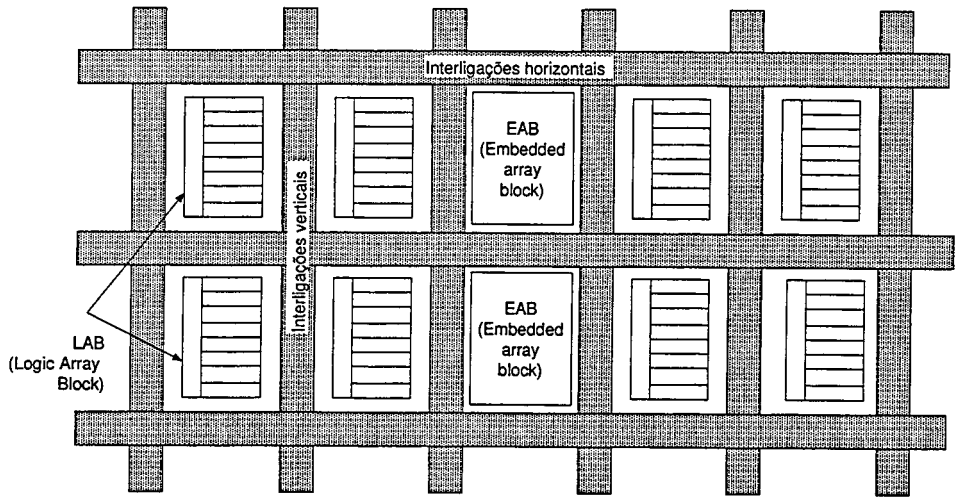


Figura 2.15: Estrutura geral dos circuitos FPGA Flex 10K.

aos blocos.

Altera Flex10K

A estrutura geral dos circuitos FPGA Flex10K da companhia Altera está ilustrada na figura 2.15. Estes circuitos, programáveis por memória estática, contém dois tipos de blocos diferentes. Os blocos designados por EAB (*Embedded Array Block*) são essencialmente blocos flexíveis de memória estática com registos nas entradas e saída, e que podem ser usados para diversos fins:

- como tabela de grandes dimensões para implementar funções combinacionais complexas (por exemplo, um multiplicador de 5×4 bits;
- memória estática de acesso directo;
- memória estática com dois portos de acesso.

Os blocos LAB (*Logic Array Block*) correspondem aos blocos lógicos dos circuitos FPGA mais tradicionais. Conforme se pode ver na figura 2.16, cada bloco é composto por oito elementos lógicos ligados por uma matriz de encaminhamento. Cada elemento lógico tem uma granularidade média e é composto por uma tabela de quatro entradas (também configurável como duas tabelas de três entradas) e um elemento de memória (configurável como *flip-flop* tipo D, T, RS ou JK). Cada elemento inclui ainda circuitos de suporte à composição de vários elementos em funções de muitas entradas:

- Circuitos de transporte (*carry logic*) para somadores, contadores e comparadores;

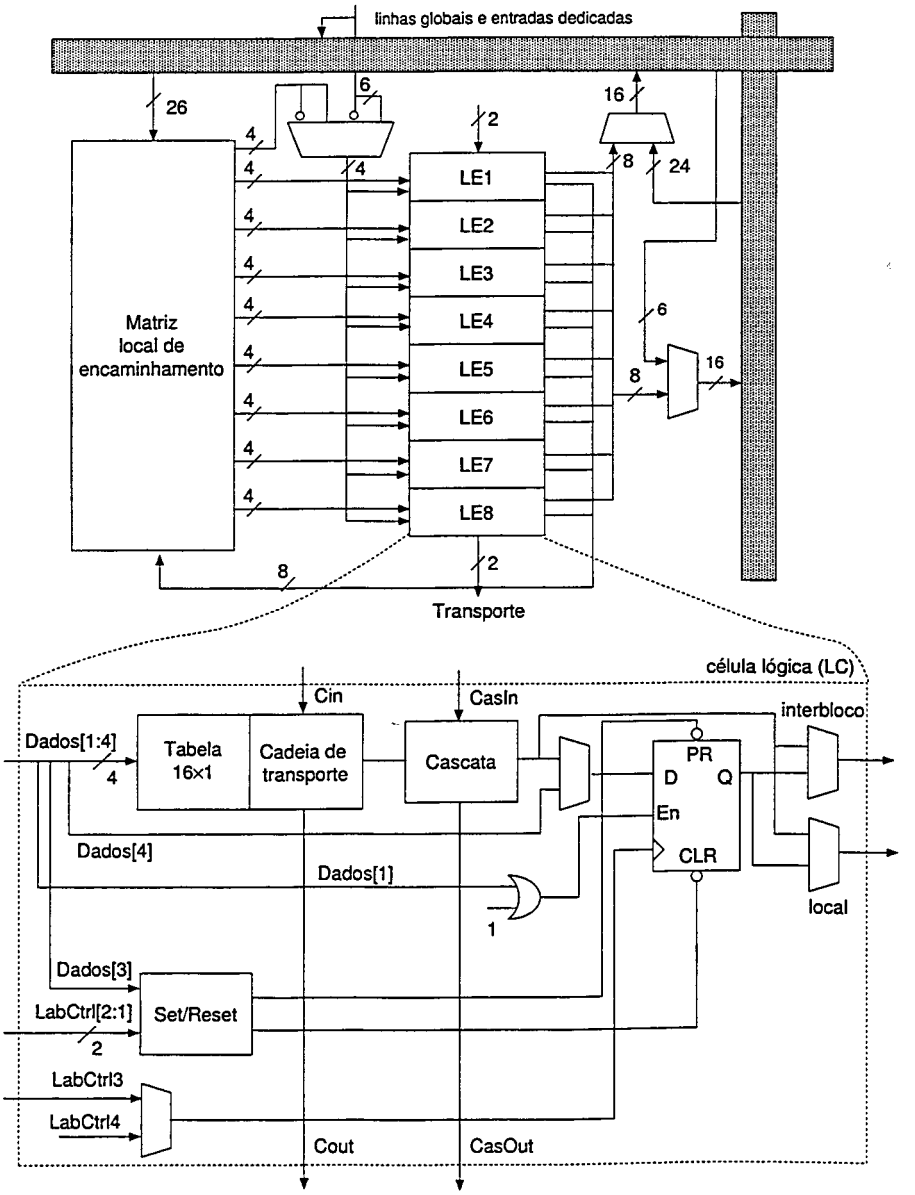


Figura 2.16: Bloco LAB do circuito FPGA Flex10K.

- Ligações em cascata para combinar as saídas de um conjunto de elementos (cascatas de portas AND ou OR).

As duas saídas de cada elemento ligam respectivamente à matriz de encaminhamento local e aos barramentos globais.

A organização desta família de circuitos revela muitas influências dos circuitos CPLD. Cada bloco LAB tem uma estrutura semelhante a um circuito PAL, com cada elemento equivalendo a uma macro-célula. A organização hierárquica dos recursos lógicos e a interligação por matriz de encaminhamento também são típicas dos circuitos CPLD. Estas opções arquitecturais resultam naturalmente da larga experiência da companhia com circuitos CPLD, em cujo mercado detém uma posição dominante.

2.3.2 Granularidade e desempenho

A arquitectura dos blocos lógicos tem forte influência sobre as características dos circuitos com eles implementados, embora o seu efeito seja indissociável de outros factores como a estrutura de rede de interligações, a tecnologia de programação e a qualidade das ferramentas de projecto assistido por computador. Existem contudo alguns estudos empíricos que procuram determinar a granularidade ideal, tanto ao nível do desempenho como da área usada. Os seus resultados permitem, na medida em que podem ser considerados representativos, estabelecer alguns critérios de avaliação de arquitecturas particulares[2]. Todos os estudos pressupõem, implícita ou explicitamente, uma organização do tipo *gate array*, pelo que os seus resultados não se aplicam em geral a matrizes computacionais homogéneas.

A influência da granularidade sobre o desempenho tem dois factores principais, de evolução oposta. A redução da granularidade (i.e., o aumento da<dimensão dos blocos lógicos) permite reduzir o número de blocos necessários para implementar uma dada funcionalidade; contudo o atraso intrínseco de cada bloco também aumenta.

A investigação empírica descrita por Singh *et al.* [1992] e Brown *et al.* [1992] envolve blocos lógicos baseados em memórias estáticas, multiplexadores, portas NAND de várias entradas e circuitos do tipo AND/OR. A investigação teve duas fases: na primeira foi estimado o tempo de atraso de cada tipo de bloco lógico para um processo CMOS 1.2 μm; na segunda fase, dezasseis circuito foram mapeados para os blocos lógicos escolhidos. O caminho crítico foi então calculado para os atrasos calculados na primeira fase e para diferentes valores típicos do tempo de atraso das interligações.

Os resultados indicam que tabelas de cinco/seis entradas ou blocos do tipo ACT (ver pág. 24) levam a implementações com os menores tempos de atraso; pelo contrário, blocos de granularidade fina resultam em tempos de atraso superiores, em alguns casos três vezes mais. As

[2]Naturalmente, os fabricantes também efectuam os seus estudos arquitecturais, possivelmente muito mais extensos que os referidos aqui (por exemplo, envolvendo para além dos aspectos arquitecturais propriamente ditos, outros factores como a facilidade de suporte for ferramentas automáticas, fiabilidade e manufacturabilidade). Os resultados destes estudos raramente são divulgados.

portas AND/OR levam a resultados intermédios. Contudo, no caso particular desta última alternativa, os resultados não são conclusivos porque as ferramentas de apoio ao projecto usadas não estavam adaptadas a esta arquitectura. As melhores arquitecturas representam o compromisso mais favorável entre poucos níveis de interligação e tempos de atraso intrínsecos não excessivos.

Outra questão que mereceu diversos estudos foi a da relação entre granularidade e área utilizada. A diminuição da granularidade implica a diminuição do número de blocos (para uma dada funcionalidade) mas leva a um aumenta da área de interligação por bloco. Este efeito, embora contra-intuitivo, foi verificado empiricamente por [Rose *et al.*, 1990], e também foi observado em circuitos *gate array* e justificado teoricamente por El Gamal [1981]. A explicação baseia-se em dois factos: i) blocos de maior dimensão têm geralmente mais pinos de acesso; b) o comprimento médio das interligações ponto-a-ponto aumenta, embora o seu número total diminuia. A conjugação dos dois factores leva a um aumento da área gasta, por bloco, em interligações.

As investigações empíricas de Rose *et al.* [1990] mostram que a área de interligações é o factor de maior peso na área total (70% a 90%). O resultado final de todos estes factores é que o aumento de área por bloco pode compensar, e mesmo ultrapassar, a diminuição de área provocada pela diminuição do número de blocos. Os resultados empíricos para blocos baseados em tabelas indicam que são as tabelas de quatro entradas aquelas que permitem obter áreas totais menores; este valor é praticamente independente da tecnologia de implementação. Os resultados também indicam que a inclusão de um *flip-flop* tipo D leva quase sempre à redução da área usada em circuitos sequenciais.

Estes resultados são confirmados por outros trabalhos [Singh *et al.*, 1992; Kouloheris e El Gamal, 1991]; em particular, tabelas com quatro entradas e uma saída permitem obter a menor área entre todas as tabelas de K entradas ($1 < K \leq 9$). Este resultado é válido para diferentes tecnologias de programação e distâncias de interligação. Blocos lógicos do tipo PLA com 8-10 entradas, 3–4 saída e 12–13 termos de produto também levam a bons aproveitamentos de área, comparáveis aos melhores obtidos com tabelas de RAM.

Do que atrás fica dito resulta que a utilização de tabelas RAM de quatro ou cinco entradas parece corresponder às situações mais favoráveis de área e desempenho. De facto, trata-se da opção usada em muitas famílias, por exemplo, XC4000, AT40K e Flex10K, de entre as referidas anteriormente.

Em algumas destas arquitecturas as tabelas surgem inseridas em grupos maiores (*cluster*). Em geral, todos os blocos de um grupo partilham um conjunto de linhas de entrada e podem usar directamente as saídas dos outros blocos do grupo como entradas (fig. 2.17), obtendo assim uma certa localidade de processamento que é benéfica para o desempenho global (visto que não é necessário recorrer a recursos de interligação mais lentos) e para o tempo de projecto (já que o agrupamento de blocos permite reduzir o esforço de colocação).

De acordo com estudos empíricos de blocos constituídos por tabelas com quatro entradas,

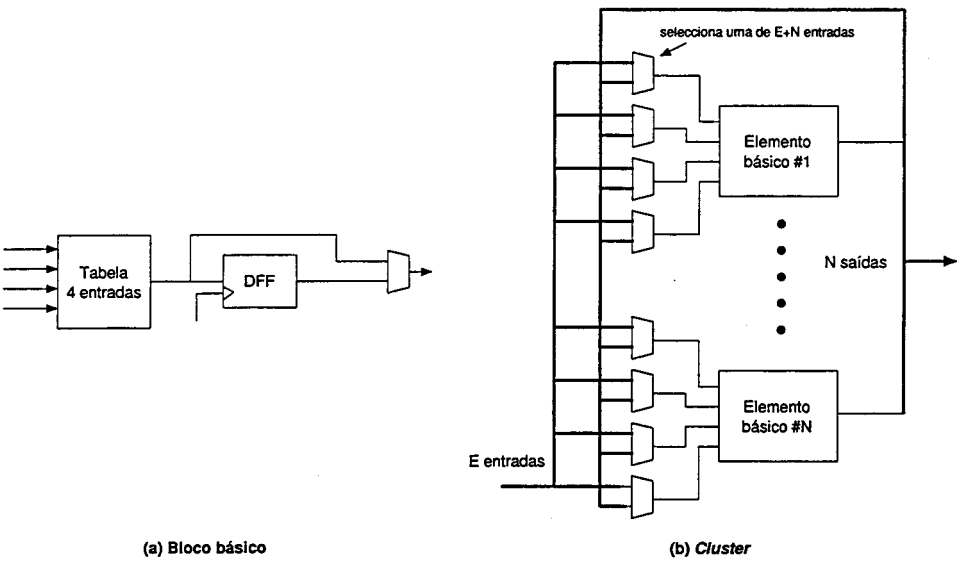


Figura 2.17: Estrutura genérica de um bloco complexo.

a melhor utilização de área é obtida com grupos de $N = 4$ tabelas [Betz e Rose, 1997]. O menor número de entradas para o grupo que permite obter a utilização de todos os blocos é $E = N + 2$; as restantes $4N - (2N + 2) = 2N - 2$ entradas são ligadas, se necessário, a saídas de blocos do mesmo grupo.

Comparando este valores com as arquitecturas comerciais já referidas, nota-se que estas fornecem geralmente um número superior de entradas. Assim, A família XC5200 tem $N = 4$ e $E = 16$, substancialmente mais que o valor $E = 10$ sugerido pelo estudo; também a família Flex 10K tem $N = 8$, $E = 30$ (e mesmo $E = 34$ em alguns modelos), valores substancialmente superiores aos sugeridos. De notar que os pressupostos iniciais destes estudos não abrangem muitas outras arquitecturas comerciais, por exemplo, as famílias XC4000 e AT40K.

2.4 Comunicação entre blocos lógicos

Nos circuitos FPGA as interligações entre blocos são fixas em número, posição e comprimento; apenas o estabelecimento de contacto eléctrico entre os diversos segmentos pode ser configurada. Uma versão genérica da estrutura de interligação está representada na figura 2.18. Uma rede bidimensional de canais é percorrida por pistas metálicas; a ligação entre canais ortogonais é estabelecida em *blocos de encaminhamento* que permitem ligar uma entrada a um conjunto das saídas; o estabelecimento de contacto entre as entradas/saídas de um

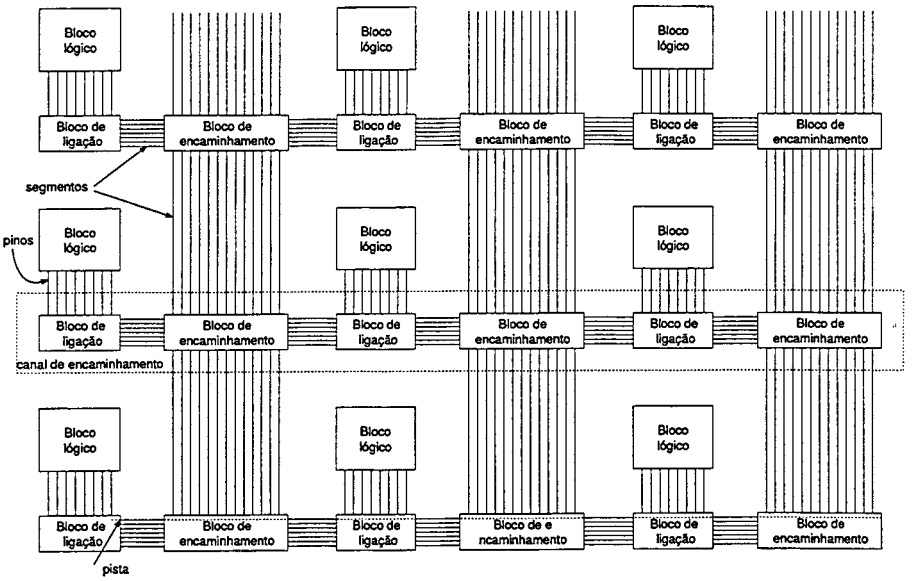


Figura 2.18: Estrutura geral da rede de interligações de um circuito FPGA.

bloco lógico e as pistas é efectuado por *blocos de ligação*.

A maioria das interligações necessárias para implementar um circuito têm carácter local, restringindo-se à vizinhança do bloco que gera o sinal. Apenas uma percentagem reduzida de ligações tem âmbito global. Consequentemente, a utilização de pistas que percorrem o circuito de lado a lado não é a mais vantajosa, tanto em termos de área como de desempenho. A área é mal aproveitada porque mesmo uma ligação curta ocupa a área correspondente a uma linha completa (fig. 2.19b); neste caso, a área ocupada cresce com o quadrado do número de pistas. O desempenho vem prejudicado pelas capacidades parasitas associadas a pistas compridas e aos interruptores associados. A solução consiste em utilizar pistas divididas em segmentos de tamanhos diversos (fig. 2.19c). Pode ser obtida maior flexibilidade se os segmentos puderem ser ligados por interruptores de maneira a obterem-se segmentos de tamanho apropriado a cada situação (fig. 2.19d).

Por outro lado, a utilização de segmentos tão curtos que permitam obter a flexibilidade dos circuitos *gate array* também não é viável, porque requer um elevado número de interruptores no percurso do sinal (fig. 2.19e); como o tempo de atraso é proporcional ao quadrado do número de interruptores nesse percurso, o atraso para ligações mais compridas torna-se rapidamente inaceitável.

O problema do encaminhamento em canais segmentados é de dificuldade semelhante ao do encaminhamento para circuitos *gate array*, mas muitos casos particulares podem ser re-

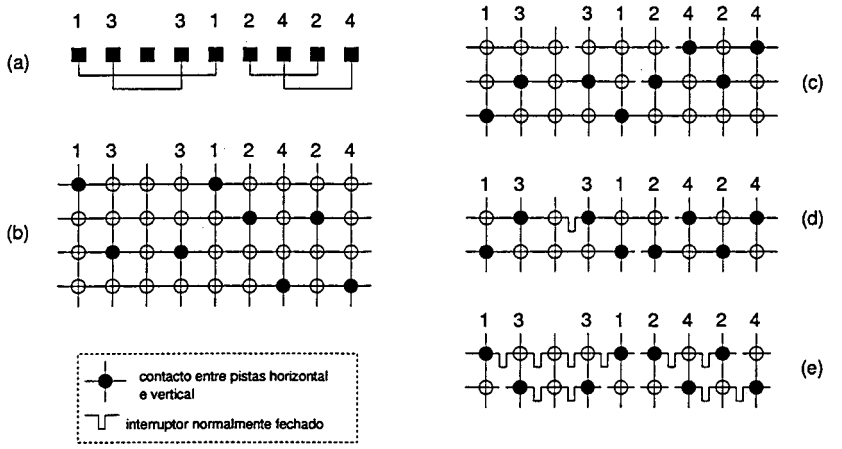


Figura 2.19: Influência de esquemas de segmentação no encaminhamento de sinais.

solvidos mais eficientemente, existindo heurísticas que permitem resolver algumas versões do problema em poucos minutos [El Gamal *et al.*, 1991].

2.4.1 Organização da rede de de interligações

O problema que se põe ao arquitecto de circuitos FPGA é o de escolher o número de pistas bem como a distribuição dos comprimentos dos respectivos segmentos. No caso dos circuitos *gate array* existem modelos estatísticos que permitem determinar a largura de canal mais apropriada [El Gamal, 1981]; tais modelos não existem para circuitos FPGA. Contudo, a experiência tem mostrado que mesmo uma distribuição de segmentos *ad hoc* pode ter bom desempenho. Alguns estudos indicam que o número de pistas não deve ser muito superior ao esperado em circuitos *gate array* (apenas duas ou três pistas mais) [Rose e Brown, 1991].

Apresentam-se a seguir, a título ilustrativo, alguns exemplos concretos de estruturas de interligação usadas em circuitos FPGA comerciais.

Xilinx XC4000

A organização da estrutura de interligações da família XC4000 segue de bastante perto o esquema esboçado na figura 2.18. Tomando como exemplo a série XV, cada canal vertical contém 34 linhas e cada canal horizontal 30. Os canais verticais comportam ainda oito linhas de redes globais de distribuição de sinais de relógio. Cada bloco lógico tem ainda duas linhas de ligação directa aos blocos vizinhos abaixo e à esquerda. O bloco de ligação permite ligar saídas e entradas a quase todas as linhas. O bloco de comutação permite ligar cada sinal de entrada às saídas nos outros três lados do bloco. As pistas simples são compostas

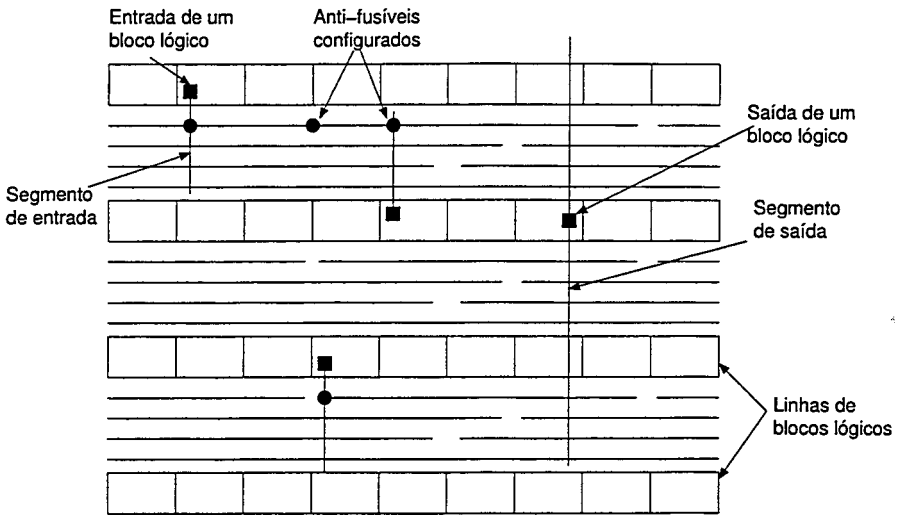


Figura 2.20: Interligações na família ACT-3.

por segmentos que ligam blocos de encaminhamento contíguos; as pistas duplas são compostas por segmentos que estabelecem ligações de dois em dois blocos de encaminhamento. De forma análoga, pistas quadruplas são constituídas por segmentos que ligam blocos de encaminhamento colocados de quatro em quatro blocos lógicos; estes blocos de encaminhamento também servem para restaurar a qualidade eléctrica do sinal. A área ocupada por todos estes recursos de interligação é várias vezes superior à área do bloco lógico.

A família XC5200, para além das ligações intra-bloco referidas anteriormente (pág.30), contém uma estrutura de interligações genericamente muito similar à da família XC4000, embora com um número inferior de recursos: por exemplo, não tem segmentos quadruplos tem 20 segmentos horizontais simples, 4 duplos e oito longos, e apenas duas linhas globais distribuídas.

Actel ACT

Os circuitos da família ACT estão organizados por linhas, ao estilo dos circuitos *gate array*. A figura 2.20 mostra a organização geral, incluindo as ligações verticais. O bloco de ligação é, nesta arquitectura, constituído pelos canais horizontais. Cada pino de entrada pode ser ligado a todas as pistas do canal que está situado ao lado do bloco. A saída estende-se verticalmente por dois canais acima e dois canais abaixo da posição do bloco, podendo ligar-se a qualquer pista neles situada. Existem ainda cinco pistas verticais por bloco que percorrem toda a altura do circuito.

Como se pode ver, esta arquitectura não tem um bloco de encaminhamento geometrica-

mente bem definido. O encaminhamento é feito de maneira distribuída ao longo dos canais horizontais. Todas as pistas verticais podem ligar-se a todas as pistas horizontais incidentes. Este tipo de conectividade é mais flexível que o proporcionado pelo bloco de encaminhamento XC4000. O tamanho do elemento de configuração, o anti-fusível, é a principal razão desta abundância de recursos.

Cada canal horizontal tem 22 pistas divididas em segmentos de diferentes comprimentos, desde aqueles que apenas permitem a ligação entre dois blocos adjacentes a segmentos que abarcam toda a linha. Esta distribuição variada torna provável que se encontre o segmento de comprimento desejado para cada ligação. Como resultado, cada ligação intra-canal requer poucos interruptores (apenas 2–3% dos antifusíveis de um circuito necessitam de ser configurados).

Os circuitos FPGA das companhias Crosspoint (CP20K) e Quicklogic (pASIC 3) utilizam organizações que obedecem a princípios gerais semelhantes.

Xilinx XC6200

Este circuito, tal como grande parte das matrizes computacionais homogéneas, assenta a sua estrutura de interconexões em ligações locais de organização facilmente replicável. Neste caso é estabelecida uma hierarquia de interligações que está subjacente à organização dos blocos lógicos em agrupamentos hierárquicos sucessivamente maiores, de 4×4 e 16×16 blocos (fig. 2.21). Agrupamentos de 4×4 comunicam com os agrupamentos vizinhos através de pistas de dimensão equivalente à largura de quatro blocos lógicos; analogamente, os agrupamentos de 16×16 também estão ligados por pistas de comprimento adequado. Existe ainda um conjunto de pistas que percorrem toda a área do circuito. Todo este arranjo é simétrico na vertical e na horizontal. Esta organização hierárquica tenta aproveitar o processamento localizado existente na maior parte dos sistemas digitais.

Cada bloco lógico é atravessado por quatro pistas de dimensão 4, quatro pistas de dimensão 16 e quatro pistas longas. Todo o encaminhamento é efectuado por multiplexadores colocados na periferia dos agrupamentos correspondentes: blocos individuais (para ligações entre vizinhos—ver figura 2.9), grupos 4×4, 16×16 e circuito completo (para as pistas longas). Só nesses locais podem ser introduzidos sinais eléctricos nas pistas; os respectivos valores podem ser usados em qualquer bloco situado sob as pistas correspondentes. Todo este arranjo confere ao circuito uma estrutura completamente regular e simétrica, tornando a implementação de macro-blocos (sub-circuitos) largamente independente da sua localização na matriz.

Todos os multiplexadores servem simultaneamente de blocos de ligação e de encaminhamento. A função de ligação é muito simples, porque as ligações às entradas dos multiplexadores são fixas; a função predominante destes é, naturalmente, a de encaminhamento.

Atmel AT6000 e AT40K

As duas famílias de circuitos FPGA da companhia Atmel usam os mesmos princípios gerais nas suas redes de interconexões, embora o número de recursos seja muito superior nos ele-

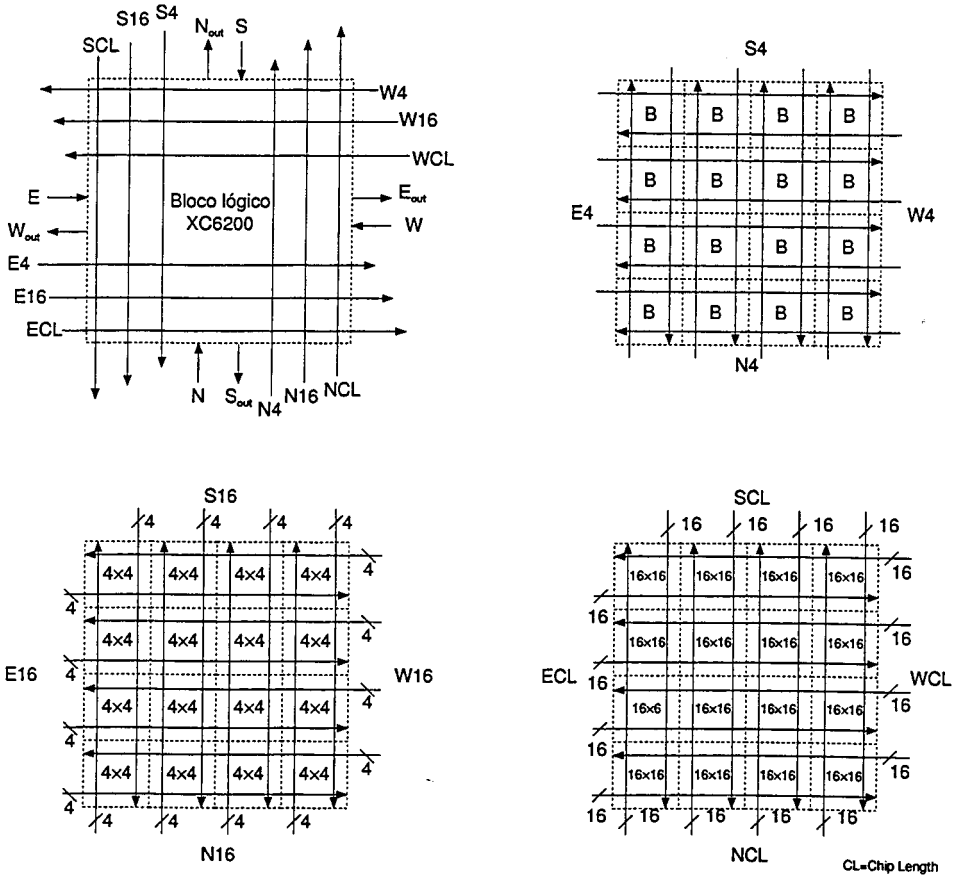


Figura 2.21: Estrutura de interligações da família XC6200.

mentos da família AT40K. A figura 2.22 ilustra a distribuição de recursos para os elementos da família AT6000. Trata-se de um arranjo regular e simétrico. Para além das ligações aos vizinhos mais próximos, cada célula tem ligações a duas pistas horizontais e duas pistas verticais. A ligação é feita por portas de passagem existentes nos blocos (fig. 2.8), o que permite a implementação de barramentos de alta impedância. Existem ainda mais quatro linhas paralelas às anteriores (linhas "expresso"); não estando directamente ligadas aos blocos lógicos, estas linhas apresentam melhores características eléctricas. De oito em oito blocos existem repetidores programáveis que podem

- isolar os segmentos;
- ligar dois segmentos locais;

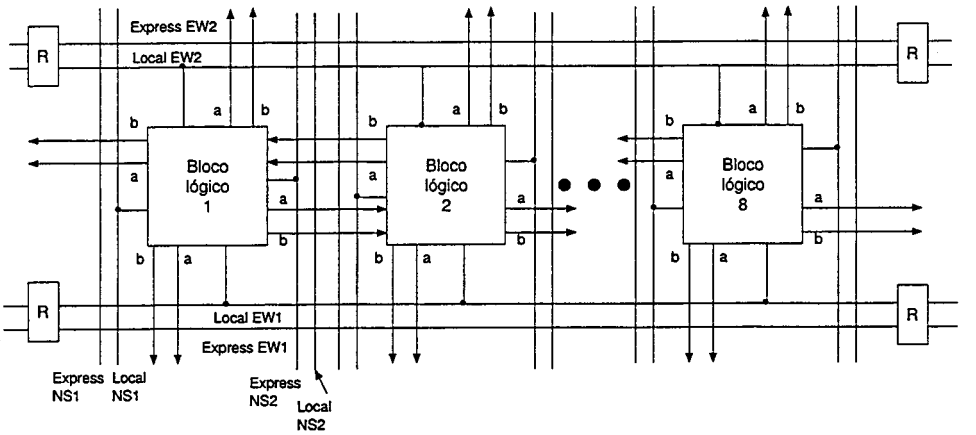


Figura 2.22: Recursos de interligação da família AT6000.

- ligar dois segmentos expresso;
- ligar um segmento local a um segmento expresso;
- restaurar os níveis eléctricos dos sinais.

Algumas funções de encaminhamento, por exemplo, a passagem de um segmento vertical para um horizontal, são implementadas pelos circuitos do bloco lógico (as portas de transmissão X e Y indicadas na figura 2.8).

Os elementos da família AT40K têm os seus recursos de interligação distribuídos por cinco planos, todos com um arranjo similar (fig. 2.23). Cada plano tem um segmento local e dois segmentos expresso em cada direcção. Cada repetidor liga dois segmentos locais e dois segmentos expresso. Cada segmento local abrange quatro blocos lógicos; segmentos expresso abrangem oito. Os repetidores regeneram os níveis eléctricos dos sinais e podem interligar quaisquer segmentos a que estejam ligados. Portas de passagem permitem interligar segmentos locais sem usar os repetidores e criar barramentos de alta impedância. A interligação de segmentos locais é feita nos blocos lógicos (usando as portas de passagem V1–V5 e H1–H5 da figura 2.11). Ligações entre segmentos expresso são implementadas por portas de passagem distribuídas ao longo da matriz.

Altera Flex10K

A estrutura de interligações da família Flex10K tem dois níveis hierárquicos. O primeiro é composto pela matriz de encaminhamento existente em cada bloco (fig. 2.16). O segundo nível é constituído por um conjunto de pistas contínuas (não segmentadas) horizontais e verticais (fig. 2.15). O desempenho proporcionado por este tipo de arquitecturas é mais pre-

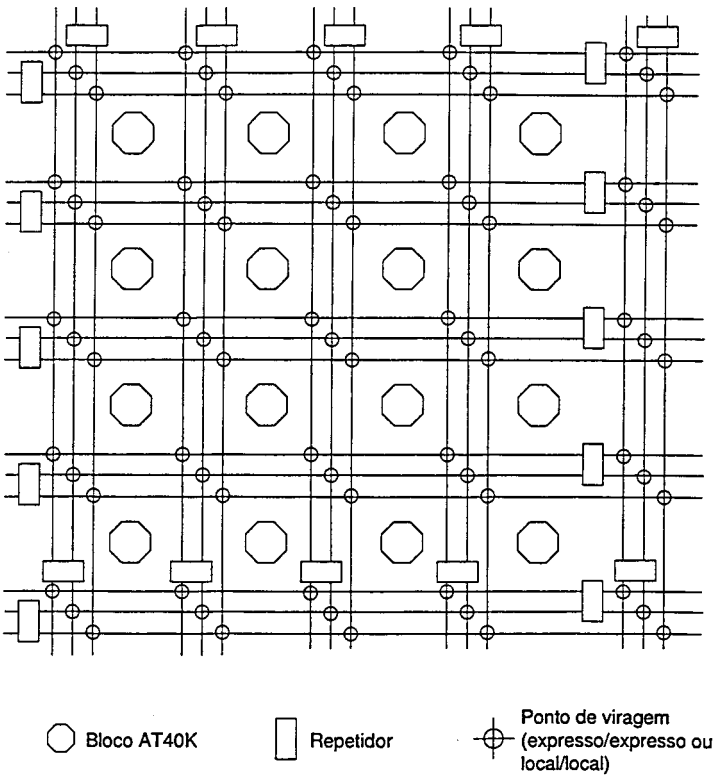


Figura 2.23: Recursos de interligação da família AT40K.

visível que o obtido com arquitecturas segmentadas. As pistas horizontais são divididas em dois segmentos iguais, para se obter maior flexibilidade.

No maior circuito da família, EPF10K250E, existem vinte canais horizontais, com 456 pistas por canal, e setenta e seis canais verticais, com 40 pistas por canal. Os blocos de encaminhamento são implementados por multiplexadores com saída de alta impedância. Embora não apresentadas na figura, existem ainda ligações directas entre blocos vizinhos. A ligação de entradas faz-se através de linhas dedicadas da matriz local de encaminhamento de cada bloco; as saídas também são ligadas por multiplexadores às pistas do barramento de comunicação.

A organização dos canais é muito regular e permite o seu empacotamento denso e eficiente, já que não existem interruptores nos interiores dos canais. Adicionalmente, o tempo de atraso de cada pista é independente da configuração, o que é muito útil quando se pretende estimar o desempenho do circuito; também torna o desempenho relativamente independente de modificações pontuais do projecto. Como desvantagem, é de assinalar que arquitecturas segmentadas podem obter melhor desempenho, porque o comprimento das suas interligações

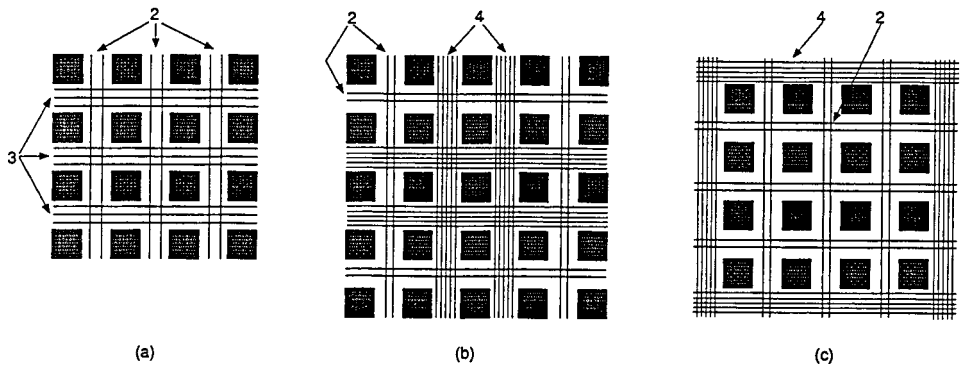


Figura 2.24: Distribuição não-uniforme de pistas de interligação.

se adapta às necessidades.

2.4.2 Aproveitamento de recursos e desempenho

À semelhança do que acontece com a granularidade dos blocos lógicos, também os efeitos da organização da estrutura de interligações sobre o aproveitamento de área e a facilidade de efectuar o encaminhamento foram estudados na literatura. O compromisso básico estabelece-se entre a disponibilização de um número elevado de interruptores configuráveis (que aumentam a flexibilidade da estrutura) e a área por eles ocupada. A maior parte dos estudos efectuados utiliza o modelo genérico referido anteriormente e ilustrado na figura 2.18. Neste contexto, os dois parâmetros principais são F_c, o número de pistas a que um pino de um bloco lógico pode ligar directamente, e F_s, o número de saídas a que cada entrada de um bloco de encaminhamento pode ser ligada. Neste modelo considera-se que todos os pinos ou pistas têm igual grau de conectividade.

Os estudos de Rose e Brown [1991] e Brown *et al.* [1992] procuram identificar a influência dos parâmetros F_c e F_s sobre a facilidade de obter um encaminhamento viável (*routability*). As conclusões indicam que a conectividade do bloco de ligação deve ser elevada (F_c superior a 50% do número de pistas por canal), com valores recomendados na casa dos 70–80%. Pelo contrário, a flexibilidade do bloco de encaminhamento pode ser relativamente reduzida, sendo $F_s = 3$ o valor mínimo recomendado. Naturalmente que a qualidade do encaminhamento (medida pelo número de pistas por canal necessárias para a sua realização) melhora com o aumento de F_s e F_c; contudo, os resultados experimentais indicam que os ganhos geralmente não justificam o aumente da área utilizada pelos interruptores programáveis (ou seja, esta área começa a ser superior à área poupada pela diminuição do número de pistas por canal).

A distribuição do número de pistas por canal foi objecto de estudos empíricos descritos

em [Betz e Rose, 1996, 1998]. O objectivo é determinar se existe alguma vantagem em ter canais com número diferente de pistas. Algumas famílias têm canais centrais de maior capacidade (por exemplo, a família ORCA 2, da companhia Lucent Technologies); outras aumentam a capacidade dos canais situados na periferia do circuito por forma a facilitar o encaminhamento quando a associação de sinais internos a pinos externos é fixa (i.e., não pode ser escolhida livremente pelas ferramentas de colocação e encaminhamento). Um primeiro cenário considera a utilidade de arquitecturas com distribuição não-uniforme de pistas, em que canais numa direcção (por exemplo, horizontal) têm maior capacidade que os canais na outra direcção (fig. 2.24). A situação uniforme encontra-se na família XC4000; o segundo caso na família ACT-3.

As conclusões do estudo de Betz e Rose [1996] indicam que a arquitectura uniforme é superior a todas as outras quando os pinos dos blocos lógicos estão uniformemente distribuídos ao longo da sua periferia (como acontece com a família XC4000). Quando os pinos ocupam apenas dois lados da periferia, então a situação mais favorável consiste em ter canais de maior capacidade contíguos aos pinos e canais de menor capacidade na direcção ortogonal (o que corresponde, grosso modo, às opções da família ACT-3).

Um segundo cenário considera organizações não uniformes, em que alguns canais têm maior capacidade que outros. A experiência com circuitos *sea-of-gates* e *standard cell* indica que uma distribuição não uniforme poderia ser vantajosa, já que estes surgem na maior parte dos casos. Esta situação reforça a noção intuitiva que o centro do circuito é um lugar de maior congestão de comunicações. Os resultados de Betz e Rose [1996] indicam que, de facto, as arquitecturas uniformes têm quase sempre vantagem em termos de área usada; não só os pontos de congestão se encontram espalhados por todo o circuito, como uma arquitectura uniforme parece ser mais facilmente aproveitada pelas ferramentas de encaminhamento. Por outro lado, a analogia com circuitos *sea-of-gates* revela-se desajustada, porque nestes os canais mais largos não são colocados em posições fixas, mas dependem da implementação; nos circuitos FPGA, a necessidade de definir esses canais à partida faz com que a sua colocação raramente seja a mais apropriada para obter o melhor aproveitamento.

A única variação da uniformidade geral considerada útil por este estudo, consiste em aumentar cerca de 25% a capacidade dos canais adjacentes à periferia do circuito para compensar as restrições resultantes de fixar previamente a funcionalidade dos pinos exteriores. Estas situações, frequentes quando são feitas modificações a um circuito já implementado ou quando os circuitos FPGA se encontram inseridos em sistemas reconfiguráveis, são objecto de análise empírica por Khalid e Rose [1995], tendo-se constatado que levam a um aumento sensível da utilização de recursos de encaminhamento (aumentos entre 15% e 100%) dependendo da arquitectura. São precisamente os circuitos com canais periféricos alargados que registam os menores aumentos.

Como na maioria dos estudos empíricos, é necessário ter em conta que os resultados podem ser representativos das situações analisadas, mas a sua generalização deve ser encarada com

cautela. Apesar deste defeito, estudos empíricos deste tipo parecem ser a única forma de obter algumas orientações para estabelecimento dos factores que influenciam o desempenho e aproveitamento de área das diversas arquitecturas existentes ou a serem criadas.

2.5 Sistemas digitais reconfiguráveis

As possibilidades abertas pela reconfiguração rápida dos circuitos FPGA foram cedo identificadas e aproveitadas, com estes componentes a serviram de base para sistemas digitais que, combinando de forma única adaptabilidade e velocidade, permitem obter um elevado desempenho em tarefas específicas. Os sistemas digitais reconfiguráveis (SDR) vieram situar-se entre os computadores tradicionais (sistemas de uso geral capazes de executar qualquer tipo de computação, mas sujeitos essencialmente a um modelo sequencial de interpretação e execução de instruções) e os sistemas digitais de aplicação específica (especializados para uma dada tarefa ou aplicação, mas de utilização nula fora desse âmbito restrito). De facto, os SDR podem ser configurados para resolver todos os problemas que se ajustem à sua arquitectura geral, explorando o paralelismo maciço característico das implementações directas em *hardware*. A sua adaptação pode ser mesmo levada até ao nível da instância individual de um problema; neste caso o SDR assume (temporariamente) um comportamento totalmente específico, determinado não apenas pelas características do problema, mas também pelos dados específicos da instância particular em consideração.

Para além da utilidade evidente dos SDR na prototipagem rápida e emulação de sistemas digitais, bem como no ensino de sistemas digitais e arquitecturas de computadores, também a sua utilização como sistemas computacionais dedicados (*custom computing machines*—CCM) apresenta grande interesse. Uma primeira fase, de finais da década de 80 até meados da década de 90, caracterizou-se por um ênfase em sistemas computacionais de grandes dimensões, geralmente compostos por dezenas de circuitos FPGA. Um conjunto de projectos de grande dimensão (entre outros, Splash [Buell *et al.*, 1996], PAM [Vuillemin *et al.*, 1996] e Teramac [Amerson *et al.*, 1995]) confirmaram a viabilidade, utilidade e desempenho desta categoria de sistemas. Mais recentemente outras áreas passarem a ser objecto de investigação mais intensa, por exemplo as abordagens híbridas microprocessador/FPGA e as técnicas de reconfiguração dinâmica (cf. cap. 3). Áreas mais exploratórias, como *hardware* evolutivo, têm igualmente registado actividade crescente.

2.5.1 Um novo paradigma e as suas aplicações

Como já foi referido, os circuitos FPGA surgem como um meio de aproveitar as crescentes capacidades do fabrico de semicondutores para reduzir o tempo de desenvolvimento de sistemas digitais. O seu mercado natural é, pois, constituído por aplicações em que podem substituir proveitosamente conjuntos de circuitos lógicos SSI, MSI ou PAL, ou por aplicações

em que substituem os CIAE, sem os respectivos custos iniciais elevados e com tempos de produção muito curtos. Com o decorrer do tempo tornou-se claro que a volatilidade dos circuitos FPGA de memória estática, longe de constituir uma desvantagem, abria caminho para novas áreas de aplicação. O processo de configuração rápido e electricamente reversível, semelhante à "configuração" de um microprocessador por programas diferentes, colocou este tipo de circuitos FPGA no âmago de muitos sistemas reconfiguráveis.

2.5.1.1 Emulação e prototipagem rápida

Emulação e prototipagem rápida de sistemas digitais são duas faces de uma mesma actividade. Enquanto a prototipagem rápida coloca o acento na exploração de diferentes opções de implementação, um pouco como uma placa de montagem virtual onde o projectista pode experimentar as suas ideias, a emulação concentra-se na verificação da funcionalidade de um dado projecto, permitindo avaliá-la mais rapidamente que através de simulação. Em qualquer dos casos, os circuitos FPGA de memória estática são aproveitados como meio de assegurar a rápida adaptação de uma infraestrutura genérica.

A prototipagem rápida tem elevado interesse comercial porque permite obter uma versão operacional do sistema desejado, rapidamente e com baixos custos. Para além de permitir avaliar diferentes abordagens, a existência de um protótipo impulsiona o desenvolvimento de todos os elementos de *hardware* ou *software* que dele dependem, reduzindo assim o tempo total de desenvolvimento do produto. Estes sistemas têm ainda vantagens pedagógicas importantes, conforme mostra o elevado número de sistemas deste tipo desenvolvidos para ambientes académicos [Alves, 1997; van den Bout *et al.*, 1992].

A emulação de sistemas digitais complexos, em especial circuitos integrados de aplicação específica, é uma vertente de grande sucesso desta tecnologia. O projectista de um CIAE precisa de se assegurar que os circuitos projectados implementam correctamente a funcionalidade desejada; esta necessidade é tanto mais premente quanto um erro de projecto ou implementação comporta custos muito elevados, tanto em dinheiro como em tempo perdido. A simulação é o meio mais tradicional para proceder à verificação da funcionalidade pretendida. Trata-se de uma solução flexível, mas demorada. Dois factores contribuem para esta situação: (i) os circuitos a simular podem conter um elevado número de elementos individuais (na ordem dos milhões); (ii) circuitos complexos têm comportamentos complexos e necessitam, portanto, de sequências de estímulos longas. Para além disso, a simulação da interacção com o "mundo real" é sempre problemática, muitas vezes mesmo impossível de efectuar com rigor. A distribuição do trabalho de simulação por vários computadores e a utilização de aceleradores de *hardware* são duas das medidas que permitem atenuar os problemas causados pelo crescimento destes dois factores.

Uma solução alternativa consiste em mapear o circuito num sistema multi-FPGA por forma a que este realize o mesmo comportamento funcional. A emulação permite ter a funcionar

uma versão do sistema pretendido várias ordens de grandeza mais depressa que uma simulação [Butts *et al.*, 1992a; Dahl *et al.*, 1994; Ganapathy *et al.*, 1996]. O sistema MARS da empresa Quickturn foi usado para emular o microprocessador MC68060 à frequência de 1 MHz [Kumar *et al.*, 1995]. A versão emulada necessita de 165 minutos para fazer a inicialização do sistema UNIX; o tempo estimado que um simulador de alto nível levaria para completar a mesma tarefa é de 13 anos; uma versão do mesmo processador operando a 50 MHz necessita de 100 segundos.

O processo de mapeamento, embora automático, demora tipicamente algumas horas (dependendo naturalmente da dimensão e complexidade do circuito a emular); consequentemente a emulação não é uma opção vantajosa para circuitos pequenos. Por outro lado, um sistema de emulação para CIAE complexos necessita de um número elevado de circuitos FPGA e de numerosos circuitos de apoio, o que resulta em produtos comerciais de preço muito elevado.

A possibilidade de dispor de um sistema capaz de operar em tempo real (ou quase) numa fase precoce do projecto permite o desenvolvimento concorrente de outros elementos, como placas de circuito impresso e *software* de controlo. A consequente redução do tempo total de desenvolvimento reveste-se de grande importância comercial.

2.5.1.2 Implementação de algoritmos em *hardware*

Uma outra linha de aproveitamento das potencialidades dos circuitos FPGA de memória estática consiste na sua utilização como substrato computacional de uso genérico. Esta abordagem leva aos sistemas computacionais dedicados (SCD), em que (partes de) algoritmos são realizados directamente em *hardware*, neste caso um ou mais circuitos FPGA. Embora estes sistemas não possuam a versatilidade dos microprocessadores, podem atingir níveis de desempenho muito elevados em problemas com estrutura computacional apropriada, i.e., cujas operações (e respectivo sequenciamento) possam ser mapeadas eficientemente nos recursos disponibilizados pelos circuitos FPGA.

Os SCD com circuitos FPGA constituem uma alternativa recente no conflito permanente entre generalidade e eficácia que resulta de uma observação empírica básica: quanto mais genérico é um sistema e maior é o número das tarefas que permite efectuar, menor é a sua eficiência em cada uma das tarefas em particular.

Desde a década de 60 que existem esforços no sentido de criar computadores cuja organização possa variar de acordo com a estrutura específica das computações a executar [Estrin *et al.*, 1963; Reddi e Feustel, 1978]. Neste contexto, o termo "reconfiguração" refere-se à reorganização e reagrupamento das unidades funcionais e periféricas, bem como ao estabelecimento de ligações entre elas [Vick *et al.*, 1980]. Uma evolução similar nos sistemas ocorre com processadores micro-programados. Para além do objectivo original de estruturar o sistema de controlo do processador e de permitir emular diferentes conjuntos de instruções, as unidades micro-programadas com memória de leitura/escrita foram aproveitadas para adaptar o processador

a problemas individuais, geralmente através da migração vertical da funcionalidade dos programas de alto nível para microprogramas dedicados [Abd-Alla e Karlgaard, 1974; Rauscher e Agrawala, 1978; Papachristou e Immaneni, 1993].

O conceito de reconfiguração surge igualmente na área dos sistemas tolerantes a falhas. Também aí se trata de reestruturar a interligação das unidades do sistema, desta feita com o objectivo de isolar um componente ou ligação com mau funcionamento, preservando a operacionalidade do sistema global [Sami e Stefanelli, 1986].

Ao longo desta dissertação, o termo **reconfiguração** é sempre associado a circuitos FPGA ou sistemas que os contenham. De igual modo, todas as referências a sistemas computacionais dedicados têm por objecto sistemas baseados em circuitos FPGA.

Os SCD baseados em circuitos FPGA apresentam características bastante diferentes do microprocessador convencional. Este baseia-se na arquitectura tradicional do tipo von Neumann, com uma unidade sequencial de processamento associada a uma memória de leitura/escrita para armazenamento de instruções e dados. Por sua vez, os SCD com circuitos FPGA exploram o paralelismo inerente à existência simultânea de numerosos recursos de processamento simples mapeando as instruções em configurações de *hardware*. Enquanto a operação de um microprocessador varia de instrução para instrução, a configuração do SCD mantém-se constante[3]. Existe assim uma dualidade entre tempo e espaço, que se pode resumir dizendo que microprocessadores variam as suas operações no tempo (execução de instruções sucessivas) enquanto os SCD com circuitos FPGA variam as suas operações no espaço, realizando em cada zona diferentes etapas do processamento [Hauck, 1998b].

Destas observações pode concluir-se que os SCD com circuitos FPGA estão em vantagem nas aplicações com inerente processamento paralelo, especialmente quando as operações são simples e uniformes; os microprocessadores estarão em vantagem sempre que as aplicações exibam um fluxo de controlo complexo e computações irregulares [DeHon, 1996a]. A separação nítida entre estes esquemas de computação dilui-se quando se consideram sistemas de reconfiguração dinâmica e sistemas híbridos FPGA/processador (cf. cap. 3).

Para além da especialização de um SCD para tratamento de um dado problema, é natural ir mais longe e procurar obter a especialização para uma *instância particular*, atingindo-se assim um nível de especialização que não se poderia aproveitar economicamente de outro modo. A implementação de cifra RSA [Rivest *et al.*, 1978] no sistema PAM [Vuillemin *et al.*, 1996] é um bom exemplo desta abordagem: para cada chave de codificação escolhida pelo utilizador, o sistema de apoio gera automaticamente um circuito de cifragem (ou decifragem, conforme a função desejada) cuja estrutura depende dessa mesma chave, i.e., a chave fica embebida no circuito de processamento. Os circuitos gerados desta maneira conseguem um desempenho superior ao de um conjunto de CIAE da companhia AT&T (cf. tabela 2.3).

[3]Em sistemas reconfiguráveis dinamicamente a configuração varia ao longo da execução da aplicação; mas mesmo neste caso, a taxa a que são executadas as operações de reconfiguração é muito inferior à taxa de execução de instruções.

Algoritmo/Aplicação	Sistema	Alternativa	Ganho	Referência
Comparação de DNA	Splash-2	Sparc 10	4300	Hoang [1993]
Criptografia RSA	DECPeRLe-1	CIAE	16	Vuillemin *et al.* [1996]
Modelo de Markov	1 FPGA	Sparc 10	24	Schmit e Thomas [1995]
Detecção de arestas	Splash 2	Sparc 20	271	Ratha *et al.* [1995]
Descodificador Viterbi	RACER	Sparc 5	820	Yeh *et al.* [1996]

Tabela 2.3: Desempenho de alguns sistemas computacionais dedicados.

É sempre possível implementar uma solução baseada em CIAE capaz de obter melhor desempenho que um sistema reconfigurável genérico; contudo, nem sempre se tratará de uma solução economicamente viável. Um sistema reconfigurável pode ser a melhor abordagem em várias situações. Avaliar e quantificar estas vantagens constitui precisamente um dos principais objectivos de vários projectos de grande envergadura. De um modo geral, os SCD com circuitos FPGA justificaram as suas expectativas, conforme se pode depreender da tabela 2.3, que faz um apanhado de alguns resultados notáveis, comparando-os com os melhores resultados obtidos por outros meios. A superioridade dos SCD em algumas áreas é inquestionável, embora seja preciso ter sempre em conta que se trata de computações com estrutura particularmente adaptada a este tipo de sistemas. Quando tal não acontece, os resultados podem ser muito maus [Buell *et al.*, 1996], pelo que as melhorias de desempenho indicadas na tabela não são generalizáveis, mas apenas indicadores das potencialidades da abordagem. Uma situação análoga acontece com os supercomputadores científicos, que podem atingir desempenhos extremamente elevados em algoritmos bem adaptados às suas características estruturais, mas que ficam substancialmente abaixo do desempenho de pico quando tais condições se não verificam.

2.5.1.3 Flexibilidade de reconfiguração

A exploração das possibilidades dos sistemas reconfiguráveis pode ser levada ainda mais longe. Em vez de nos limitarmos ao estabelecimento de configurações numa etapa anterior à execução das aplicações propriamente ditas (reconfiguração inter-aplicação) é possível entrelaçar reconfiguração e execução; o substrato digital é considerado como um recurso a ser multiplexado no tempo de maneira a implementar *hardware* virtual. Esta abordagem baseia-se na observação que muitas computações são disjuntas no tempo, o que leva a que grande parte dos circuitos de um sistema não estejam activos simultaneamente [Eldredge e Hutchings, 1996]. Logo, pode ser possível intercalar operações de reconfiguração no fluxo de computação, de forma a que, em cada instante, o sistema implemente a parte computacionalmente activa do sistema total, que assim apenas existe virtualmente. Estas abordagens são designadas por "reconfiguração dinâmica" ou "reconfiguração em tempo de execução" (*run-time reconfiguration*).

O sistema descrito por Villasenor *et al.* [1995] é um exemplo elementar deste tipo de abordagens. Um processo de codificação em três etapas de imagens vídeo para transmissão pode ser implementado usando um circuito FPGA por etapa. Como essas etapas devem ser executadas sequencialmente, é possível reutilizar um mesmo circuito FPGA para todas as operações, efectuando a substituição sincronizada das várias configurações. A reconfiguração dinâmica permite assim reduzir a infraestrutura de *hardware* (e os custos associados), para além de possibilitar facilmente acrescentar outras etapas de processamento.

Em muitas situações não é necessário reconfigurar totalmente o circuito FPGA; uma reconfiguração parcial, em que apenas uma parte do circuito é afectada, pode ser suficiente. (É precisamente o que acontece com o sistema de Villasenor *et al.* [1995]). Nestes casos, a transição entre cada etapa do algoritmo pode ser substancialmente encurtada. Também pode ser possível proceder a uma reconfiguração parcial enquanto o sistema funciona normalmente, o que faz com que o tempo de reconfiguração não influencie o desempenho global do sistema. Esta técnica é designada por "reconfiguração dinâmica local" já que afecta apenas uma parte localizada do sistema [Hauck, 1998b].

Outra perspectiva é aberta pela possibilidade de combinar circuitos FPGA com microprocessadores [Hauser e Wawrzynek, 1997; DeHon, 1994, 1995], o que já deu mesmo origem a alguns produtos comerciais [Faura *et al.*, 1997b]. A integração dos dois paradigmas de computação, além de abrir perspectivas interessantes, apresenta ainda muitos problemas. Esta combinação de paradigmas está também associada à reconfiguração dinâmica, já que é natural permitir a alteração das configurações pelo programa em execução (embora não exista ainda trabalho significativo nesta área).

A facilidade e rapidez de reconfiguração são as principais características exploradas para a realização de uma classe de sistemas geralmente agrupados sob a designação de "*hardware* evolutivo" (*evolvable hardware*). Este termo aplica-se a sistemas capazes de modificar dinamicamente a sua arquitectura e comportamento, de forma autónoma e num processo de interacção com o seu ambiente [Yao e Higuchi, 1996]. Frequentemente, os seus princípios organizacionais têm origem biológica [Sipper *et al.*, 1997]. De um modo geral podem distinguir-se neste domínio particular duas perspectivas, a primeira das quais enfatiza o desenvolvimento evolutivo de configurações como alternativa ao projecto de circuitos, enquanto a segunda dá prioridade a circuitos cujo comportamento apresente características biológicas como adaptabilidade, aprendizagem, replicação e auto-reparação (cicatrização).

Na primeira área podemos referir como exemplo o trabalho de Thompson [1996], que descreve o desenvolvimento de um circuito (i.e., configuração de FPGA) capaz de descriminar entre dois tons (a 1 kHZ e a 10 kHz). O desenvolvimento é guiado por um algoritmo genético em que os *bits* de configuração de um circuito Xilinx XC6216 constituem os "genes" manipulados. O resultado final do processo evolutivo apresenta o comportamento desejado, mas opera num regime não-digital e tem pouca robustez, deixando de funcionar quando as condições envolventes são alteradas (p.ex., a mesma configuração usada em outro componente da

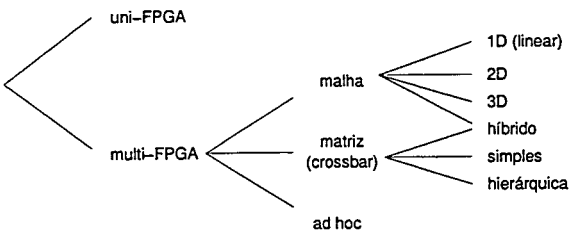


Figura 2.25: Classificação estrutural de sistemas reconfiguráveis.

mesma família apresenta comportamento diferente). Outros trabalhos nesta área são descritos por [Higuchi *et al.*, 1997] e [Keymeulen *et al.*, 1998].

O melhor exemplo da segunda vertente é talvez o trabalho de Tempesti [1998], em que se implementa uma primeira versão (baseada em circuitos FPGA com uma arquitectura especial) de um sistema capaz de auto-reparação e de auto-replicação.

Um outro projecto, designado por "CAM-Brain", pretende construir um sistema, baseado em circuitos Xilinx XC6264, capaz de simular um cérebro; o sistema tem uma capacidade prevista de 40 milhões de neurónios, cuja configuração será determinada por uma algoritmo genético [de Garis *et al.*, 1999; Nawa *et al.*, 1998]. O cérebro destina-se a controlar um gato *robot*. Trata-se de um trabalho em curso, com muitos aspectos em aberto.

A área de *hardware* evolutivo regista bastante actividade (com centro geográfico na Europa e no Japão) e apresenta algumas perspectivas intrigantes, mas os resultados são ainda preliminares, pouco representativos ou mesmo inconclusivos; é uma área de muitas promessas e desafios, mas também com muitas questões fundamentais ainda em aberto.

2.5.2 Estrutura e função

Para além da classificação por áreas de aplicação esboçada na secção anterior, os sistemas reconfiguráveis podem ser classificados segundo outros critérios. Embora cada sistema particular raramente se encaixe exactamente nas categorias referidas adiante—a utilização de estruturas reconfiguráveis fomenta uma grande diversidade de abordagens—a combinação de diferentes classificações permite obter uma visão global do sistema em análise.

2.5.2.1 Classificação estrutural

A organização estrutural de um SDR é caracterizada pelo número de circuitos FPGA usados e pela arquitectura da respectiva rede de interligações. Uma classificação muito simples é apresentada na figura 2.25.

Os sistemas mais simples contêm apenas um circuito FPGA, geralmente apoiado por memó-

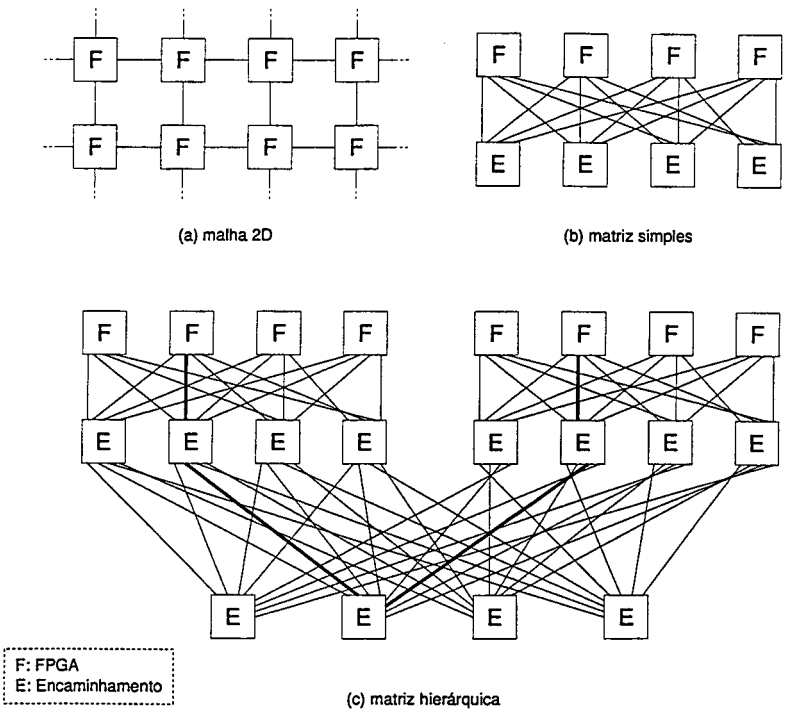


Figura 2.26: Topologias de interligação de circuitos lógicos e de encaminhamento.

ria RAM e circuitos de interface. São exemplos desta categoria as placas EVC1 [Chow *et al.*, 1995] e Hades [Ludwig, 1997].

A característica mais importante dos sistemas multi-FPGA é a topologia das respectivas interligações, que pode ser sumariamente classificada em três categorias:

malha A topologia em malha caracteriza-se por dotar cada circuito FPGA de ligações aos seus vizinhos mais próximos (fig. 2.26a). A localidade das interligações facilita a expansibilidade, porque torna fácil acrescentar de forma regular mais elementos ao sistema. A uma dimensão a estrutura torna-se num arranjo linear de elementos, como por exemplo no sistema Splash [Gokhale *et al.*, 1991]. Entre os exemplos de malhas 2D encontram-se os sistemas Springbok [Hauck *et al.*, 1994] e VirtualWires [Babb *et al.*, 1993]. Um sistema com malha 3D é descrito por Quénot *et al.* [1994].

matriz Neste caso, matrizes de barramentos cruzados (*crossbar*) são utilizadas para fazer o encaminhamento dos sinais entre circuitos lógicos. Existe geralmente uma separação funcional clara entre circuitos lógicos e circuitos de encaminhamento. Estes últimos podem ser simplesmente circuitos FPGA em que os blocos lógicos não são usados, ou

então circuitos específicos para estas funções [I-Cube, 1999]. Nos sistemas mais simples o encaminhamento entre quaisquer dois pinos é feito apenas através de apenas um circuito de encaminhamento, o que produz desempenhos globais previsíveis porque os tempos de atraso através da rede de interligações permanecem constantes (fig. 2.26b). O sistema BORG de Chan *et al.* [1992] e a arquitectura de encaminhamento dinâmico de Li e Cheng [1995] são exemplos desta topologia.

O sistema de interligações pode ser organizado hierarquicamente, recorrendo a circuitos de encaminhamento para interligar sub-matrizes de menor capacidade numa estrutura em árvore simétrica (figura 2.26c). Apenas os circuitos de encaminhamento do nível mais baixo estão directamente ligados aos circuitos lógicos. Para efectuar o encaminhamento de um sinal determina-se o nível a que os pinos de origem e destino partilham um mesmo ascendente; o percurso seleccionado vai do pino de origem a um desses ascendentes comuns e daí para o pino de destino: a figura 2.26c ilustra um percurso possível entre dois circuitos FPGA situados em sub-matrizes diferentes. O sistema de emulação Realizer [Butts *et al.*, 1992b; Varghese *et al.*, 1993] emprega este tipo de topologia.

A matriz hierárquica de encaminhamento combina a expansibilidade das topologias em malha com o encaminhamento relativamente simples dos arranjos em matriz. Em contrapartida, os sinais lógicos podem ter de passar por múltiplos circuitos de encaminhamento. Além disso, a quantidade de recursos dedicados apenas a encaminhamento pode ser substancial.

O carácter experimental dos SDR deu origem a muitas variações topológicas da rede de interligações, incluindo sistemas híbridos cuja topologia combina arranjo em malha e matriz de encaminhamento. Entre os sistemas que empregam topologias híbridas malha/matriz estão Splash 2 [Buell *et al.*, 1996], DECPeRLe-1 [Vuillemin *et al.*, 1996] e AnyBoard [van den Bout *et al.*, 1992].

ad hoc Existem muitos sistemas reconfiguráveis que usam topologias específicas únicas. Trata-se frequentemente de sistemas construídos para aplicações particulares, cuja topologia é cuidadosamente adaptada ao domínio de aplicação. Entre os sistemas desta categoria estão: GANGLION [Cox e Blanz, 1992], um sistema de classificação baseado em redes neuronais; ENABLE++ [Högl *et al.*, 1995], um sistema para reconhecimento de padrões em experiências de física das partículas; Virtual Computer [Casselman, 1993], um sistema computacional dedicado; e Spyder [Iseli, 1996], um processador reconfigurável com múltiplas unidades de execução.

Os sistemas multi-FPGA necessitam de apoio adicional de ferramentas automáticas para serem usadas com proveito [Hauck, 1995]. A principal tarefa consiste em decompor a implementação desejada pelos diversos circuitos FPGA, respeitando as restrições impostas pelo esquema de interligação (*partitioning and placement*). Posteriormente, é necessário efectuar

a atribuição de sinais lógicas a pinos de circuito FPGA (*pin assignment*), bem como efectuar o seu encaminhamento através da rede de interligações (*global routing*). As restantes tarefas são executadas ao nível do circuito FPGA individual.

2.5.2.2 Classificação funcional

Os sistemas computacionais dedicados podem ser classificados de acordo com a forma como estão associados ao sistema hospedeiro, ou seja, que função desempenham na organização global do sistema. A sua função depende essencialmente da forma como se efectua a comunicação com a unidade central de processamento (UCP), que por sua vez depende da posição que ocupam na hierarquia de barramentos do sistema. Quanto mais afastado o SCD estiver da unidade central, menor é débito de transferência de dados e, consequentemente, maior deve ser a granularidade da computação para permitir a sua utilização eficiente. Por ordem progressiva de afastamento da unidade central temos [Rubini, 1995]:

Unidade funcional integrada A unidade funcional integrada é um componente do processador, situada no seu barramento interno tal como a unidade lógico-aritmética ou de vírgula flutuante. Não tem capacidade própria de armazenamento de dados, sendo alimentada a partir dos registos do processador. Pode executar operações de muito curta duração (talvez mesmo de apenas um ciclo de relógio). Os sistemas híbridos FPGA/processador constituem um modo promissor de expandir a capacidade dos sistema baseados em microprocessadores. Para promover a sua adopção, é preciso mostrar que trazem mais benefícios que aproveitamentos alternativos da área de silício utilizada. Para além de diversos estudos e protótipos académicos [Razdan, 1994; Hauser e Wawrzynek, 1997; DeHon, 1994; French e Taylor, 1993], existem mesmo algumas implementações comerciais destinados ao mercado dos sistemas embutidos [Faura *et al.*, 1997b].

Co-processador O co-processador é uma unidade externa, com alguma capacidade de armazenamento. Encontra-se geralmente ligada ao barramento de memória (de alto débito) e torna-se efectivo para sequências curtas de operações (granularidade de computação intermédia). O co-processador é alimentado por acessos externos do processador, podendo eventualmente realizar operações de acesso directo à memória. Trata-se de uma abordagem usada com sucesso em outras áreas, por exemplo, cálculos em vírgula flutuante e aceleradores gráficos. Os SCD usados como co-processador são naturalmente maiores e mais sofisticados que as unidades funcionais integradas. Os protótipos são inúmeros; entre eles refiram-se os sistemas PRISM [Athanas e Silverman, 1993], CM-2X [Cuccaro e Reese, 1993] e ArMen [Raimbault *et al.*, 1993].

Unidade periférica As unidades periféricas são SCD com capacidade de funcionamento autónomo e armazenamento de dados. Encontram-se geralmente ligadas a barramentos de entrada/saída, naturalmente com débitos relativamente menores e tempos de latência

superiores aos das alternativas mencionadas atrás, mesmo quando usando técnicas sofisticadas de transferências de dados, como ,por exemplo, DMA (*direct memory access*). Frequentemente, o sistema hospedeiro tem apenas funções de controlo global e de interface com o utilizador. O débito de comunicação limita o desempenho destes sistemas; em contrapartida, podem ter dimensões físicas grandes e capacidade lógica muito avultada. Para amortizar o custo das transferências de dados, estes sistemas são geralmente responsáveis pelo esforço computacional do grosso da aplicação (granularidade de computação elevada). Quase todos os grandes sistemas pertencem a esta categoria: Splash 2 [Buell *et al.*, 1996], Teramac [Amerson *et al.*, 1995], Enable++ [Högl *et al.*, 1995], e outros.

Alguns autores (por exemplo, Hauck [1998b]) distinguem entre unidades periféricas internas (inseridas no interior no sistema hospedeiro) e externas (com embalagem autónoma). Esta distinção resulta mais de diferenças nas dimensões físicas do que de diferenças funcionais.

A função desempenhada por um SCD determina o tipo de algoritmos a que pode ser eficientemente aplicado. O aspecto mais importante é a combinação das operações de transferência de dados e das operações de processamento. O intuito é minimizar a contribuição das primeiras para o tempo total de execução, já que apenas as segundas contribuem efectivamente para o resultado final. Para um dado problema, a categoria funcional do SCD usado privilegia, logo à partida, aquele tipo de algoritmo cujas características (em termos de granularidade de computação) melhor se lhe ajustam.

2.6 Resumo

Os circuitos FPGA, surgidos apenas há década e meia, têm vindo a mostrar uma evolução impressionante em termos de capacidade: enquanto os primeiros circuitos podiam ser usados para implementar cerca de 2000 portas lógicas equivalentes, estão anunciados para o fim do ano 2000 circuitos com 10 milhões de portas lógicas equivalentes (utilizando para o efeito cerca de 500 milhões de transístores). A evolução tecnológica tem acompanhado o forte crescimento do mercado de componentes CMOS configuráveis, que atingiu o valor de 2,6 mil milhões de dólares em 1999.

Os circuitos FPGA evoluíram historicamente como resposta ao problema do aproveitamento económico e rápido da crescente capacidade de integração da tecnologia microelectrónica. As arquitecturas internas destes circuitos revelam a influência de três abordagens diferentes à implementação de sistemas digitais: matrizes de portas lógicas, matrizes homogéneas de componentes e circuitos configuráveis tipo PAL. A elevada densidade lógica e a configurabilidade destes circuitos tornaram-nos extremamente populares.

De entre as tecnologias de configuração adoptadas (anti-fusível, EEPROM e SRAM), foi a configuração por células SRAM que abriu caminho aos desenvolvimentos mais inovadores, por permitirem variar de forma rápida e confortável a funcionalidade dos sistemas nelas baseados.

Os principais recursos dos circuitos FPGA são os blocos lógicos (que executam o cálculo de funções lógicas) e a rede de interconexão (que assegura a comunicação entre blocos). Estes dois aspectos não são independentes, com opções numa área a influenciarem a outra. Conforme mostra a diversidade de arquitecturas comerciais e confirmam alguns estudos empíricos, não existe uma arquitectura óptima, mas um espectro de alternativas, cuja adequação varia com a área de aplicação.

Mais recentemente tem havido uma evolução no sentido de usar circuitos FPGA para implementar sistemas de um componente (*systems-on-a-chip*), o que tem levado os fabricantes a incluir módulos específicos como blocos de memória e circuitos PLL.

O surgimento dos circuitos FPGA conduziu a uma explosão de sistemas digitais reconfiguráveis que procuram aproveitar as suas potencialidades em novos e antigos domínios de aplicação. Aquando da sua última actualização, em Março de 1999, a lista informal de SDR mantida por Steve Guccione (`http://www.io.com/~guccione/HW_list.html`) registava mais de 90 sistemas.

Embora a diversidade de abordagens impeça uma categorização rigorosa dos SDR existentes, é possível classificá-los por área de aplicação (sendo as principais categorias emulação/prototipagem de sistemas digitais e sistemas computacionais dedicados), estrutura geral de comunicação (sistemas em malha ou com matriz de encaminhamento) ou integração com o sistema hospedeiro (unidade funcional, co-processador ou unidade periférica).

Comercialmente, são os sistemas de emulação aqueles que maior impacto tiveram, existindo diversos produtos no mercado, de companhias como Aptix, Quickturn e IKOS, que são usados activamente no desenvolvimento e validação de sistemas complexos (por exemplo, microprocessadores).

A prototipagem baseada em SDR também já é usada naturalmente sempre que se torna necessário desenvolver sistemas digitais específicos, existindo diversas empresas activas nesse mercado (para além de numerosos sistemas não-comerciais). As áreas de aplicação final são praticamente ilimitadas, indo desde as telecomunicações até à optimização combinatória [Ferreira *et al.*, 1998; Alves *et al.*, 1999].

A utilização de SDR como plataforma genérica de computação também mereceu a atenção de numerosas equipas de investigação, com vários sistemas de grande dimensão (DECPeRLe-1, Splash 2, Teramac) a mostrarem a validade da abordagem e os ganhos apreciáveis que podem ser obtidos ao mapear algoritmos directamente para estruturas de *hardware* apropriadas. Em muitos domínios parece razoável esperar que os sistemas dedicados com circuitos FPGA mantenham a sua vantagem competitiva, mesmo apesar dos avanços dos processadores de uso genérico [Vuillemin, 1998].

A flexibilidade de reconfiguração abre também novas vias, por exemplo, permitindo a especialização do SDR para instâncias específicas de um problema, a variação dinâmica de configurações e mesmo o surgimento de *hardware* evolutivo.

3 Reconfiguração dinâmica

Muito genericamente define-se reconfiguração dinâmica como a modificação de configurações *durante* a execução da aplicação, com as operações de reconfiguração e processamento a ficarem entrelaçadas temporalmente.

A reconfiguração dinâmica é uma forma de melhorar o aproveitamento dos recursos disponíveis através da reutilização de recursos lógicos e de interligação que de outra forma seriam (temporariamente) desaproveitados. Desta forma podem ser obtidas implementações com menor área (i.e., menor número de circuitos FPGA ou circuitos de menor capacidade) e consequentemente menor consumo e custo. A flexibilidade destes sistemas também abre caminho para novas aplicações, nomeadamente na área dos sistemas adaptativos.

Os sistemas digitais de reconfiguração dinâmica (SDRD) envolvem explicitamente o factor tempo, o que torna a sua concepção e validação potencialmente mais difíceis, e aumenta a necessidade de recorrer a ferramentas de apoio ao projecto. A evolução temporal das configurações pode ser usada como critério de classificação dos sistemas reconfiguráveis.

O objectivo deste capítulo é conferir uma visão panorâmica de diversos aspectos da reconfiguração dinâmica, que permita situar apropriadamente o trabalho exposto na segunda parte. O capítulo começa pela apresentação de vários critérios de classificação aplicáveis a circuitos de reconfiguração dinâmica, bem como uma caracterização genérica das suas características. De seguida são apresentados alguns circuitos representativos do tipo de suporte disponível actualmente para a reconfiguração dinâmica e também alguns sistemas em que são usados. Em terceiro lugar descrevem-se diferentes estratégias de reconfiguração dinâmica, a sua utilização e os problemas associados. Por fim são abordadas questões relacionadas com a implementação de ferramentas de apoio para a realização e utilização de sistemas de reconfiguração dinâmica, e são descritos os requisitos impostos às ferramentas de apoio, as técnicas de especificação usadas neste contexto, e ainda algoritmos de sequenciamento temporal, a principal tarefa específica deste tipo de sistemas.

3.1 Classificação e características

Para abordar os diversos aspectos de que se pode revestir a reconfiguração dinâmica, é útil diferenciar os SDR em função do modo de operação da infraestrutura reconfigurável, i.e., da forma como mudam os modos de operação durante o funcionamento do sistema. Grande parte dos SDR é configurada antes da execução da aplicação; essa configuração é mantida durante toda a execução. Este modo de operação do SDR pode ser designado por "reconfiguração inter-aplicação," por oposição à reconfiguração "intra-aplicação," "dinâmica" ou "em tempo de execução" (*run-time reconfiguration*) em que a configuração do sistema varia durante a execução.

Para caracterizar os SDRD em termos da evolução temporal da sua configuração existem dois aspectos a ter em consideração. O primeiro refere-se à maneira como é feito o sequenciamento temporal das configurações e permite distinguir entre sequenciamento fixo (estático) ou variável (dinâmico). O segundo aspecto diz respeito à ocasião em que são produzidas as configurações (quer totais, quer parciais) e permite distinguir entre configurações pré-definidas (isto é, produzidas antes da execução da aplicação ou *off-line*) e configurações produzidas durante a execução (*on-line*), possivelmente em função dos dados processados. A combinação dos dois factores permite distinguir quatro categorias de sistemas e aplicações:

I Os sistemas com **sequenciamento fixo** e **configuração pré-definida** vão mudando sucessivamente as suas configurações por uma ordem fixa. Essas configurações são escolhidas de entre um conjunto previamente definido. O sistema RRANN [Eldredge e Hutchings,

1994] é um bom exemplo desta abordagem aplicada a redes neuronais: o algoritmo de treino da rede (retro-propagação) é dividido em três estágios a executar sequencialmente; os circuitos FPGA são configurados para um estágio de cada vez. Os três estágios são repetidos iterativamente durante a fase de treino da rede neuronal. Esta abordagem permite aumentar cinco vezes o número de neurónios implementados em cada circuito FPGA.

II Nos sistemas com **sequenciamento variável** e **configuração pré-definida**, a escolha das configurações (a partir de um conjunto pré-estabelecido) e o instante da sua substituição são decididos em tempo de execução, dependendo do processamento efectuado. Este tipo de organização também aparece designado com o nome comercial de "cache logic" [Rosenberg, 1995]: uma memória é usada para guardar as diferentes configurações que são carregadas para o circuito FPGA à medida que são necessárias.

Outro exemplo desta categoria é o sistema DISC [Wirthlin e Hutchings, 1995], um processador muito simples (implementado num circuito FPGA), cujas instruções são implementadas como módulos removíveis, que podem ser trocados segundo as necessidades do programa em execução. A utilização de reconfiguração dinâmica remove a necessidade de distribuir os recursos existentes entre instruções de uso geral; instruções de aplicação específica podem ser usadas quando necessárias e descartadas posteriormente.

Outros sistemas de reconfiguração dinâmica pertencentes a esta categoria incluem os descritos por Chia *et al.* [1998] e Wirthlin e Hutchings [1997].

III Sistemas com **sequenciamento fixo** e **configuração dependente de dados** seguem sempre a mesma sequência de operações de reconfiguração; contudo, as configurações podem ser diferentes de iteração para iteração, geralmente dependendo da informação entretanto processada. Os filtros adaptativos descritos por Woolfries *et al.* [1998] são um exemplo desta abordagem. Nesta aplicação de transmissão vídeo, a supressão de ruído é efectuada por um filtro adaptativo, que é periodicamente ajustado às características do ruído estimadas a partir de imagens de teste inseridas no fluxo normal de dados. O sistema alterna periodicamente entre dois modos de operação. Na fase de treino correspondente à recepção das imagens de teste, o circuito FPGA é configurado para calcular as estatísticas relevantes (configuração fixa). Em seguida, um processador de apoio reconfigura o circuito FPGA para a fase de tratamento, gerando uma configuração nova com base nos dados obtidos na fase anterior.

IV Sistemas dos tipos II ou III podem ser naturalmente generalizados para sistemas com **sequenciamento variável** e **configuração dependente de dados**, potencialmente capazes de apresentar um comportamento adaptativo sofisticado. A existência de um número elevado de graus de liberdade coloca problemas à concepção e torna difícil o aproveitamento desta categoria de sistema, pelo que muito poucos exemplos se encontram descritos na literatura técnica. Um deles é o projecto de processador auto-reconfigurável descrito por

French e Taylor [1993]. O processador contém um interpretador de instruções e uma unidade reconfigurável. Da primeira vez que um fragmento de código é executada (pelo interpretador de instruções), o processador gera a configuração de um circuito funcionalmente equivalente; essa configuração é carregada para a unidade reconfigurável se e quando o fragmento for executado novamente.

A utilização de reconfiguração dinâmica acrescenta duas novas classes de problemas ao projecto dos sistemas [Hutchings e Wirthlin, 1995]. O primeiro resulta da necessidade de decompor o algoritmo em segmentos mutuamente exclusivos no tempo e a determinação da sequência de reconfigurações. Estes processos são designados por *decomposição temporal* e *sequenciamento*. Analogamente ao que acontece na decomposição estrutural, a decomposição temporal serve para organizar os elementos lógicos em grupos que operam (ou tendem a operar) simultaneamente; desta forma o algoritmo é dividido em fases distintas, correspondentes aos diversos modos de operação, cada um com a respectiva configuração.

O segundo problema introduzido pela reconfiguração dinâmica envolve a coordenação e transmissão de informação entre as diversas fases do algoritmo. Geralmente, cada fase produzirá resultados intermédios a serem utilizados em fases posteriores. O projecto deve ter necessariamente em consideração cuidadosa a forma como a coordenação é efectuada, que por sua vez depende fortemente dos meios disponibilizados pela infraestrutura usada.

Ao nível do sistema também se coloca a questão de como é feito o controlo das reconfigurações. As abordagens dependem naturalmente da composição do sistema, mas frequentemente a reconfiguração é controlada por circuitos exteriores ao circuito FPGA. Quando está disponível um processador convencional, por exemplo se o circuito FPGA funciona como coprocessador ou se temos uma unidade interna reconfigurável (como acontece com o microcontrolador FIPSOC [Faura *et al.*, 1997a]), o mais natural é deixar o controlo dessas operações ao processador. Contudo também é concebível que o circuito FPGA controle a própria reconfiguração, desde que possa funcionar normalmente durante a reconfiguração (como acontece com os circuitos da família Xilinx XC6200).

Um outro aspecto das técnicas de reconfiguração dinâmica prende-se com o seu âmbito [Hutchings e Wirthlin, 1995]. Uma reconfiguração global redefine de uma vez só todos os recursos disponíveis; uma reconfiguração local apenas afecta uma (possivelmente pequena) parte deles (figura 3.1). Esta última apenas é viável se a infra-estrutura suportar reconfigurações parciais. Reconfigurações globais são geralmente mais fáceis de gerir, podendo mesmo ser implementadas com circuitos FPGA convencionais. A divisão temporal do algoritmo é preferencialmente feita por fases, em que cada fase deve utilizar a grande parte ou totalidade dos recursos existentes. A reconfiguração local permite uma adaptação mais fina ao perfil de execução do algoritmo. Esta tipo de reconfiguração faz-se geralmente segundo uma divisão funcional da implementação, em que esta é repartida num conjunto de módulos que vão sendo carregados para o circuito FPGA à medida das necessidades; um mesmo módulo

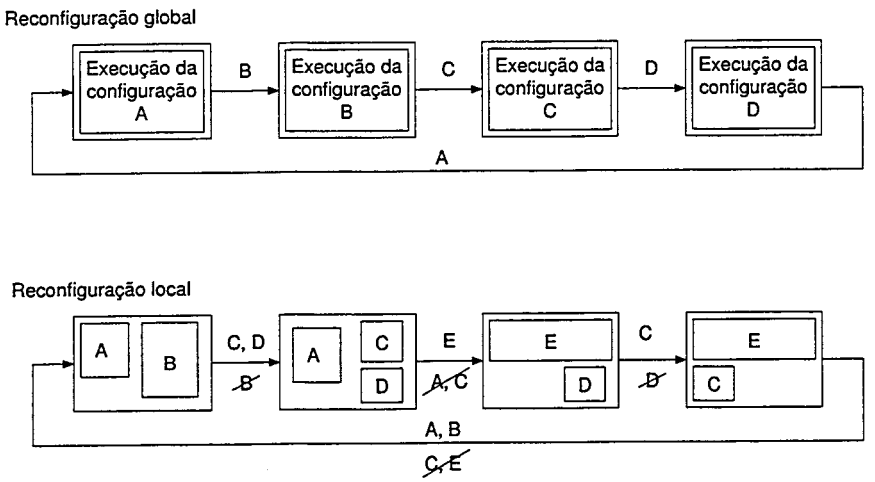


Figura 3.1: Reconfiguração global vs. local.

deve poder ser usado em diversas fases do algoritmo, de cada vez associado a outros módulos diferentes. A segunda abordagem é importante para aplicações que não são facilmente decompostas em fases mutuamente exclusivas de dimensão semelhante.

Segundo Hughes e Gunther [1998] a reconfiguração parcial pode ser isomórfica ou polimórfica. Na reconfiguração isomórfica apenas a funcionalidade de um módulo é alterada, e não as suas características físicas exteriores (número e posição dos blocos lógicos usados e respectivas interligações); a reconfiguração polimórfica pode alterar todos os aspectos do circuito. As restrições físicas impostas pela reconfiguração isomórfica facilitam grandemente a implementação das alterações dinâmicas, mas também restringem a sua aplicabilidade. Na prática, reconfiguração polimórfica pode ser empregue num número muito maior de situações e apresenta mais oportunidades de melhoria. Na prática, mesmo os sistemas polimórficos impõem algumas limitações, tanto para satisfazer restrições arquitecturais dos circuitos FPGA como para facilitar a colocação e interligação dinâmica dos módulos.

É uma observação conhecida que, num dado momento, uma parte substancial de um sistema lógico não contribui activamente para a computação. Esta observação é frequentemente explorada para diminuir o consumo energético das respectivas implementações, por exemplo, por inactivação de módulos temporariamente não usados. Também os simuladores digitais exploram o facto de apenas uma percentagem diminuta dos sinais lógicos mudar num dado intervalo de tempo, geralmente através da simulação selectiva dos eventos significativos (*event-driven simulation*).

Assim, a reconfiguração dinâmica pode ser encarada como um meio de aumentar a *densidade funcional* de uma implementação [Wirthlin e Hutchings, 1998]. Esta métrica é definida

como

$$D = \frac{1}{AT}$$

em que A representa a área ocupada (por exemplo, o número de blocos lógicos) e T o tempo de execução (tempo gasto na produção do resultado desejado). A métrica assim definida é uma medida do débito computacional (operações por segundo) por unidade de *hardware*; é, também, o valor inverso do custo computacional $C = AT$ de Seitz [1984].

No caso da reconfiguração dinâmica é necessário acrescentar ao tempo de execução o tempo despendido na reconfiguração. Nesse caso, a expressão anterior vem

$$D_{din} = \frac{1}{A(T_{exec} + T_{conf})}$$

em que, normalmente $T_{exec} < T$, porque a reconfiguração dinâmica permite utilizar implementações mais especializadas das operações. Esta expressão considera que, em qualquer dos casos, a área usada é idêntica, o que corresponde a assumir a utilização total do circuito FPGA. A expressão assume também que reconfiguração e execução são mutuamente exclusivas. Contudo, em alguns casos a reconfiguração pode ser efectuada sem interromper a execução; nesse caso podemos ter $T_{conf} = 0$.

O impacto das operações de configuração sobre o desempenho do sistema dependem do valor relativo do tempo de configuração face ao tempo total de execução. A razão de configuração $f = \frac{T_{conf}}{T_{exec}}$ é um parâmetro importante de um SDRD. A densidade funcional vem então dada por

$$D_{din} = \frac{1}{AT_{exec}(1+f)},$$

mostrando que reconfigurações longas podem ser admitidas desde que amortizadas durante tempos de execução correspondentemente maiores, de maneira a manter o valor de f pequeno.

Ainda segundo [Wirthlin e Hutchings, 1998], a melhoria M da densidade funcional de uma implementação dinâmica em comparação com a versão estática convencional pode ser avaliada por

$$M = \frac{D_{din} - D}{D} = \frac{D_{din}}{D} - 1.$$

O valor máximo da densidade funcional de um SDRD é obtido quando $f \to 0$:

$$D_{max} = \frac{1}{AT_{exec}},$$

a que corresponde a maior melhoria possível

$$M_{max} = \frac{D_{max}}{D} - 1.$$

Este parâmetro é uma boa medida do potencial aproveitamento de reconfiguração dinâmica para uma dada aplicação: quanto maior for M_{max}, maior é o aproveitamento potencial que pode ser retirado de uma abordagem dinâmica. Mostra-se facilmente por manipulações algébricas que a condição $D_{din} \geq D$ implica

$$f \leq M_{max}$$

ou seja, a reconfiguração dinâmica justifica-se quando a razão de configuração for inferior à máxima melhoria obtenível. A expressão justifica a noção intuitiva de que quanto maior for a vantagem potencial da reconfiguração dinâmica, menos exigentes são os limites para os custos adicionais (*overhead*) de reconfiguração.

Diversos estudos empíricos mostram que, em circunstâncias apropriadas, a reconfiguração dinâmica pode melhorar significativamente a densidade funcional de uma aplicação [Wirthlin e Hutchings, 1997; DeHon, 1996b], e um exemplo incluído em [Eldredge e Hutchings, 1996] exibe uma melhoria de 500%.

3.2 Circuitos e sistemas

A reconfiguração dinâmica é muito mais versátil e poderosa se for apoiada por circuitos FPGA com características apropriadas. Esta secção descreve alguns desses circuitos, bem como diversos protótipos de SDRD neles baseados.

3.2.1 Circuitos FPGA com suporte para reconfiguração dinâmica

São relativamente poucas as famílias de circuitos FPGA com suporte para reconfiguração dinâmica. A família Xilinx XC6200 é a que dispõe do suporte mais amplo para esta abordagem, possuindo de facto muitas das características consideradas desejáveis por Lysaght e Dunlop [1993] para este tipo de circuitos:

- reconfiguração rápida;
- reconfiguração dinâmica sem alterar operação das partes não afectadas (reconfiguração parcial);
- menor unidade de reconfiguração possível;
- acesso aos circuitos de controlo de reconfiguração a partir da matriz de blocos;
- recursos lógicos e de interligação regulares nas duas dimensões;
- endereçamento ortogonal (linha/coluna) dos blocos individuais;
- acesso para leitura e escrita aos elementos de memória dos blocos individuais.

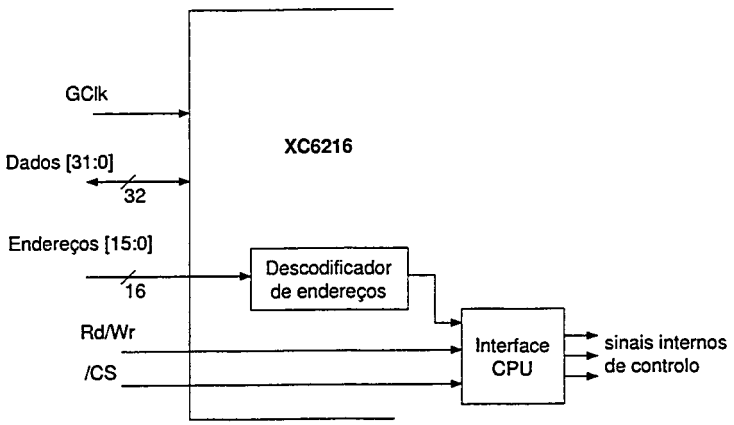


Figura 3.2: Interface de programação da família XC6200.

A arquitectura do bloco lógico desta família foi tratada na secção 2.3 (pág. 25) e o esquema de interligações foi descrito na secção 2.4 (pág. 39).

A interface FastMap descrita por Churcher *et al.* [1995], ilustrada na figura 3.2, permite o acesso directo tanto à informação de configuração como às saídas de todos os blocos lógicos; permite ainda colocar dados nos elementos de memória de qualquer bloco lógico individual. Todos os acesso podem ser feitos por um microprocessador convencional como se o circuito FPGA fosse uma memória; o endereço de cada item pode ser construído de forma sistemática a partir da posição geométrica do bloco lógico associado. Todos os aspectos da operação do bloco lógico e das respectivas interligações podem ser configurados independentemente. Desta forma é possível aceder directamente (para leitura ou escrita) a toda a informação de interesse.

A interface inclui ainda um registo que permite configurar o acesso em grupo aos elementos de memória dos blocos lógicos. Por configuração desse registo de mapeamento, o utilizador pode agrupar as saídas de até 32 blocos de uma mesma coluna para constituírem um registo, que passa a estar acessível do exterior como uma unidade, isto é, uma única transferência permite aceder a todos os elementos simultaneamente. Este mecanismo permite um acesso extremamente cómodo à informação que está a ser processada pelo circuito: é extremamente simples colocar valores dos operandos em registos definidos pelo utilizador, bem como aceder aos resultados produzidos. Todos os acessos usam a mesma interface, sem necessidade de utilizar pinos adicionais do circuito FPGA.

Para avaliar bem da comodidade desta interface, é preciso ter em conta que todas as outras famílias requerem que as configurações sejam formatadas numa sequência de *bytes* de estrutura mais ou menos complexa *(bitstream)*, que depois é descarregada para o circuito FPGA

por uma interface série. A necessidade de construir essa sequência dificulta sensivelmente a reconfiguração dinâmica.

A família XC6200 também permite o acesso interno aos sinais da interface, o que permite a uma implementação detectar acessos a certas posições e agir em conformidade, por exemplo, calculando um novo valor para um registo de saída ou dando início ao processamento do valor colocado num registo de entrada.

Um aspecto muito importante na prática é o da imunidade do circuito a erros de configuração. A arquitectura dos blocos lógicos e das interligações garante que um erro de configuração não causa danos eléctricos ao circuito (que poderiam ocorrer, por exemplo, se fosse criado inadvertidamente um curto-circuitos entre nós internos). Trata-se de uma característica extremamente benéfica que distingue esta família de quase todas as outras.

Esta família de circuitos, que já não se encontra em produção actualmente, foi sempre considerada pelo fabricante mais como um projecto experimental que como um produto comercial.

Para além da família XC6200, existem ainda outras famílias de circuitos FPGA comerciais com apoio, ainda que mais reduzido, à aplicação de técnicas de reconfiguração dinâmica. Dentre essas destacam-se as famílias AT6000 e AT40K, cuja arquitectura já foi analisada anteriormente (páginas 23 e 28). Beneficiando de arquitecturas uniformes de granularidade relativamente fina (especialmente no caso da família AT6000), estas famílias permitem a reconfiguração parcial sem afectar a operação da parte não alterada do circuito. A granularidade de reconfiguração é muito fina, já que cada bloco pode ser alterado de forma independente. Contudo, a configuração é feita descarregando uma sequência de *bytes*, cujo formato é relativamente complexo, através de uma interface série. Este facto dificulta sensivelmente o aproveitamento das capacidades de reconfiguração parcial, tanto mais que erros de configuração podem resultar em curto-circuitos internos (em especial, através das linhas de alta impedância).

Outros pontos negativos (por comparação com a família XC6200) são a ausência de apoio nativo para funcionamento como coprocessador e a impossibilidade de aceder aos elementos de memória internos. Consequentemente, a transferência de dados de e para o circuito FPGA não pode usar a interface de configuração, e obriga o utilizador a definir a sua própria infra-estrutura de comunicação recorrendo aos restantes pinos do circuito.

A família Virtex (já descrita na página 30) também tem algum suporte para reconfiguração parcial. Estes circuitos podem ser reconfigurados por colunas; contudo o formato da informação é bastante complicado [Kelem, 1999; Carmichael, 1999]. A informação é transferida por uma interface bidireccional simples de 8 bits. Também a leitura da configuração residente e dos valores dos elementos de memória pode ser efectuada através da mesma interface; o formato da informação é igualmente complicado. Durante a reconfiguração, a operação do circuito deve ser suspensa para evitar contenção nas linhas internas. Todas estas restrições resultam, na prática, num aproveitamento mais difícil destas capacidades.

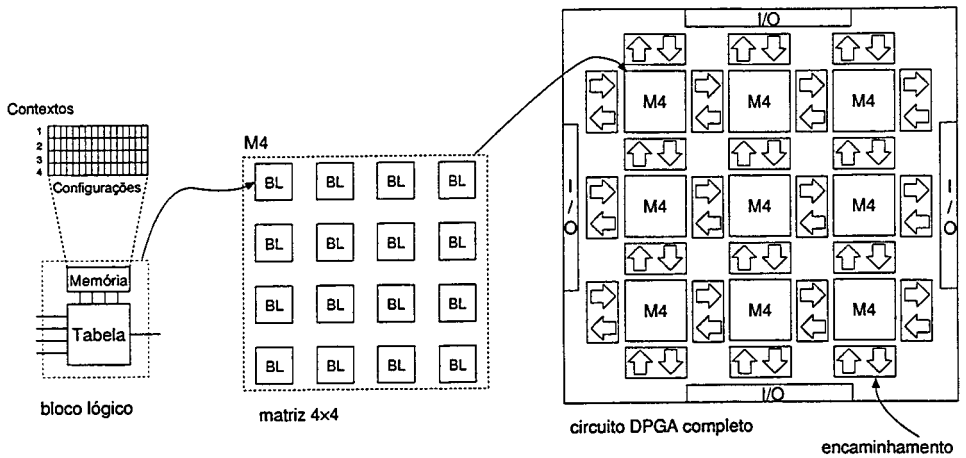


Figura 3.3: Arquitectura básica do protótipo DPGA.

Embora reconfiguração parcial seja uma das soluções possíveis para aumentar rapidez de reconfiguração, não é contudo a única. Outra possibilidade consiste em manter concorrentemente no mesmo circuito diversas configurações, de entre as quais apenas uma está activa num dado instante. O contexto de configuração activo pode ser mudado quase instantaneamente. Circuitos que suportam esta abordagem são designados por circuitos "multi-contexto" ou com "multiplexagem no tempo". Geralmente também é possível modificar as configurações inactivas sem perturbar a operação do circuito. Entre os circuitos que suportam esta abordagem contam-se o protótipo DPGA [Tau *et al.*, 1995; Brown *et al.*, 1995; DeHon, 1996b] e o microcontrolador comercial FIPSOC [Faura *et al.*, 1997a].

O circuito DPGA *(Dynamically Programmable Gate Array)* descrito por DeHon [1996b] (figura 3.3) é sumariamente caracterizado por:

- configuração por células de memória dinâmica (DRAM);
- blocos lógicos baseados em tabelas de 16 bits (quatro entradas);
- memória para quatro contextos de configuração;
- carregamento de novas configurações sem perturbar a operação do circuito;
- barramento de alta velocidade para transferência de configurações;
- refrescamento automático das células de memória dinâmica;
- estrutura de interligações com dois níveis.

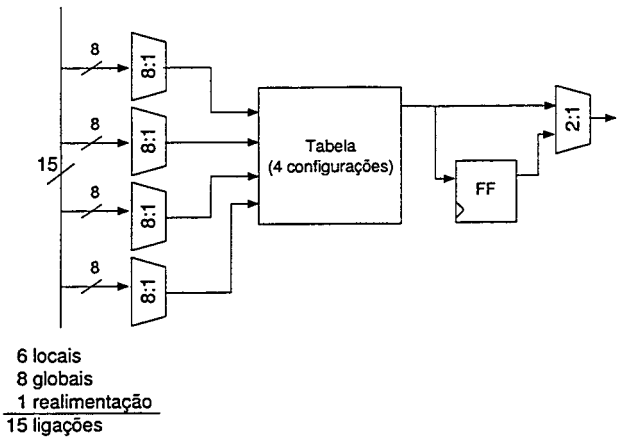


Figura 3.4: Bloco lógico do protótipo DPGA.

Cada bloco lógico (figura 3.4) é composto por uma tabela RAM com quatro entradas e um *flip-flop* opcional na saída. Um bloco lógico tem 32 bits de configuração, pelo que cada bloco inclui um bloco de 4×32 bits de memória dinâmica para armazenar os quatro contextos de configuração.

Os blocos lógicos estão agrupados em matrizes de 4×4, que formam o nível inferior da hierarquia de interligações. Cada saída de um bloco lógico é distribuída horizontal e verticalmente pela matriz; assim cada bloco tem seis sinais de entrada provenientes dos blocos da sua linha ou coluna. Cada bloco tem ainda acesso a oito linhas globais, provenientes das matrizes vizinhas.

Entre as matrizes 4×4 existem pares de circuitos de encaminhamento que levam as 16 saídas de uma matriz às 8 entradas não-locais da matriz adjacente. A cada circuito de encaminhamento 16×8 está associado um bloco de memória DRAM 4×32 que armazena as respectivas configurações.

A interface de configuração permite aceder à memória DRAM como uma única memória síncrona de palavras de 32 bits e suporta comunicação de alto débito com um processador associado.

O mapeamento automático de diversos exemplos para o protótipo DPGA permitiu obter reduções de área de 20% a 40% em relação a um circuito FPGA convencional [DeHon, 1996b]. Para o caso especial de máquinas de estados finitas essa redução varia entre 30% e 40%. Os resultados são ainda melhores quando o mapeamento é feito manualmente, obtendo-se então uma redução da área total para cerca de um terço.

O circuito FIPSOC (figura 3.5) inclui uma matriz de blocos lógicos, um conjunto de células analógicas e mistas analógico/digitais (amplificadores, filtros, comparadores, conversor A/D e

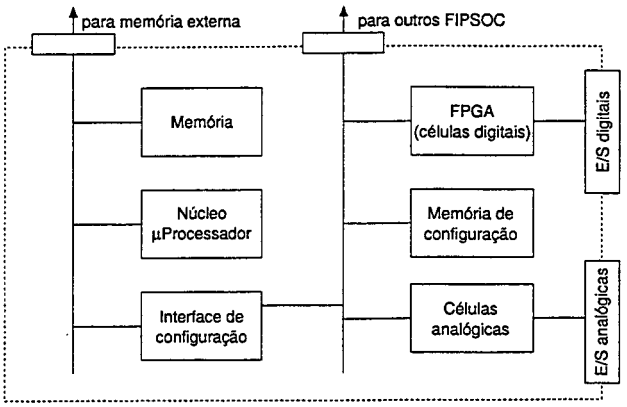


Figura 3.5: Diagrama de blocos do circuito FIPSOC.

D/A) e um núcleo de microcontrolador compatível com o processador Intel 8051.

Cada bloco lógico (figura 3.6) está dividido em duas secções, uma combinacional e outra sequencial, ligadas por um circuito de encaminhamento. A secção combinacional é composta por quatro tabelas de quatro entradas cada. Pares de tabelas partilham duas entradas e podem ser combinadas para implementar uma função de 5 entradas ou um multiplexador 4:1. As quatro tabelas de um bloco podem ser combinadas para formarem funções booleanas de 6 entradas. A secção combinacional pode ainda ser configurada como duas memórias SRAM 16×2 ou como um somador/subtractor de 4 bits. A secção sequencial inclui quatro *flip-flops* de duas entradas que podem ser interligados em cascata para formarem registos de deslocamento ou contadores.

A rede de interligações é composta por pistas que abrangem um, dois ou quatro blocos, tanto vertical como horizontalmente. Também existem pistas longas que abarcam toda a largura e altura da matriz, bem como pistas dedicadas para a distribuição de sinais globais de relógio e *reset*. Os interruptores da rede de interligações são controlados por células de memória SRAM. O processador pode aceder às saídas de todos os blocos lógicos, bem como ler ou modificar todos os elementos de memória. As saídas digitais dos blocos mistos também podem ser ligadas à matriz reconfigurável.

A configuração do circuito é gerida pelo processador interno. A memória de configuração está organizada em palavras mapeadas no espaço de endereçamento do processador. Esta memória dispõe de capacidade para armazenar dois contextos de configuração (um dos quais constitui o contexto activo), bem como uma cópia dos valores de todos os *flip-flops*. O processador pode aceder à memória de configuração para modificar o contexto de configuração inactivo. O contexto de configuração pode ser comutado sob comando do processador no tempo equivalente a um ciclo de escrita. A reconfiguração também pode ser limitada a um

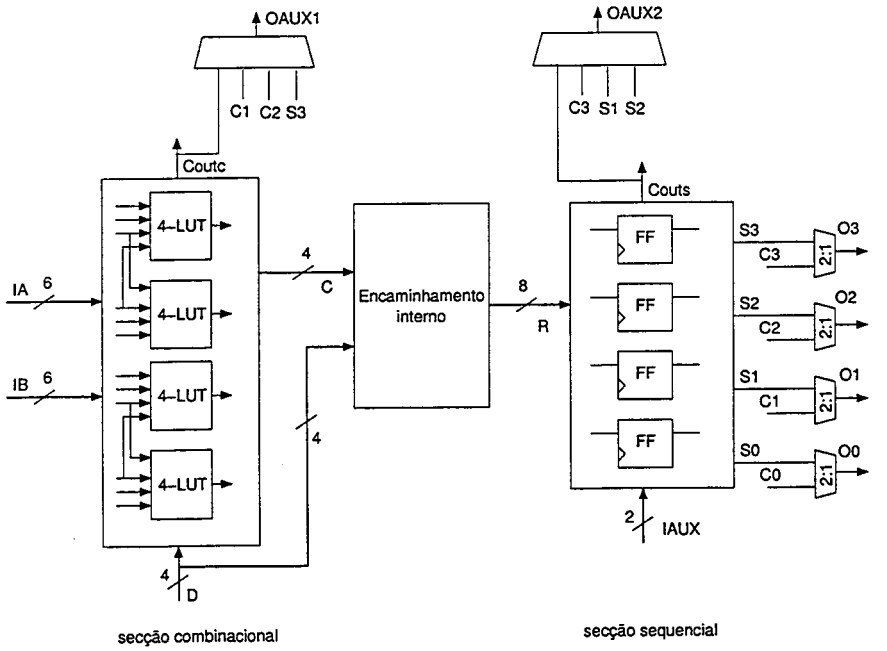


Figura 3.6: Bloco lógico simplificado do circuito FIPSOC.

subconjunto das células. A inclusão dos valores do elementos de memória no contexto de configuração permite que o estado deste elementos seja preservado durante as trocas ou inicializado antes da activação.

Existem outros exemplos de circuitos multi-contexto. Um dos mais antigos é a arquitectura Dharma [Bhat *et al.*, 1993], cujo diagrama de blocos pode ser visto na figura 3.7. O objectivo desta arquitectura é fomentar a partilha de recursos e simplificar as ferramentas de apoio. A configuração da matriz de encaminhamento e dos blocos lógicos (tabelas de memória) é modificada em cada ciclo de relógio, permitindo a reutilização dos recursos em cada fase da execução. A matriz de encaminhamento permite ligar qualquer entrada a qualquer saída, o que associado à simplicidade do arranjo uni-dimensional dos blocos lógicos, facilita a tarefa das ferramentas de apoio.

Trimberger *et al.* [1997] descrevem a extensão de um circuito da família Xilinx XC4000E com memória para oito configurações adicionais. A activação de um contexto pode ser feita num único ciclo de relógio (40 ns). Configurações podem ser carregadas sem interromper a execução da aplicação. Memória de configuração não usada é acessível em bloco para utilização como memória de dados. O protótipo suporta três modos de operação: para além do modo estático, em que não são efectuadas modificações de configuração, o protótipo pode emular

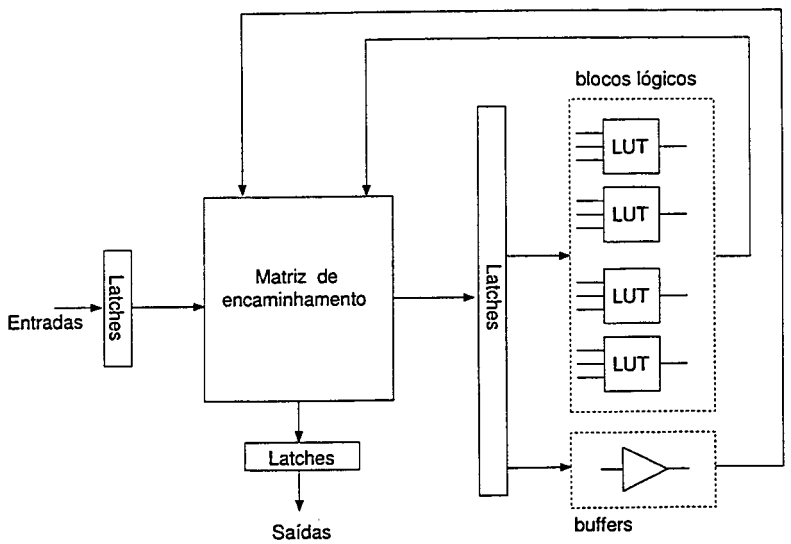


Figura 3.7: Diagrama de blocos da arquitectura Dharma.

um único circuito FPGA de grande capacidade (o valor de um elemento de memória pode ser acedido de qualquer contexto) ou vários circuitos FPGA em comunicação (cada contexto preserva o valor dos seus elementos de memória).

Fujii *et al.* [1999] descrevem um circuito com suporte para oito configurações simultâneas e capaz de trocar de configuração em 4.6 ns, sem interromper a operação do circuito um único ciclo de relógio. A arquitectura inovadora dos blocos lógicos permite que estes funcionem em modo lógico (implementando funções booleanas) ou em modo de encaminhamento. A vantagem desta arquitectura "multi-modo" reside na possibilidade de variar a distribuição de recursos entre interligações e funções lógicas da maneira que se adaptar melhor ao circuito em causa.

3.2.2 Alguns sistemas digitais reconfiguráveis dinamicamente

A literatura técnica refere vários SDRD construídos em torno de alguns dos circuitos FPGA mencionados previamente. A grande maioria são protótipos, projectados para servirem de veículo a trabalhos de investigação.

A figura 3.8 mostra a organização do sistema H.O.T. Works [VCC], um dos poucos sistemas deste tipo disponíveis comercialmente (sob várias designações diferentes) e que serve de base ao trabalho descrito na segunda parte desta dissertação. O sistema, que se destina a operar como coprocessador num computador pessoal convencional (compatível IBM PC), contém um circuito Xilinx XC6216, 2 MB de memória estática e uma interface para barramento PCI.

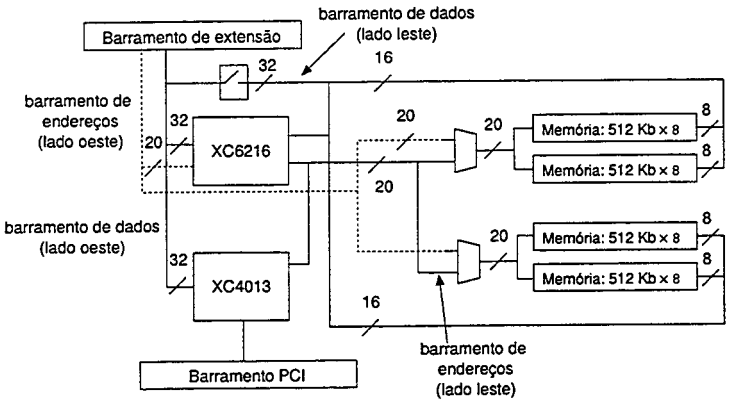


Figura 3.8: Diagrama do sistema H.O.T. Works.

A interface PCI ocupa metade de um circuito FPGA XC4013E; o resto é usado para circuitos de controlo, incluindo a monitorização da corrente consumida pelo circuito FPGA principal. O acesso ao circuito XC6216 é completo: tanto a memória de configuração como os blocos individuais podem ser acedidos directamente.

A memória está organizada em dois bancos, cada um com duas memórias estáticas de 512K×8. Um banco pode ser acedido a partir da interface PCI ou do circuito XC6216. Os bancos de memória têm dois barramentos de endereços independentes e quatro sinais independentes de leitura/escrita. Os diferentes modos de acesso a memória estão indicados na figura 3.9, ilustrando a flexibilidade organizacional do sistema.

O sistema encontrou aplicação na prototipagem de aplicações de reconfiguração dinâmica, por exemplo, na área de processamento de vídeo [Luk *et al.*, 1998].

RIFLE-62 é outro sistema de prototipagem baseado em circuitos da família XC6200 [Vasilko e Long, 1998a,b]. A arquitectura do sistema está indicada na figura 3.10. A placa pode operar isoladamente ou inserida num computador pessoal com barramento PCI. São usados três circuitos FPGA: qualquer elemento da família XC6200 pode ser usado como elemento reconfigurável dinamicamente; um circuito XC4013E é usado para implementar a interface PCI e um circuito da série XC3000 implementa o controlador de memória dinâmica.

A placa tem capacidade para 128 MB de memória dinâmica. A memória estática está organizada em dois bancos, o primeiro com uma capacidade de 512 KB×32 e o segundo 512 KB×16. O primeiro banco serve para armazenar configurações para o circuito XC6200, enquanto o segundo guarda os respectivos endereços; ambos também podem ser usados para armazenar dados genéricos.

O sinal de relógio para a placa pode ser fornecido exteriormente (barramento PCI, interface para microprocessador ou interface para placa auxiliar) ou criado internamente por um

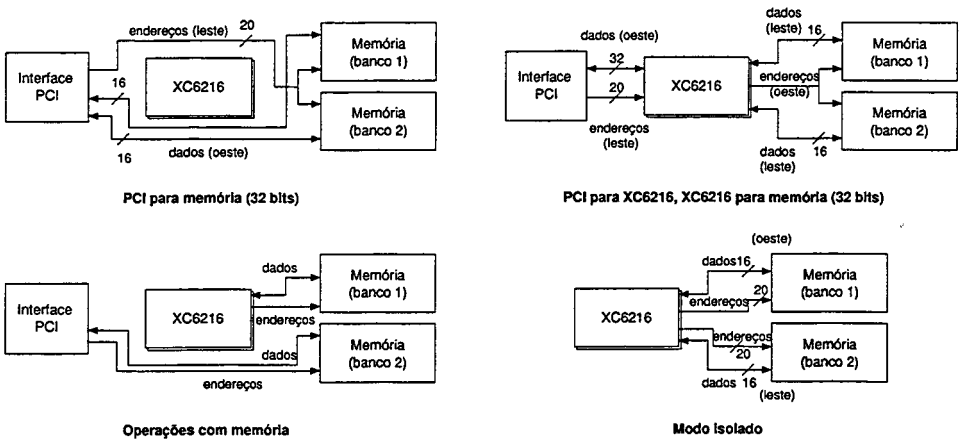


Figura 3.9: Modos de operação da placa H.O.T. Works.

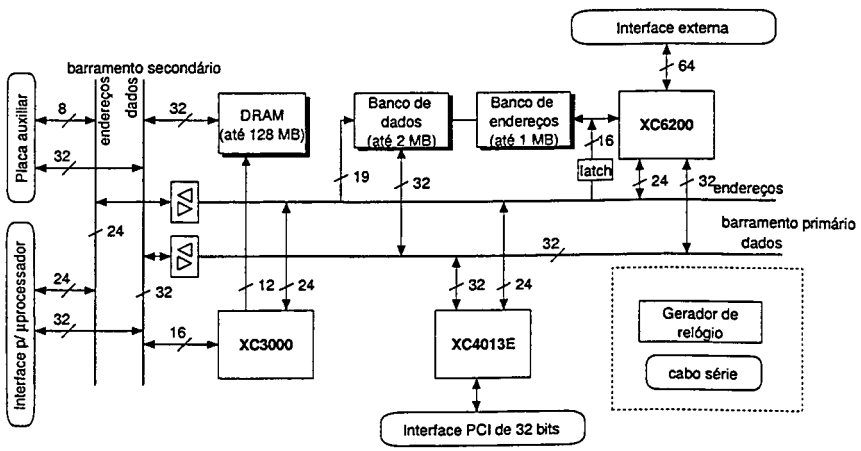


Figura 3.10: Diagrama de blocos do sistema RIFLE-62.

gerador dedicado.

Os circuitos FPGA estão ligados por um barramento comum com 32 bits de dados e 24 bits de endereços. O barramento está separado em duas secções, primária e secundária, que podem ser usadas de forma independente: por exemplo,a memória dinâmica pode ser carregada a partir da interface de microprocessador , enquanto os restantes elementos processam dados da memória estática.

RIFLE-62 dispõe de quatro interfaces:

Interface PCI Interface normalizada usando um barramento de 32 bits multiplexado para dados e endereços;

Interface de extensão XC6200 64 sinais provenientes do circuito XC6200 estão ligados a um conector; destinam-se a possível expansão do sistema (circuitos da série XC6200 adicionais) ou a interligar dois sistemas RIFLE-62;

Interface para microprocessador Interface para um processador dedicado residente numa placa de expansão; fornece acesso à secção secundária do barramento interno do sistema.

Interface para placa auxiliar Interface simples para placas auxiliares com necessidades pouco sofisticadas de comunicação; partilha alguns sinais com a interface anterior.

Novas configurações podem ter três origens: cabo série, memória PROM residente (especialmente útil durante a inicialização) e computador hospedeiro via barramento PCI (após configuração da interface). A reconfiguração dinâmica do circuito XC6200 pode ser controlada pelos circuitos FPGA XC3000 e XC4013E, por um controlador externo ligado à interface de microprocessador ou pelo próprio circuito XC6200. O circuito XC3000 pode ser reconfigurado directamente a partir do circuito XC4013E.

Mackinlay *et al.* [1997] apresentam um outro sistema de prototipagem baseado na família XC6200. A arquitectura do sistema Riley-2 está ilustrada na figura 3.11. O sistema inclui um microprocessador (Intel i960RP a 33 MHz) e quatro circuitos FPGA XC6216, cada um associado a uma memória estática local de 128K×32. Riley-2 tem dois barramentos, o barramento local do microprocessador e um barramento para os recursos reconfiguráveis (32 bits). Ligada ao barramento do microprocessador está ainda uma memória dinâmica de 16 MB, partilhada pelos diversos elementos. O microprocessador inclui uma interface PCI que permite mapear a memória partilhada e os recursos reconfiguráveis no espaço de endereçamento do sistema hospedeiro (qualquer computador pessoal com barramento PCI). A interface entre os dois barramentos é implementada por um circuito FPGA, que controla ainda um porto de entrada/saída com 44 pinos para acesso a dispositivos externos de alto débito (por exemplo, fontes de sinais vídeo). O sistema tem encontrado uso em aplicações de processamento de imagem.

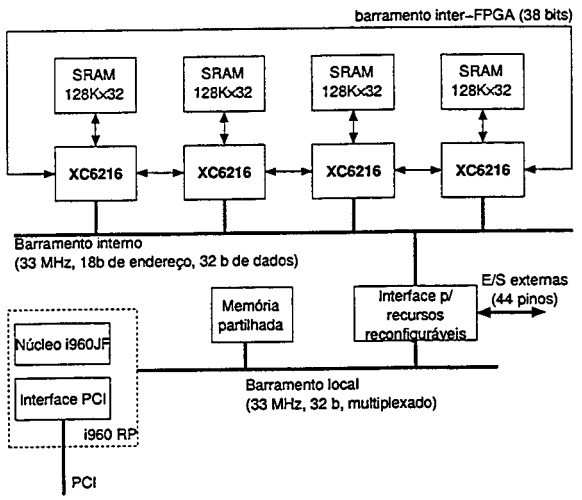


Figura 3.11: Diagrama de blocos do sistema Riley-2.

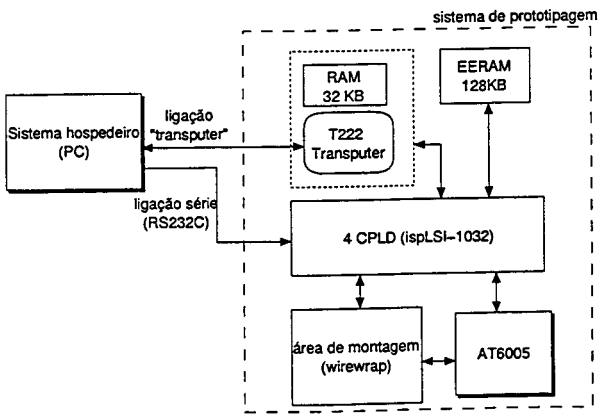


Figura 3.12: Diagrama de blocos do sistema de prototipagem de Lysaght *et al.* [1995].

Embora todos os SDRD mencionados usem circuitos da família XC6200, existem outras alternativas. O sistema de prototipagem de Lysaght *et al.* [1995] reúne um processador Transputer T222 (com 32 KB de memória local), 128 KB de memória EEPROM e um circuito Atmel AT6005 numa mesma placa (figura 3.12). Quatro circuitos PLD são usados para implementar a interface entre *transputer* e circuito FPGA. A memória pode ser usada para guardar dados de configuração, com os circuitos CPLD a fornecerem a interface entre esta e o circuito FPGA. A memória também pode ser acedida a partir do *transputer*, igualmente sob o controlo da lógica implementada nos circuitos PLD. O sistema inclui ainda uma área onde podem ser montados componentes adicionais.

O sistema é controlado a partir de um computador pessoal equipado de uma placa com um *transputer*. A ligação série bidireccional entre este *transputer* e o que existe na placa de prototipagem constitui o principal meio de transferência de dados. Os circuitos PLD podem ser configurados através de uma ligação série RS232C.

3.3 Estratégias de reconfiguração

O objectivo principal da reconfiguração dinâmica é o aproveitamento de recursos através da sua reutilização no decurso da execução. Adicionalmente, a reconfiguração dinâmica é uma técnica que pode ser usada para implementar sistemas adaptativos, porque permite alterar a implementação em reacção a actividade exterior ao sistema. Esta subsecção ilustra a forma e a variedade de que o aproveitamento desta técnica se reveste.

O uso da reconfiguração dinâmica está naturalmente condicionado pela área de aplicação e pela arquitectura global do sistema. Reciprocamente, a utilização da reconfiguração dinâmica influencia a abordagem ao problema e inspira novas possibilidades. Os exemplos apresentados a seguir ilustram as diferentes facetas deste jogo de forças para aplicações sucessivamente mais complexas.

3.3.1 Adaptação a condições iniciais específicas

A literatura técnica regista um grande número de formas de explorar a reconfiguração em tempo de execução *(on-line)*, desde as mais simples (em que se procura apenas adaptar o sistema a condições iniciais específicas) até às mais sofisticadas, em que o comportamento dinâmico complexo é a característica mais relevante.

Um exemplo da forma mais simples de reconfiguração em tempo de execução é descrito por Gunther *et al.* [1996] no contexto da pesquisa de palavras numa base de dados de textos não indexados. Trata-se de um sistema que determina quais os textos (de entre um vasto conjunto) que contêm um dado conjunto de palavras. A pesquisa envolve passar todos os textos por um sistema de *hardware* dedicado que detecta a ocorrência das palavras procuradas. O sistema é composto essencialmente por uma matriz 4×4 de circuitos FPGA CAL1024 (Algotronix).

A pesquisa é efectuada em duas fases. Na primeira, as palavras a pesquisar são embebidas nos circuitos por reconfiguração parcial de um circuito pré-definido, isto é, os andares de detecção são configurados para detectarem especificamente cada letra das palavras pretendidas na sua posição correcta. Na segunda fase, o conjuntos de textos (na ordem das dezenas de *Megabytes*) é passado através do sistema dedicado, sendo contabilizadas as detecções para avaliar da relevância do documento. A infra-estrutura digital tem capacidade para cerca de 100 palavras com um comprimento médio de 12 caracteres. Numa aplicação envolvendo cerca de 90 palavras, foram necessários 3.21 s para preparação da pesquisa (0.16 s na síntese dos detectores e 3.05 s na reconfiguração). A pesquisa da base de dados foi executada a uma taxa de 20 MB/s.

A aplicação de princípios semelhantes já tinha sido feita por Lemoine e Merceron [1995] no contexto da pesquisa de bases de dados genéticas. A infra-estruturada usada foi o sistema DECPeRLe-1 [Vuillemin *et al.*, 1996]. Também neste caso, a configuração inicial é manipulada para representar os padrões genéticos procurados. Foram ensaiadas duas abordagens alternativas para gerar as configurações específicas. A primeira consiste em modificar directamente a informação de configuração *(bitstream)* antes de a enviar para o sistema DECPeRLe-1; por razões de simplicidade as modificações possíveis são restringidas à modificação do conteúdo das tabelas dos blocos lógicos (ou seja, era possível mudar a função lógica dos diversos elementos, mas não as suas interligações). A abordagem alternativa baseia-se numa ferramenta dedicada que, a partir da descrição dos padrões genéticos procurados, permite gerar rapidamente uma colocação dos diversos elementos de tal forma que o subsequente passo de encaminhamento também é muito rápido (no total, são necessários cerca de 90 s para a geração da configuração).

A modificação de *bitstreams* como técnica de reconfiguração já tinha sido usada no sistema GANGLION [Cox e Blanz, 1992], uma rede neuronal artificial com neurónios digitais implementada sobre uma infra-estrutura de circuitos FPGA da família XC3000. Após a fase de treino, a informação de configuração é manipulada de forma a incluir directamente os valores dos diferentes parâmetros da rede neuronal: todos os pesos, valores de referência e factores de escala são embebidos directamente na estrutura da implementação.

Os exemplos referidos fazem uma utilização restrita das capacidades de reconfiguração, que apenas são usadas numa primeira fase para fazer a adaptação do sistema; não é sequer indispensável ter suporte específico nos circuitos FPGA. Já os exemplos seguintes ilustram uma exploração mais agressiva das possibilidades de reconfiguração.

3.3.2 Sequenciamento fixo de configurações

O aproveitamento mais óbvio de reconfiguração dinâmica consiste em trocar ciclicamente a configuração activa (sistemas do tipo I na classificação da secção 3.1, pág. 60): é o que acontece, por exemplo, no sistema de identificação de alvos em imagens de radar descrito por Chia

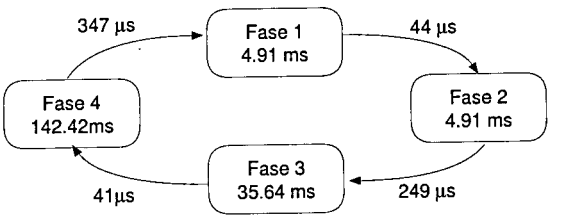


Figura 3.13: Ciclo de reconfiguração do sistema de interpolação de imagens de Hudson *et al.* [1998].

et al. [1998]. Para levar a cabo a identificação de alvos, o sistema necessita de correlacionar milhares de padrões (correspondentes a objectos conhecidos) com secções de uma imagem de radar. Para tal são criados circuitos que calculam a correlação do padrão fixo com a imagem do radar; a sua estrutura lógica é determinada pelo padrão correspondente. Estes circuitos são agrupados em conjuntos de oito para formar uma configuração. O processamento de uma imagem consiste em fazê-la passar pelo circuito FPGA configurado apropriadamente, que calcula as possíveis correlações com os primeiros oito padrões. De seguida o circuito FPGA é reconfigurado para calcular a reconfiguração com os oito padrões seguintes, e assim sucessivamente. O sistema Mojave é construído em torna de dois circuitos FPGA XC4013, um dos quais é reconfigurado dinamicamente (reconfiguração global), enquanto o outro implementa a interface a um microprocessador Intel i960. O sistema, operando a 12.5 MHz, necessita de 16 ms para avaliação da correlação (com oito padrões) e de 30 ms para reconfiguração, atingindo assim um débito de 168 padrões por segundo, desempenho semelhante ao obtido por uma implementação alternativa com cinco CIAE.

A presença de um microprocessador no sistema permite contemplar abordagens mais sofisticadas, como por exemplo restringir os conjuntos de padrões a correlacionar com base nos resultados das imagens anteriores. Neste cenário, não explorado por Chia *et al.* [1998], já teríamos um sistema de tipo II, em que as configurações são fixas mas o seu sequenciamento temporal varia com o decurso da execução.

O sistema de interpolação de imagens de Hudson *et al.* [1998] é outro exemplo da categoria I. Neste caso um circuito FPGA XC6264 é reconfigurado num ciclo de quatro fases sucessivas, cada qual implementando uma parte de um algoritmo para aumentar a resolução de imagens digitais. O tempo gasto na execução de cada fase e nas reconfigurações entre fases está indicado na figura 3.13. O recurso à reconfiguração parcial permite manter a duração desta actividade abaixo de 1% do tempo total. A latência total do sistema é de 188 ms por imagem, o que se traduz num débito de cerca de 5 imagens por segundo. Como o algoritmo é facilmente paralelizável, um sistema com oito circuitos FPGA poderia obter um débito total de 40 imagens por segundo.

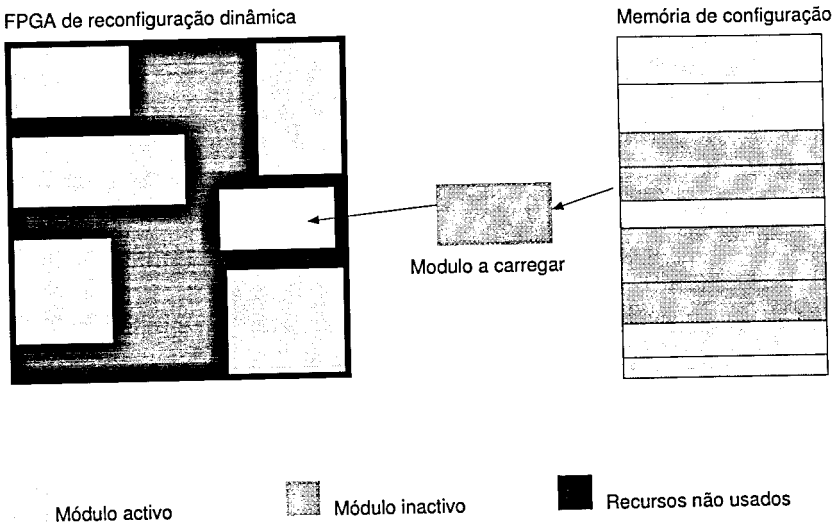


Figura 3.14: O conceito de *cache logic*.

A reconfiguração parcial traz benefícios consideráveis aos sistemas que a adoptem, já que pode reduzir substancialmente o tempo de reconfiguração. O sistema RRANN, já mencionado na página 60, é um exemplo claro deste facto. A versão original usa reconfiguração dinâmica *global* para implementar um algoritmo de treino de redes neuronais. A análise dos seus resultados mostra que o sistema passa grande parte do seu tempo em reconfiguração: o seu desempenho só ultrapassa o de uma implementação estática para redes grandes (mais de 23 circuitos FPGA), e mesmo neste caso, o sistema emprega cerca de 80% do seu tempo em reconfiguração. Uma segunda versão, RRANN2 [Hadley, 1995], usa reconfiguração parcial para obter uma considerável melhoria de desempenho já que apresenta uma redução de 25% do tempo de reconfiguração e um aumento de 50% no número de neurónios implementados.

3.3.3 *Hardware* virtual

Os exemplos apresentados até aqui pertencem à categoria I da classificação apresentada na secção 3.1. É uma extensão natural considerar sistemas em que o sequenciamento temporal das diferentes configurações não está fixado à partida, mas vai evoluindo de acordo com as necessidades da aplicação.

Uma parte importante da actividade desta área gira em torno dos conceitos de *hardware virtual* ou *cache logic*. A ideia consiste em implementar um dado sistema digital de grandes dimensões em circuitos FPGA de menor capacidade, instanciando nestes apenas as secções activas do circuito global, que nunca é realizado simultaneamente na sua totalidade.

O conceito de *cache logic* [Lysaght e Dunlop, 1993] resulta da observação de que circuitos FPGA (encarados como memórias activas) representam um nível da hierarquia de memória (baixa capacidade, elevado custo) diametralmente oposto à memória RAM externa (elevada capacidade, baixo custo). Os circuitos FPGA funcionam assim como memórias *cache* de circuitos lógicos activos (ver figura 3.14). Os diferentes sub-circuitos são colocados no circuito FPGA quando necessário; quando não estão em utilização podem ser substituídos por outros circuitos activos provenientes da memória externa. O controlo das operações de reconfiguração pode ser feito por um processador externo ou, alternativamente, por uma unidade de controlo colocada no circuito FPGA, desde que este suporte reconfiguração parcial sem interrupção das outras operações. Esta estratégia também foi divulgada pela companhia Atmel para utilização com os seus circuitos FPGA [Rosenberg, 1995], infelizmente sem ferramentas de apoio apropriadas.

O conceito de *hardware* virtual é similar, mas estabelece a analogia com outra parte da hierarquia de memória tradicional. Num sistema operativo moderno, o espaço de endereçamento disponível para uma aplicação é dividido em páginas. Uma página pode estar activa (residente em memória) ou inactiva (residente em memória secundária, por exemplo, disco duro). Quando o processador tenta aceder a um endereço de uma página inactiva, o sistema de gestão de memória transfere a a página de disco dura para memória principal, se necessário substituindo uma página residente. Singh e Bellec [1994] preconizam uma abordagem análoga aos sistemas reconfiguráveis, em que o circuito FPGA é a memória principal para onde são transferidos os sub-circuitos residentes em memória externa. Desta forma pode ter-se um sistema virtual de muito maior dimensão que a área física do substrato reconfigurável. A principal diferença reside no facto de os sub-circuitos não serem necessariamente todos de idênticas dimensões, ao contrário do que sucede com as páginas de memória virtual.

Embora apresentada originalmente no contexto de uma biblioteca de módulos para processamento gráfico, a abordagem é generalizável a sistemas cuja funcionalidade seja decomponível em módulos funcionais a operarem de forma mutuamente exclusiva no tempo. Numa situação ideal, o circuito virtual seria projectado sem atender à capacidade do substrato reconfigurável e a colocação dos módulos activos no circuito FPGA seria feita em tempo de execução por um sistema dedicado de gestão de recursos. Fornaciari e Piuiri [1998] descrevem mesmo um cenário em que um coprocessador FPGA é partilhado (sob gestão do sistema operativo) por diversos processos de um sistema multi-tarefa. Cada processo teria a ilusão de ter o coprocessador completamente disponível só para si. Todos os autores referidos ilustram as suas propostas com exemplos conceptuais, mas sem nenhuma implementação concreta.

As analogias apresentadas indicam toda uma série de questões que correspondem a problemas que também se levantam nesses domínios, como quais os módulos a substituir e em que situações. Para algumas questões pode ser possível adoptar soluções análogas. Por exemplo, é importante minimizar o tempo de espera pela reconfiguração do módulo necessário. Carregar o módulo por antecipação *(pre-fetching)* tal como é feito em sistemas de memória virtual e em

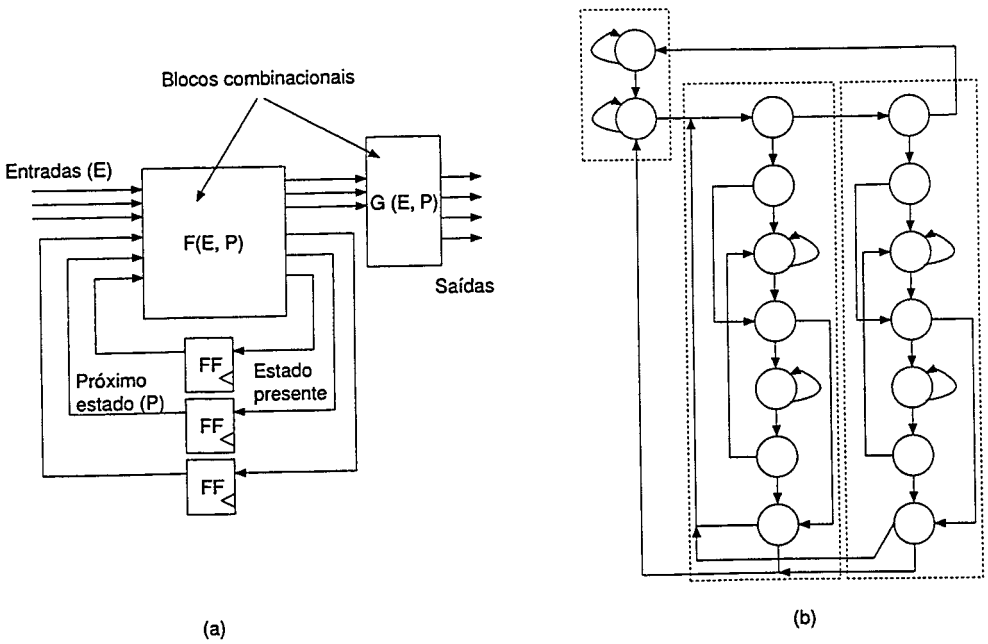


Figura 3.15: Divisão estrutural e funcional de uma máquina de estados finita.

memórias *cache* é uma hipótese [Hauck, 1998a].

Existem outras questões que não resultam directamente das analogias estabelecidas: Como coordenar a comunicação entre módulos que podem não estar simultaneamente presentes no substrato reconfigurável? Como tornar os módulos independentes da posição, por forma a facilitar a colocação dinâmica? Como preservar o estado de um módulo quando este é retirado do circuito FPGA? Todas estas questões necessitam de ser consideradas durante a concepção de um sistema de *hardware* virtual e de uma aplicação implementada em tal sistema.

Este tipo de sistemas levanta também a questão da decomposição do circuito virtual, isto é, como escolher os diferentes módulos. A abordagem mais natural é dividir a implementação ao longo de fronteiras funcionais, por forma a que cada módulo implemente uma funcionalidade bem definida. O caso particular de uma máquina de estados finita permite ilustrar estas considerações. Uma análise estrutural da implementação canónica permite distinguir entre parte combinacional e o registo de estado; uma análise mais fina pode levar a dividir a parte combinacional em duas, uma para determinar o próximo estado e outra para definir o valor das saídas (ver figura 3.15a). Em contrapartida, uma análise funcional permitiria obter uma decomposição em sub-máquinas (figura 3.15b), apenas uma das quais estaria residente no circuito FPGA num dado momento [DeHon, 1996b]. A decomposição seria escolhida

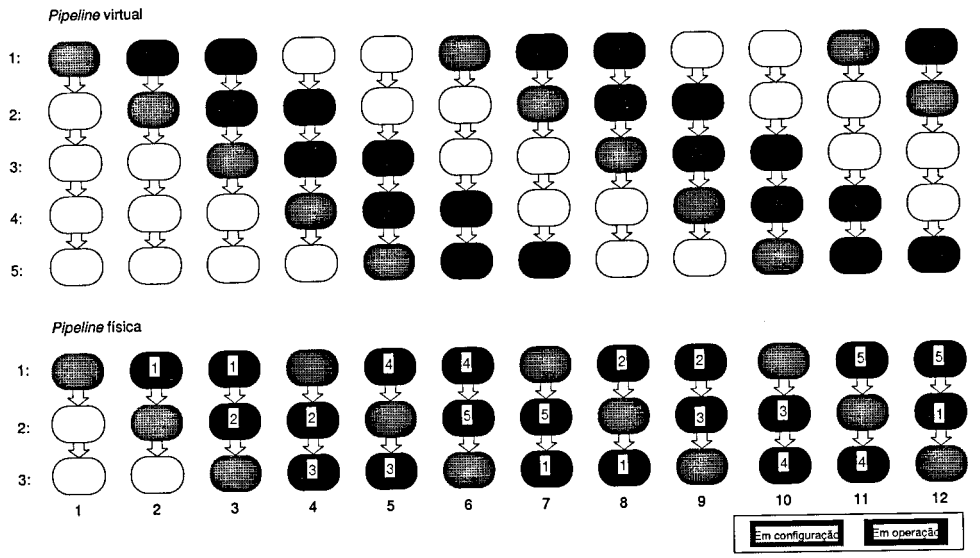


Figura 3.16: *Pipeline* virtual de 5 andares mapeada num circuito de 3 andares.

de forma a que existisse apenas um número reduzido de transições entre as diversas sub-máquinas. Poder-se-ia mesmo encarar um sistema mais complexo, composto por diversas máquinas de estados intercomunicantes, cada qual decomposta em sub-máquinas; em cada instante, apenas uma componente de cada máquina estaria activa. Neste cenário a estratégia de reconfiguração é mais sofisticada, porque pode ser necessário reconfigurar duas ou mais máquinas simultaneamente.

A implementação genérica do conceito de *hardware* virtual apresenta muitas dificuldades, pelo que os esforços referenciados na literatura técnica se têm centrado em versões mais restritas. Schmit [1997], por exemplo, apresenta uma abordagem aplicável a circuitos tipo *pipeline*. O objectivo principal é mapear um circuito de N andares num circuito FPGA especial com capacidade para P andares, com P<N. Para tal, os andares do circuito FPGA vão sendo reconfigurados quando necessário. Usando a analogia com sistemas de memória virtual referida anteriormente, podemos designar o circuito original como circuito virtual, do qual uma parte está realizada no circuito físico (o circuito FPGA) num dado instante. A figure 3.16 ilustra a situação em que um circuito virtual de cinco andares é implementado em três andares físicos. A parte de cima da figura mostra a evolução temporal do estado de cada andar do circuito virtual durante cinco ciclos seguidos; a parte inferior mostra o mapeamento correspondente para os três andares físicos. Neste arranjo, são produzidos dois resultados cada cinco ciclos. Conforme ilustrado, o circuito físico constitui uma "janela"que é deslocada sobre a parte activa

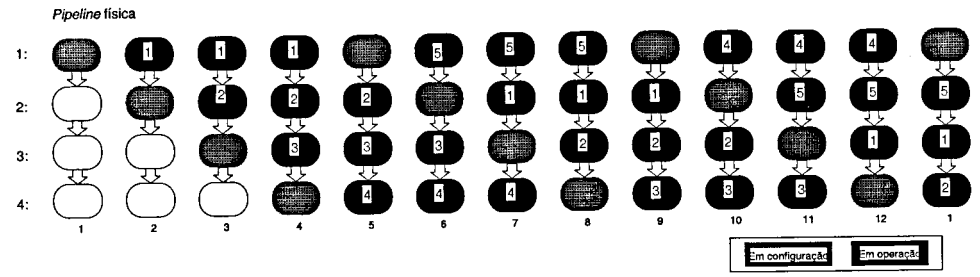


Figura 3.17: Escalonamento alternativo num sistema com quatro andares físicos.

do circuito virtual. Existem vários aspectos interessantes a ter em conta:

- A configuração de um andar é feita durante a execução dos outros andares, pelo que não limita o desempenho (embora implique que num substrato físico com N andares, apenas N-1 podem estar activos simultaneamente).
- A unidade de reconfiguração é o "andar"; a sua reconfiguração deve ser feita num ciclo de relógio.
- Qualquer andar virtual pode ser implementado em qualquer andar físico, pelo que entrada e saídas de dados devem poder ser feitas a partir de qualquer andar virtual. No exemplo da figura 3.16 os resultados do quinto andar virtual são produzidos pelo 1º andar real nos ciclos 6-7 e pelo 5º nos ciclos 11-12.
- Não podem existir dependências circulares entre andares virtuais, visto que tal levaria à necessidade de ter esses andares simultaneamente presentes no circuito físico.

O mapeamento do circuito virtual no suporte físico é feito em tempo de execução. Esta tarefa é levada a cabo por um controlador residente no circuito FPGA e depende do número de andares disponíveis. A figura 3.17 mostra o escalonamento que seria usado para o exemplo da figura 3.16 se existissem quatro andares físicos. Nesse caso, o sistema poderia produzir três resultados em cada cinco ciclos. Da mesma forma que um programa compilado pode, sem modificações, aproveitar melhorias do sistema em que é executado (por exemplo, um aumento da memória física), também o escalonamento em tempo de execução permite que uma única configuração seja adaptada dinamicamente aos recursos existentes. Outro aspecto importante diz respeito ao escalonamento dos dados; o controlador interno deve também fornecer/retirar os dados de entrada/saída dos andares correctos [Cadambi *et al.*, 1998].

Esta abordagem foi concretizada num protótipo designado por PipeRench [Cadambi *et al.*, 1998; Moe *et al.*, 1998], já que os circuitos FPGA convencionais não têm a arquitectura apropriada, nomeadamente uma organização em andares reconfiguráveis individualmente. A fi-

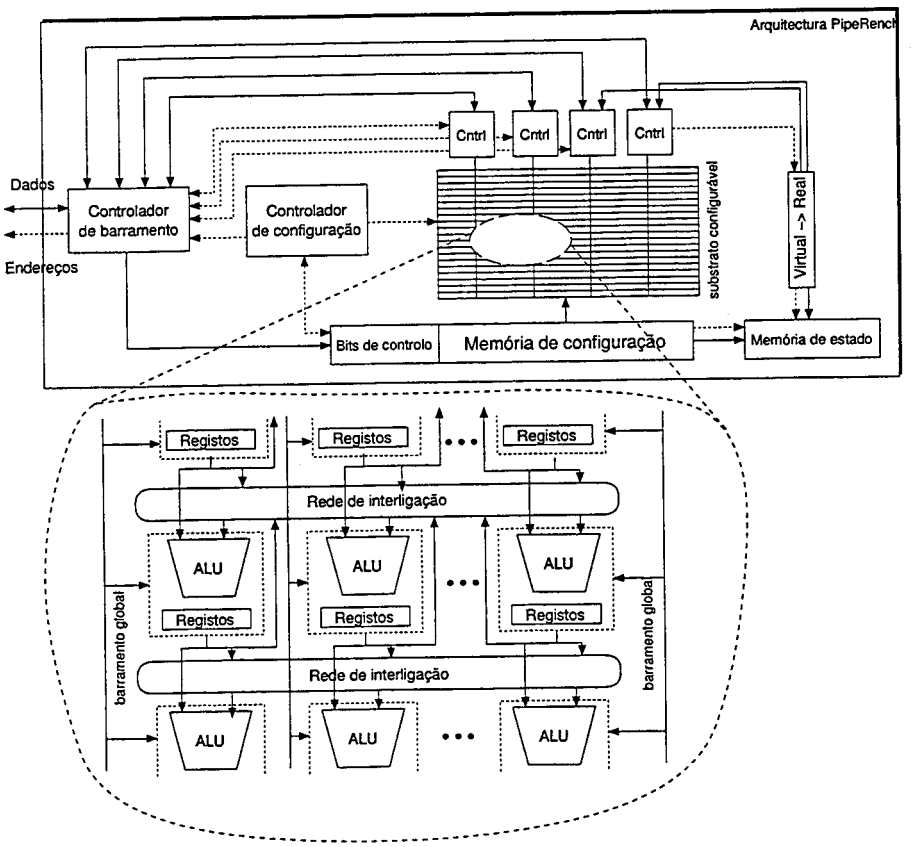


Figura 3.18: O protótipo PipeRench.

gura 3.18 apresenta uma versão muito simplificada do protótipo. Para além de uma secção reconfigurável composta por andares combinacionais com um elemento de memória, o circuito inclui uma memória estática para guardar as configurações, outra memória para guardar o estado de andares não activos, e ainda os controladores de configurações e fluxo de dados. A memória de configurações é muito larga, já que uma única palavra deve configurar um andar completo (para que a configuração possa ser feita num ciclo de relógio). Cada andar é composto por um conjunto de elementos de processamento, que são essencialmente unidades lógico-aritméticas configuráveis [Goldstein *et al.*, 1999]. Existem ainda quatro barramentos globais: dois são usados para guardar e restaurar o estado de um andar (i.e. o conteúdo dos seus elementos de memória) e os outros dois são usados para entrada e saída de valores (operandos e resultados). Um protótipo da mesma arquitectura, adaptado ao barramento PCI, é descrito por Laufer *et al.* [1999].

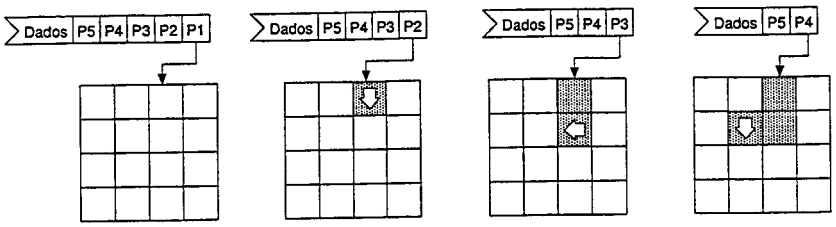


Figura 3.19: Configuração de um substrato reconfigurável por uma sequência *(stream)*.

Uma estratégia diferente é usada no sistema Colt [Bittner e Athanas, 1997]. Aqui trata-se de criar e modificar rapidamente percursos computacionais *(computational pathways)* através de uma infra-estrutura programável, usando um esquema de controlo distribuído baseado em sequências *(streams)*. No contexto desta técnica (designada por *wormhole routing*) uma sequência é o resultado da concatenação de um cabeçalho de configuração e um conjunto de dados (ver figura 3.19). O cabeçalho contém informação para configurar um percurso computacional, bem como as operações efectuadas pelos diversos elementos computacionais existentes ao longo desse percurso. À medida que a sequência progride através do sistema, a informação de configuração é retirada do cabeçalho (cujo comprimento vai assim diminuindo) e usada para configurar o próximo elemento do percurso. O cabeçalho é composto por um número arbitrário de pacotes com informação de configuração.

O protótipo de um circuito reconfigurável especialmente vocacionado para suportar esta técnica é descrito por Bittner [1997]; a sua arquitectura encontra-se ilustrada na figura 3.20. Colt tem 6 portos bidireccionais de 16 bits, uma malha 4×4 de unidades funcionais e um multiplicador de 16 bits (com resultado de 32 bits). Todos os componentes estão interligados por uma matriz de encaminhamento. Este circuito pode ser usado individualmente ou como componente de um sistema heterogéneo composto por vários subsistemas configuráveis pela mesma técnica. Cada unidade funcional aceita dois operandos de 16 bits (dos quais um passa por um circuito de deslocamento do tipo *barrel shifter*), que alimentam uma unidade lógico-aritmética configurável. Esta arquitectura está vocacionada para o tratamento de informação por palavras, em contraste com o processamento bit a bit dos circuitos FPGA convencionais.

O percurso exacto tomado pela sequência de configuração é determinado em tempo de execução, o que permite a diversos processos concorrentes "competir"pelos recursos comuns. No cenário descrito por Bittner [1997], essa função de alocação de recursos seria exercida por controladores especiais que arbitrariam o acesso aos vários recursos e modificariam os parâmetros do cabeçalho de forma apropriada. Efectuar a reconfiguração de parte do sistema é tão simples como "lançar" uma nova sequência para o substrato configurável.

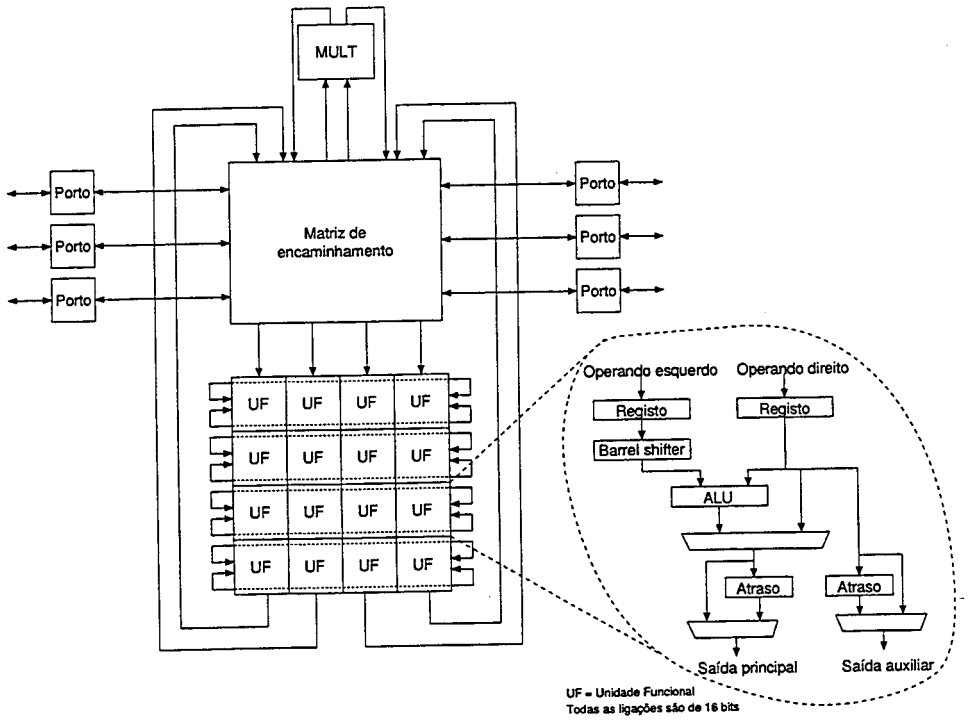


Figura 3.20: Arquitectura do protótipo Colt.

3.3.4 Sequenciamento dinâmico de unidades funcionais configuráveis

Outro cenário de sequenciamento dinâmico de configurações pré-determinadas envolve híbridos microprocessador/FPGA. Nesta área existem alguns sistemas propostos e implementados, variando em muitos aspectos da sua arquitectura interna, nomeadamente na interface processador/FPGA. Podemos distinguir dois grandes grupos. O primeiro grupo usa o módulo reconfigurável como uma unidade funcional interna (a par das outras, como por exemplo a unidades lógico-aritmética ou de vírgula flutuante), invocada por instruções específicas. A utilização de unidades integradas no percurso de dados (*datapath*) permite a utilização de recursos configuráveis em cálculos muito curtos (apenas um ciclo de relógio). Pertencem a este grupo os sistemas PRISC [Razdan, 1994], DISC [Wirthlin e Hutchings, 1995] e OneChip [Wittig e Chow, 1996].

O segundo grupo acede aos recursos reconfiguráveis como a uma memória interna, através de instruções explícitas de leitura/escrita. Com esta organização a comunicação é ligeiramente mais pesada, mas permite a operação concorrente do processador e da matriz configurável, com esta última a executar tarefas de maior duração (vários ciclos de relógio). Pertencem a

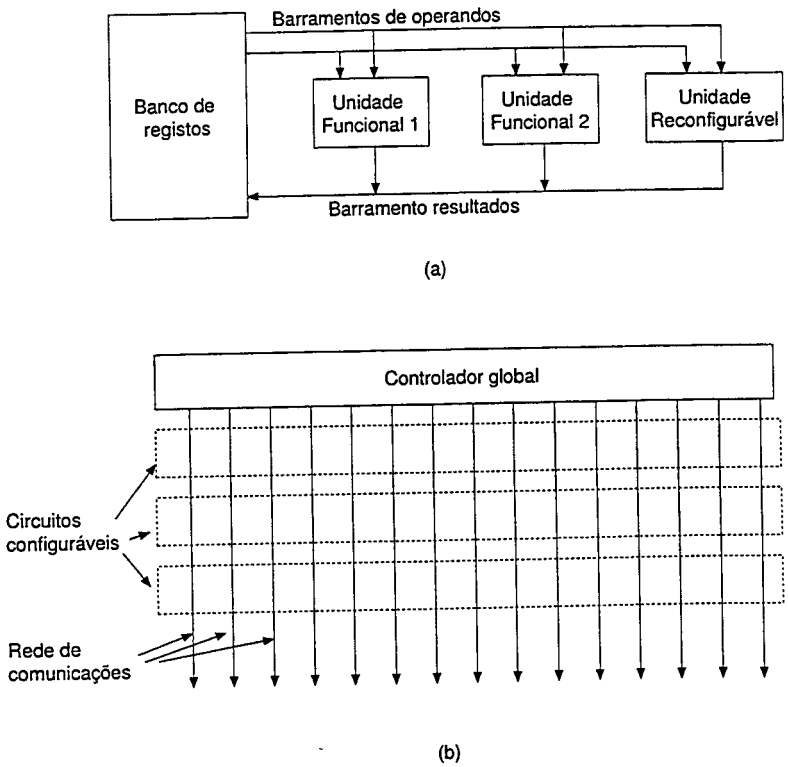


Figura 3.21: Organização das unidades reconfiguráveis incluídas em sistemas híbridos processador/FPGA.

este segundo grupo os sistemas FIPSOC [Faura *et al.*, 1997a] e Triscend E5 [Triscend].

A figura 3.21 apresenta as duas organizações mais usadas com unidades funcionais incluídas no percurso de dados. Na primeira, as unidades são colocadas em paralelo com unidades de funcionalidade fixa. A unidade é alimentada pelo mesmos mecanismos que fornecem operandos às outras unidades. Os sistemas PRISC e OneChip seguem esta organização. A alternativa consiste em ter uma área configurável, organizada por linhas, em que são colocados os módulos desejados. A área é percorrida por barramentos verticais que estabelecem a comunicação com o núcleo do processador. Este arranjo é usado nos protótipos GARP [Hauser e Wawrzynek, 1997] e DISC.

As configurações são geralmente pré-calculadas (construídas manualmente ou automaticamente durante o processo de compilação) mas a sua activação é determinada pelo fluxo de execução.Os circuitos incluem geralmente mecanismos de apoio à gestão de configurações. No caso de DISC, o sistema detecta a ausência do módulo que implementa uma instrução

específica e carrega-o quando necessário. O processador GARP admite apenas uma configuração activa mas mantém um repositório das utilizadas mais recentemente; naturalmente que a activação de uma configuração residente é muito mais rápida que o seu carregamento a partir da memória principal. No sistema PRISC a ausência de um módulo necessário provoca uma interrupção cuja rotina de atendimento é responsável pela reconfiguração apropriada.

3.3.5 Criação dinâmica de configurações

O passo seguinte no aproveitamento da reconfiguração dinâmica consiste em ir criando as próprias configurações durante a execução da aplicação. A criação dinâmica de configurações permite adaptá-las especificamente a cada invocação do sistema, mesmo que as características relevantes não sejam conhecidas ou determináveis *a priori*. Por exemplo, French e Taylor [1993] propõem um híbrido processador/FPGA em que a implementação de instruções vai sendo criada durante a execução do programa (ver página 62). De acordo com a classificação anterior, trata-se de um sistema do tipo IV em que configuração e sequenciamento podem variar simultaneamente. Existem muito poucos sistemas deste tipo (mesmo o sistema referido atrás foi apenas descrito conceptualmente), provavelmente quer por dificuldades de implementação quer por ausência de um modelo de execução que sirva para orientar a investigação. Na falta de um modelo orientador, a generalidade desta categoria dificulta a sua exploração. Ademais, as vantagens em relação a outros tipos de sistemas de reconfiguração dinâmica, especialmente de tipo II, não são claras: a complexidade adicional pode não corresponder a ganhos proporcionais.

São mais numerosos os exemplos de sistemas do tipo III (sequenciamento fixo de configurações variáveis), embora também menos frequentes que os de tipo II. Os principais exemplos de alteração de configurações encontram-se em sistema adaptativos, que vão alterando o seu comportamento por reacção a condições exteriores; a alteração de comportamento resulta directamente de uma alteração da configuração. As arquitecturas deste tipo ajustam-se preferencialmente a aplicações em que exista um tratamento periódico de informação (sequenciamento fixo no tempo) que vai variando ao longo da execução (variação da configuração). Um exemplo proveniente da área de processamento de sinal é descrito por Woolfries *et al.* [1998], e já foi referido anteriormente (página 61). Lysaght e Dick [1994] descrevem um sistema de detecção de tom de voz em que o número de amostras é variado dinamicamente, para se adaptar ao tom de diferentes falantes da maneira mais eficiente, bem como às variações de tom de um mesmo falante. Essa variação reflecte-se na dimensão de vários registos de deslocamento usados em operações de autocorrelação. No sistema descrito, é feita uma estimativa do tom em cada 10 ms. Também é mantida uma estimativa do número de amostras necessário, revista periodicamente. Cada revisão leva a uma reconfiguração do circuito FPGA Atmel AT6005 usado para implementar o sistema.

Outra área com características favoráveis a este tipo de abordagens é a dos autómatos ce-

lulares. Moreno *et al.* [1998] descrevem uma aplicação do processador FIPSOC à evolução *on-line* de autómatos celulares não-uniformes (isto é, aqueles em que cada célula pode ter regras diferentes). A actividade de um sistema deste tipo tem duas fases:

Execução As células interagem localmente e actualizam o seu estado em função do seu estado anterior e do estados dos seus vizinhos. As regras de avaliação do próximo estado não se alteram.

Evolução Cada célula é avaliada por uma função que mede a sua aptidão *(fitness)* e a regra de avaliação do próximo estado é actualizada de uma forma que depende do número de vizinhos mais aptos.

Na implementação de Moreno *et al.* [1998] para o processador FIPSOC o sistema alterna entre dois contextos correspondentes às duas fases. O segundo contexto (evolução) altera a futura configuração (contexto inactivo) para modificar as regras de cada célula. A alternância entre as duas fases é controlada por um programa do microprocessador.

Os trabalhos descritos nesta secção representam primeiras tentativas de explorar a geração dinâmica de configurações. Apesar de os resultados serem ainda preliminares e pouco sistemáticos, constituem avanços que podem vir a estender-se a outras áreas.

3.4 Projecto assistido por computador

Como área de trabalho recente que são, os SDRD contam ainda com poucas ferramentas de apoio específicas, e as que existem são de difícil utilização. O impacto comercial ainda reduzido e o carácter exploratório das actividades justificam esta situação, que por sua vez impede uma divulgação maior deste tipo de sistemas. As ferramentas existentes estão estreitamente associadas à arquitectura e constituição dos sistemas que apoiam, não se tendo ainda chegado a ferramentas de aplicação genérica.

O desenvolvimento dos SDRD pode ser feito sem apoio específico, usando-se apenas as ferramentas tradicionalmente disponíveis. Por exemplo, os dados para cada configuração podem ser produzidos individualmente com ferramentas convencionais e posteriormente combinados por métodos *ad hoc* para obter o comportamento dinâmico pretendido. Este tipo de processo de desenvolvimento é pouco eficiente, sujeito a erros e de produtividade muito baixa, sendo apenas aceitável para efeitos de demonstração de conceitos. Concepção de sistemas, exploração de alternativas e depuração são, nestas condições, tarefas penosas e demoradas.

A divulgação dos SDRD e a exploração efectiva das suas possibilidades depende pois de melhores apoios à concepção, desenvolvimento e depuração, seja por adaptação de ferramentas existentes (por exemplo, simuladores) ou por criação de novas ferramentas capazes de lidar com os problemas específicos das novas abordagens (por exemplo, o sequenciamento automático). Também existem muitas questões em aberto ao nível das técnicas de especificação e

dos sistemas de apoio à execução (sistemas de *runtime*).

O ambiente de aplicação mais estruturado é o dos híbridos processador/FPGA, onde a compilação a partir de uma linguagem de alto nível é a alternativa mais natural. Esta secção começa precisamente por tratar destes sistemas; seguidamente passa aos SDRD construídos em torno de circuitos FPGA, tratando sucessivamente de técnicas de especificação e validação, identificação de módulos dinâmicos, sistemas de apoio à execução, ferramentas de síntese de alto nível e algoritmos de decomposição temporal.

3.4.1 Compilação de linguagens de alto nível

A utilização de linguagens de alto nível, geralmente C, constitui a forma mais fácil de especificar computações para circuitos FPGA porque permite aproveitar a base de conhecimento e de código já existente. No caso dos híbridos processador/FPGA é mesmo a alternativa principal.

A principal vantagem desta abordagem é a simplicidade decorrente da utilização de uma única linguagem para toda a aplicação, tanto para a parte que executa no processador como para a que é mapeada no circuito FPGA. Contudo, a natureza sequencial das linguagens imperativas convencionais transforma num grande desafio o aproveitamento do paralelismo inerente de uma realização em *hardware*. Uma solução consiste em aumentar linguagens convencionais com meios para expressar esse paralelismo latente [Gokhale e Minnich, 1993; Guccione e Gonzalez, 1993] ou usar linguagens alternativas com suporte natural para operações concorrentes. Quanto maior é o desvio semântico destas linguagens em relação a linguagens sequenciais, mais difícil se torna encontrar programadores capazes de as usarem eficientemente.

Uma segunda alternativa é limitar a utilização de *hardware* através do uso dos mecanismos de abstracção que a linguagem disponibilize. O conjunto de ferramentas elaboradas para o sistema DISC [Clark e Hutchings, 1996] é um exemplo desta abordagem. O sistema inclui um compilador, um "assemblador" e um depurador. A aplicação é escrita em linguagem C com chamadas a módulos de *hardware* pré-definidos usando o mecanismo convencional de invocação de funções de acordo com algumas convenções adicionais. O compilador transforma essas chamadas em activações dos módulos apropriados. Neste ambiente, o desenvolvimento de uma nova aplicação consiste em repartir a funcionalidade entre *hardware* e *software*, desenvolver os módulos de *hardware*, escrever o código e fazer a depuração conjunta. O objectivo do sistema de desenvolvimento não é acelerar programas já existentes, mas proporcionar um ambiente tradicional para aplicações de reconfiguração dinâmica. Os módulos de *hardware* são projectados usando ferramentas tradicionais, o que exige utilizadores com conhecimentos específicos de projecto de sistemas digitais.

Uma última alternativa é procurar, por análise da interdependência das instruções, extrair automaticamente o paralelismo a partir de descrições sequenciais e, seguidamente, gerar au-

tomaticamente as configurações apropriadas. O projecto BRASS desenvolveu um compilador de linguagem C com estas características para o processador reconfigurável GARP [Callahan e Wawrzynek, 1998]. O compilador selecciona os ciclos interiores do programa para implementação na unidade reconfigurável. A estratégia consiste em identificar grupos de instruções, construindo hiper-blocos a partir de blocos básicos contíguos que sejam constituídos apenas por operações implementáveis em *hardware*. Blocos básicos cuja frequência de execução estimada seja baixa são excluídos do hiperbloco, bem como aqueles que requeiram demasiados recursos ou tenham um impacto adverso sobre o caminho crítico. Cada hiperbloco assim construído tem um único ponto de entrada, embora possa ter vários pontos de saída. Para cada hiperbloco é criada uma configuração por um sintetizador de *hardware* para a arquitectura GARP. O circuito gerado inclui o percurso de tratamento de dados *(datapath)* e um sequenciador de operações. O compilador também gera as instruções necessárias para configurar e activar a unidade reconfigurável. Sempre que a execução do programa atinge um ciclo interior, o processador carrega a configuração correcta, inicializa os registos do circuito FPGA e activa a unidade reconfigurável, ficando suspenso até esta terminar as suas operações. Resultados preliminares [Callahan e Wawrzynek, 1998] indicam que este método tem boa capacidade de extrair e explorar o paralelismo latente em programas sequenciais.

3.4.2 Modelação, especificação e validação

Os sistemas híbridos microprocessador/FPGA são uma pequena parte dos SDRD; a grande maioria é constituída por circuitos FPGA com suporte para reconfiguração dinâmica, possivelmente com circuitos adicionais capazes de efectuarem o controlo das reconfigurações (microprocessadores ou outros circuitos FPGA). Como a infra-estrutura usada na segunda parte deste trabalho também pertence a esta categoria, as restantes secções deste capítulo debruçam-se sobre diversos aspectos da concepção e desenvolvimento de aplicações para estes ambientes.

De uma maneira geral, é possível desenvolver aplicações para SDRD com ferramentas convencionais de apoio ao projecto: os circuitos correspondentes a cada etapa temporal são determinados por análise *ad hoc* da aplicação, procedendo-se à sua implementação individual; separadamente, procede-se à especificação e implementação (em *hardware* ou *software*) do mecanismo de controlo de configurações; procede-se enfim à reunião dos diversos elementos e à sua validação conjunta, possivelmente por execução directa sobre a infra-estrutura de *hardware*.

Esta estratégia de desenvolvimento tem muitas fragilidades quando aplicada a sistemas de reconfiguração dinâmica, resultando num ciclo de desenvolvimento penoso e pouco eficiente, que não contribui para a divulgação e exploração desta nova abordagem. Tudo isto torna importante desenvolver ferramentas de apoio às diversa fases do projecto de SDRD, e as numerosa tentativas de responder a estes desafios específicos conduziram a resultados interessantes

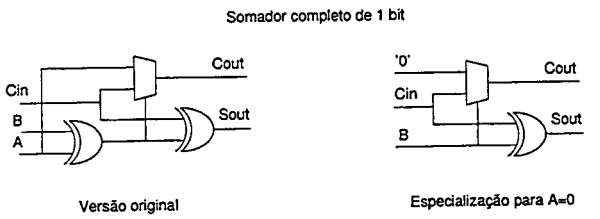


Figura 3.22: Exemplo de avaliação parcial.

em diversas frentes.

Uma primeira questão que se coloca é a maneira de modelar e especificar a reconfiguração dinâmica em linguagens de descrição de *hardware* (LDH). Uma LDH convencional como VHDL não permite especificar de forma natural oportunidades de empregar reconfiguração dinâmica, porque apenas pode representar informação estrutural estática [Singh *et al.*, 1996; Hogg *et al.*, 1996]; consequentemente, muita desta actividade centra-se em linguagens alternativas como Lava [Bjesse *et al.*, 1998], Ruby [Guo e Luk, 1995] e Pebble [Luk e McKeever, 1998].

Uma maneira implícita de modelar reconfiguração dinâmica é em termos de avaliação parcial [Singh *et al.*, 1996]. Trata-se de uma técnica que permite tirar partido do facto de os sinais de entrada permanecerem estáticos por períodos de tempo longos. O seu princípio básico reside na propagação de parâmetros constantes através da especificação para se obterem versões especializadas.

Como exemplo considere-se a seguinte definição de um somador completo em VHDL (o esquema correspondente está ilustrado na figura 3.22):

```
entity somador is
port (a, b, cin: in;
      s, cout: out bit);
end somador;

architecture structural of somador is
     signal d, e: bit;
begin
     xor_1: port map XOR2 (a, b, d);
     xor_2: port map XOR2 (cin, d, s);
     mux:   port map MUX2 (d, a, c, cout);
end structural;
```

A avaliação parcial deste circuito para o valor de entrada a=0 permite obter uma versão

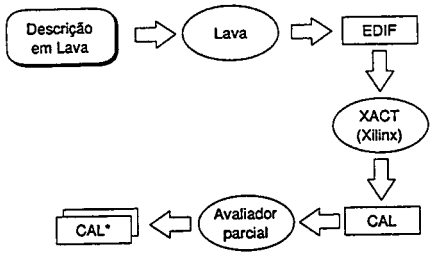


Figura 3.23: Organização das tarefas de reconfiguração dinâmica segundo Susanto e Melham [1998].

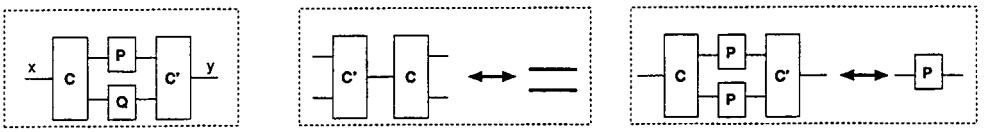
especializada, equivalente à seguinte definição:

```
entity somadorAvP is
port (b, cin: in;
      s, cout: out bit);
end somadorAvP;

architecture structural of somadorAvP is
     signal d, e: bit;
begin
     xor_1: port map XOR2 (b,c, s);
     mux:   port map MUX2 (b, '0', c, cout);
end structural;
```

De notar que, num SDRD, especialização pode ser feita em tempo de execução, não apenas em tempo de compilação. A figura 3.23 apresenta a organização das tarefas no sistema de Susanto e Melham [1998] que implementa esta abordagem. Após a inicialização do circuito FPGA com a versão genérica do módulo, o sistema entra em modo de execução. Quando o programa de avaliação parcial detecta o valor da entrada "estática", gera uma versão especializada do módulo, gerando imediatamente a nova configuração, que vai substituir a anterior no circuito FPGA. Nesta fase, o circuito fica a executar uma versão simplificada e, por isso, mais rápida do módulo original. Singh *et al.* [1996] discutem a aplicação destes princípios ao gestor de filas de um comutador ATM. Um outro avaliador parcial é descrito por McKay e Singh [1998] e aplicado a multiplicadores paralelos e filtros digitais do tipo FIR.

Esta técnica tem aplicação natural a todos os circuitos que tenham algumas entradas de variação muito mais lenta que outras. Contudo, o seu emprego apenas se justifica quando contribuir para uma aceleração global das operações. Os factores que afectam essa aceleração podem ser analisados considerando um cenário em que uma sequência de n itens de



(a) modelo básico (b) propriedade 1 (c) propriedade 2

Figura 3.24: Propriedades do modelo de um SDRD segundo Luk *et al.* [1996].

informação é processada entre reconfigurações. Neste caso temos:

T_s Tempo necessário para sintetizar o circuito especializado;

T_p Tempo de reconfiguração do circuito FPGA;

T_c Período do ciclo de relógio para a versão especializada;

T_g Período do ciclo de relógio para a versão genérica;

T_k Tempo necessário para carregar parâmetro de especialização (supondo que este deve ser mantido num registo durante a execução da versão genérica).

O factor de aceleração obtido pela utilização de uma versão especializada vem então dado por

$$A = \frac{T_k + nT_g}{T_s + T_p + nT_c}$$

A versão especializada é melhor sempre que $A > 1$.

A geração dinâmica de circuitos especializados coloca problemas sérios de verificação e validação, ainda não resolvidos de forma satisfatória. Em geral, não é possível validar os circuitos assim gerados por simulação. A alternativa mais natural é fazer a verificação formal do algoritmo de especialização, com o intuito de garantir que este opera correctamente em todos os casos. Os primeiros passos nesta direcção foram dados por Susanto e Melham [1998], que descrevem a maneira de especificar formalmente as características estruturais de configurações de circuitos FPGA XC6200 bem como as modificações a que estas podem ser submetidas. Contudo, falta ainda mostrar a correcção do algoritmo de especialização bem como da sua implementação.

Avaliação parcial fornece um contexto possível para analisar e descrever alguns aspectos da reconfiguração dinâmica, mas não permite especificar SDRD arbitrários, nem investigar o respectivo comportamento temporal. O modelo de Luk *et al.* [1996], embora de um nível de abstracção mais baixo, permite especificar directamente a exclusão mútua no tempo de dois módulos arbitrários; situações mais complexas são modeladas por aplicação sucessiva dos

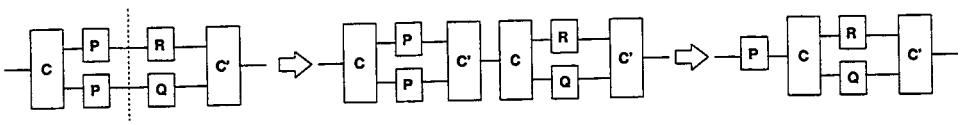


Figura 3.25: Simplificação de blocos alternativos com sub-blocos comuns.

mesmos princípios. Segundo Luk *et al.* [1996], dois módulos P e Q, mutuamente exclusivos no tempo, são modelados por uma rede estática, em que o paralelo de P e Q fica colocado entre dois blocos de controlo, C e C' (figura 3.24). O conjunto global comporta-se como P ou Q, dependendo dos blocos de controlo; a escolha pode ser feita em tempo de execução por influências de entradas adicionais (não ilustradas na figura) dos blocos C e C'. Os blocos C e C' representam entidades abstractas que não correspondem necessariamente a módulos físicos; destinam-se apenas a modelar a existência de alternativas estruturais. Por exemplo, os módulos de controlo podem ser implementados por pares desmultiplexador/multiplexador. Alternativamente, os blocos de controlo podem não ter uma implementação física, se o sistema usar reconfiguração dinâmica para transformar P em Q e vice-versa; neste caso, os sinais de controlo dos blocos C e C' podem ser usados para orientar a configuração (i.e., implementar um controlador de reconfiguração).

Os blocos C e C' satisfazem duas propriedades simples e intuitivas:

1. A ligação em série de C e C' comporta-se como uma par de pistas das quais apenas uma está activa em cada instante (figura 3.24b);

2. Se P e Q forem idênticos, todo a rede é equivalente a P (figura 3.24c).

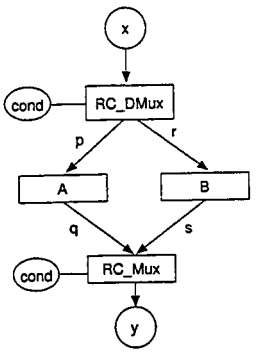
Estas propriedades estão na base de um método de refinamentos sucessivos para reduzir a granularidade dos elementos a reconfigurar e, portanto, minimizar o tempo de reconfiguração. Este método, ilustrado na figura 3.25, tem três passos a serem aplicados iterativamente:

1. Identificar blocos comuns em configurações consecutivas;

2. Inserir um par de blocos de controlo C e C' para isolar os blocos comuns (usando a propriedade 1);

3. Reunir os blocos comuns num só bloco (usando a propriedade 2).

Este modelo é expresso na linguagem Ruby por Luk *et al.* [1996], que apresentam uma aplicação sua à detecção de arestas em imagens. O modelo também é usado numa ferramenta automática de apoio ao desenvolvimento de bibliotecas de componentes reconfiguráveis, que inclui a capacidade de visualizar e simular os diferentes módulos [Luk e Guo, 1998]




```
BLOCK AB [x, cond: WIRE] [y:WIRE]
BEGIN
    RECONFIGURE IF cond THEN
        A [x] [y] END
    ELSE
        B [x] [y] END
END;
```

Figura 3.26: Reconfiguração dinâmica em Pebble.

A linguagem de descrição de *hardware* Pebble [Luk e McKeever, 1998] foi desenvolvida para facilitar a criação de projectos eficientes e re-utilizáveis, e para apoiar o desenvolvimento de aplicações de reconfiguração dinâmica. Trata-se de uma variante simplificada de VHDL estrutural capaz de representar diagramas de blocos hierárquicos e parametrizáveis. A reconfiguração dinâmica pode ser expressa por componentes especiais correspondentes aos blocos C e C' da descrição anterior ou através da instrução `RECONFIGURE-IF`. O exemplo da figura 3.26 ilustra a utilização desta instrução. Os dois ramos da instrução especificam os módulos mutuamente exclusivos no tempo; em cada instante, o valor da condição determina qual é o módulo activado.

O método de especificação utilizado em Pebble ilustra bem o facto de que, a um nível comportamental *(behavioural)*, a reconfiguração dinâmica pode ser representada por uma instrução condicional

```
IF <cond> THEN <bloco1> ELSE <BLOCO2>
```

já que os módulos correspondentes aos ramos exclusivos não precisam de estar simultaneamente presentes. Mais ainda, a sua activação é univocamente determinada pela condição de controlo `<cond>`, que pode ser usada para implementar o controlador de reconfiguração.

A alternativa ao uso de uma LDH é o emprego de linguagens de programação convencionais para a descrição de circuitos, como acontece no sistema DECPeRLe-1 [Vuillemin *et al.*, 1996]. A linguagem JHDL *(Just another HDL)* e respectivo sistema de desenvolvimento usam a linguagem Java para expressar a organização de circuitos que se alteram ao longo do tempo. Entre seus objectivos contam-se [Bellows e Hutchings, 1998]:

1. Utilização de uma linguagem convencional, sem extensões, de forma a tornar a ferramenta facilmente acessível;

2. O paradigma de controlo do SDRD é independente da infra-estrutura digital, para garantir um nível de abstracção elevado e a consequente facilidade de programação e reutilização;

3. O método de descrição suporta reconfiguração dinâmica e parcial;

4. A descrição deve servir, sem modificações, tanto para simulação completa (incluindo múltiplas configurações) como para execução no SDRD.

JHDL constitui um meio de especificação manual que combina o controlo do SDR e a definição do circuito numa única descrição integrada. A reconfiguração dinâmica deve ser explicitamente referida. Para tal, JHDL usa o mecanismo de construção/destruição de instâncias adoptado em muitas linguagens de programação orientada aos objectos. Neste mecanismo, a memória de um objecto é alocado por uma função especial, o construtor, que é invocado sempre que se cria um novo objecto; quando o objecto não é mais necessário, a sua memória é libertada por outra função especial, o destrutor. JHDL usa um mecanismo similar para gerir os recursos do SDRD. Componentes do circuito correspondem a objectos de classes especiais: quando um desses objectos é criado, o seu construtor configura os circuitos FPGA apropriadamente; quando o objecto é removido, o seu destrutor liberta a respectiva área do circuito FPGA.

Quando o programa que descreve o circuito é executado em modo de simulação, os construtores/destrutores comunicam com um núcleo de simulação. Os objectos criados são modelos de simulação que, depois de ligados ao núcleo de simulação, permitem efectuar a simulação ciclo-a-ciclo do circuito. Em modo de execução no SDRD, os construtores/destrutores comunicam directamente com o SDRD, configurando/desconfigurando os circuitos FPGA apropriadamente.

JHDL suporta reconfiguração parcial através de uma interface especial, que deve ser implementada por todos os objectos que representam circuitos mutáveis. Neste caso, os objectos mantêm informação sobre as múltiplas configurações possíveis. A configuração activa é determinada por invocação de um método especial. Reconfiguração parcial é suportada tanto em modo de simulação como de execução.

Bellows e Hutchings [1998] descrevem alguns exemplos de utilização de JHDL ao processamento de imagens. Uma versão mais recente do sistema descrita por Hutchings *et al.* [1999] inclui um depurador visual de circuitos, um gerador de esquemáticos, um gerador de descrições em EDIF e uma ferramenta visual de posicionamento de módulos nos circuitos FPGA.

A validação de aplicações de reconfiguração dinâmica é simultaneamente mais difícil e mais necessária que nos sistemas digitais convencionais. A complexidade acrescida que resulta da evolução temporal da estrutura do circuito é a razão principal desta situação. Não obstante alguns esforços de verificação formal já referidos, na grande maioria dos casos a validação é feita por simulação. Por exemplo, no caso de JHDL é posto grande valor na pos-

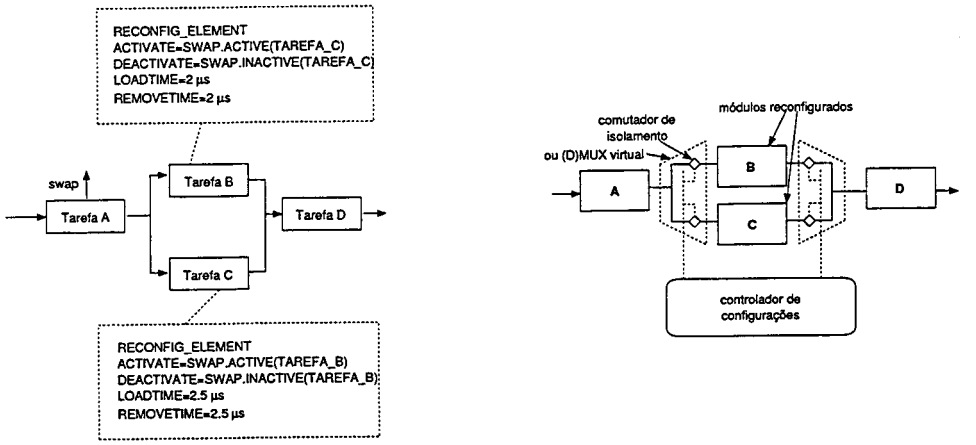


Figura 3.27: Exemplo de especificação de reconfiguração dinâmica no sistema DCS.

sibilidade de simular os circuitos ciclo-a-ciclo. Em muitas casos, uma simulação ainda mais detalhada pode ser desejável. Em particular, é importante simular a operação do circuito durante a fase de reconfiguração para se poder assegurar que a parte não reconfigurada funciona correctamente durante estes períodos e que não existe contenção de sinais nas interfaces entre módulos.

Uma possibilidade teórica é simular completamente todo o circuito FPGA, incluindo o seu mecanismo de configuração e respectiva memória. Para além de a informação necessária não se encontrar disponível, é fácil de ver que, tendo em conta que os maiores circuitos FPGA contém actualmente dezenas de milhões de transístores, tal abordagem é praticamente impossível. Em alternativa é preciso ter modelos mais simples que permitam obter a informação pretendida sem recorrer a tais extremos.

A técnica DCS *(Dynamic Circuit Switching)* de Lysaght e Stockwood [1996] é um dos primeiros métodos descritos que permite aplicar simuladores convencionais (neste caso, um simulador VHDL) a sistemas de reconfiguração dinâmica. Durante a captura esquemática são identificados os elementos reconfiguráveis e encapsulados em símbolos hierárquicos, aos quais são adicionados atributos textuais que exprimem o sequenciamento das configurações (ver o exemplo da figura 3.27). O sistema de Lysaght e Stockwood [1996] usa os seguintes atributos:

RECONFIG_ELEMENT Serve para marcar um subcircuito como sendo dinamicamente reconfigurável;

ACTIVATE Especifica a condição em que o elemento é carregado para o circuito FPGA. No exemplo da figura 3.27 o módulo B é activado no flanco ascendente do sinal `swap` (por

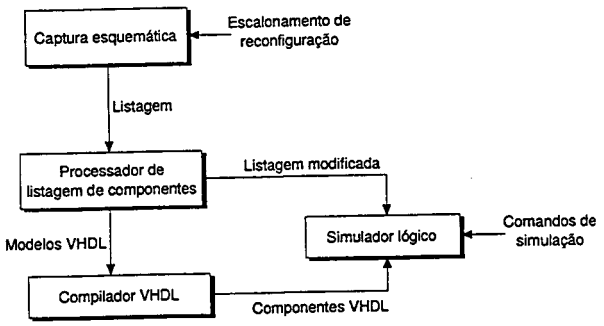


Figura 3.28: Fluxo das actividades de desenvolvimento no sistema DCS.

reconfiguração a partir do bloco C);

DEACTIVATE Especifica a condição em que o bloco é substituído;

LOADTIME Indica o tempo que demora a carregar este subcircuito;

REMOVETIME Indica o tempo necessário para re-inicializar a secção do circuito FPGA usada por este módulo.

Este método de simulação baseia-se na premissa de que é possível simular o efeito da reconfiguração dinâmica por comutação apropriada dos blocos (figura 3.27b): apenas o bloco activo é integrado dinamicamente no circuito enquanto os outro permanecem isolados. Durante a comutação, todos os blocos estão isolados, pelo que todos os sinais que daí emanam são mantidos ao nível 'X', o que permite verificar se as operações de reconfiguração afectam adversamente o restante circuito (observando se existe propagação indevida deste valor ao restante circuito). O método DCS é implementado por um andar adicional de processamento que ocorre entre a captura esquemática e a simulação (figura 3.28). A ferramenta gera uma descrição modificada do circuito, usada exclusivamente para simulação, em que são incluídos vários componentes (descritos em VHDL comportamental) para simularem as operações de reconfiguração.

O método DCS foi melhorado e integrado num ambiente de desenvolvimento mais completo por Robinson *et al.* [1998]. O novo sistema centraliza a descrição de actividades de reconfiguração num só ficheiro, suporta múltiplos níveis na hierarquia de projecto e permite especificar grupos de módulos mutuamente exclusivos no tempo (para verificação durante a simulação).

Para além do simulador, DCSim, o ambiente de desenvolvimento (figura 3.29) inclui mais duas ferramentas especializadas para reconfiguração dinâmica. DCSTech gere a interface com as ferramentas de colocação e encaminhamento: por um lado é responsável pela produção de

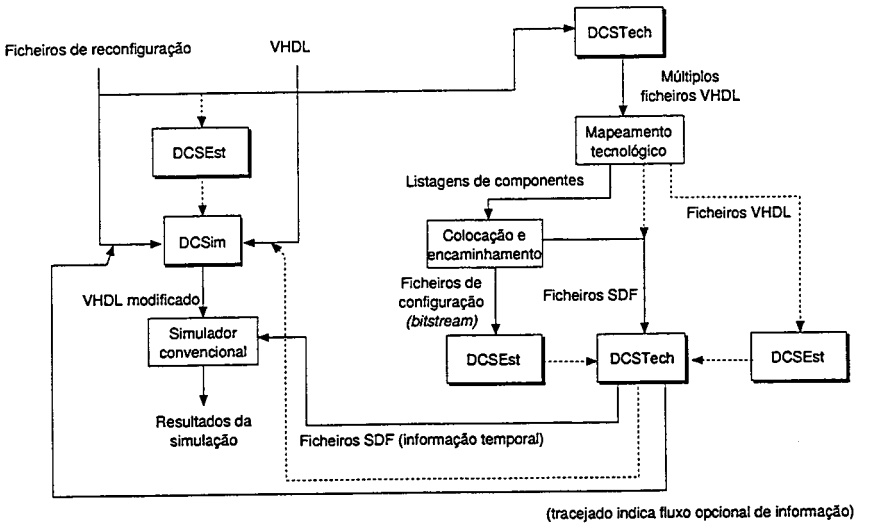
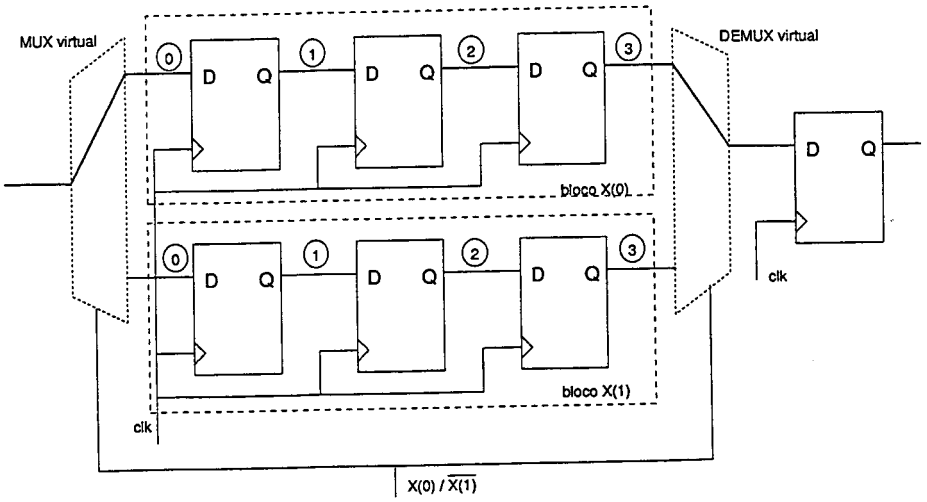


Figura 3.29: Ambiente de desenvolvimento baseado na metodologia DCS.

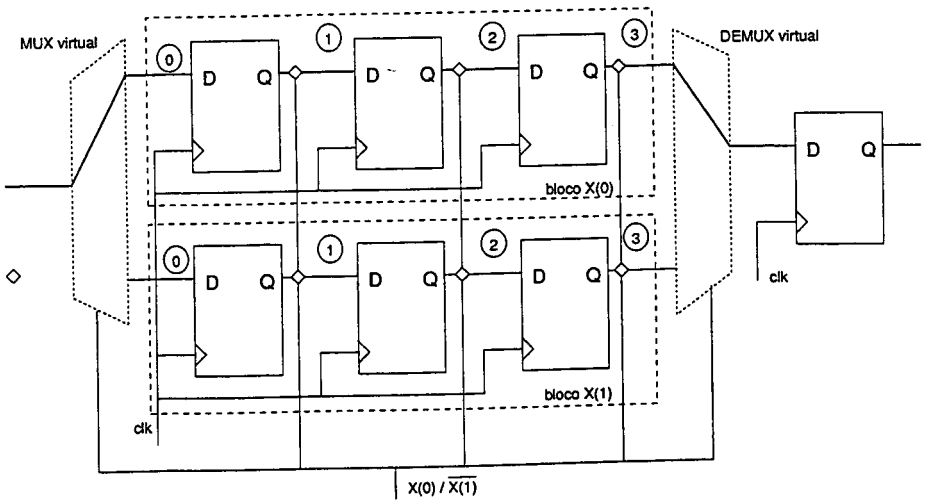
múltiplos circuitos estáticos correspondentes às diferentes possibilidades de reconfiguração; por outro, combina a informação temporal resultante do processamento dos múltiplos circuitos e produz a informação temporal agregada para a especificação original. DCSEst é usado para estimar os diversos factores que afectam o tempo de reconfiguração. Esta ferramenta pode ser usada em diversas fases do desenvolvimento, dependendo da qualidade da informação temporal disponível.

O método DCS apresenta alguns problemas na simulação de sistemas do tipo *pipelined* (figura 3.30). Quando um bloco *pipelined* é desactivado, o valor 'X' colocado à sua entrada demora alguns ciclos de relógio a propagar-se por todo o bloco; caso o bloco seja reactivado antes de essa propagação terminar, a sua saída pode assumir um valor correspondente à anterior activação (em vez de ter o valor indefinido até todo o bloco ter sido re-inicializado). A solução proposta por Vasilko e Cabanis [1999] consiste em acrescentar circuitos para isolar as saídas de todos os andares, garantindo assim que todos os elementos de memória do circuito desactivado assumem o valor indefinido. Esta solução tem a desvantagem de necessitar de modelos específicos (com comutadores entre andares) para os blocos reconfiguráveis.

Outro problema deste método é o uso do valor 'X' para representar o efeito da reconfiguração: este valor não permite distinguir a situação normal de reconfiguração da ocorrência de erros (por exemplo, devido a contenção de sinais ou falhas de inicialização). Adicionalmente, o valor 'X' não é sempre propagado pelo módulo reconfigurado (por exemplo,uma porta AND que tenha uma entrada a 'X e outra a '0' tem a saída sempre a '0'), o que não é o comporta-



(a) Simulação DCS de um circuito *pipelined*



(b) Simulação DCS com comutadores de isolamento entre andares

Figura 3.30: Simulação errada de blocos *pipelined* pelo método DCS e (b) possível correcção.

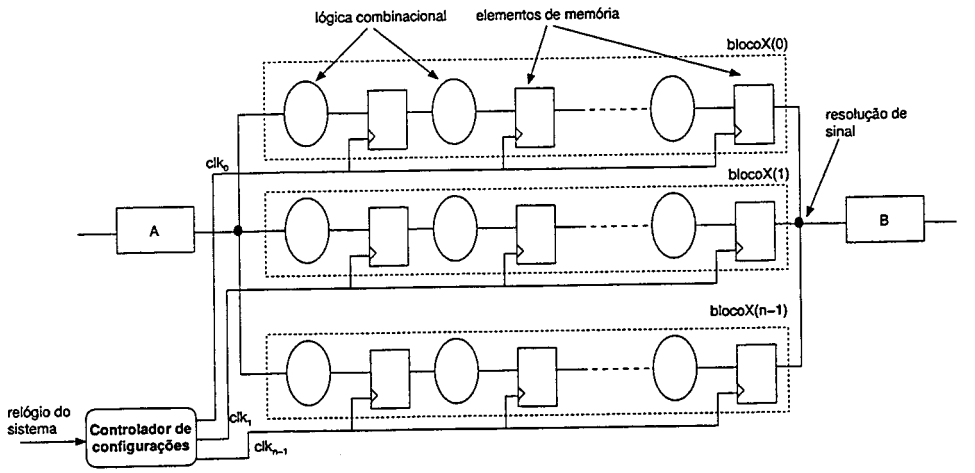


Figura 3.31: O método *Clock Morphing* de simulação de um SDRD.

mento pretendido para um sinal que representa o estado de um módulo não-residente (que se encontra, todo ele, num estado não determinado). Vasilko e Cabanis [1999] propõem o uso de um valor especial 'V' para indicar o estado de reconfiguração e mostram como especificar o seu comportamento em VHDL.

O método *Clock Morphing* proposto por Vasilko e Cabanis [1999] permite a simulação correcta de SDRD sem o recurso a versões especiais dos blocos reconfiguráveis (figura 3.31). Para tal, são definidos sinais de relógio diferentes para cada configuração, todos dependentes de um controlador de reconfiguração. Este simula a configuração do bloco seleccionado efectuando a ligação do seu sinal de relógio ao sinal global pelo período de tempo apropriado. Os sinais de relógio dos módulos desactivados são colocados ao valor 'V', que é propagado correctamente para as saídas dos elementos de memória. As regras de resolução do valor 'V' são tais que a saída combinada dos vários módulos alternativos (entrada do módulo B na figura 3.31) estará sempre numa das seguintes situações:

saída='V' Nenhum dos módulos alternativos está activo;

saída='X' Existe um problema de contenção de sinal (activação simultânea de dois módulos);

outros valores O circuito opera correctamente, estando activado apenas um dos módulos alternativos.

Uma abordagem alternativa ao problema da simulação de SDRD é descrita por Bondalapati e Prasanna [1999a]. Estes autores apresentam o sistema DRIVE (*Dynamically Reconfigurable systems Interpretative simulation and Visualization Environment*) como um ambiente de

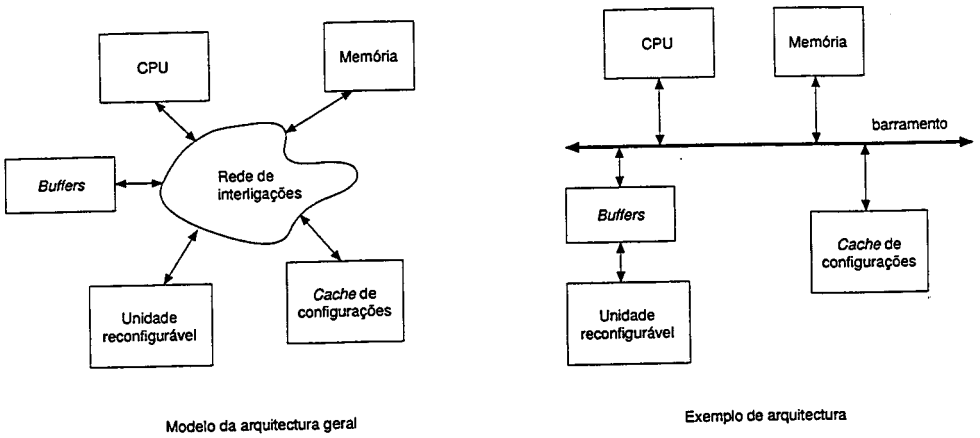


Figura 3.32: O modelo HySAM.

apoio à avaliação de desempenho de SDRD e exploração de alternativas de projecto. A infra-estrutura de *hardware* é caracterizada por um modelo paramétrico de alto nível, sobre o qual é simulado um modelo abstracto da aplicação. Por comparação com a simulação de modelos em LDH, este método permite reduzir significativamente o esforço de simulação.

Os autores do sistema DRIVE desenvolveram um modelo abstracto e parametrizável de um SDRD designado por HySAM *(Hybrid System Architecture Model)*, que está ilustrado na figura 3.32 [Bondalapati e Prasanna, 1999a]. O modelo abarca sistemas compostos por microprocessadores convencionais e circuitos reconfiguráveis. O modelo baseia-se na distinção entre capacidades (que representam a funcionalidade da aplicação) e implementações (que representam as configurações existentes). O modelo abstracto inclui, entre outros, os seguintes parâmetros:

F Conjunto de funções $F_1 \ldots F_n$ que podem ser executadas nos circuitos configuráveis (capacidades);

C Conjunto de configurações $C_1 \ldots C_m$ (implementações)

A_{ij} Conjunto de atributos da implementação da função F_i usando a configuração C_j;

R_{ij} Custo da reconfiguração de C_i para C_j;

A descrição conceptual do sistema DRIVE está ilustrada na figura 3.33. A arquitectura do sistema é caracterizada para se determinarem os valores dos parâmetros do modelo HySAM correspondente. Os módulos de *hardware* existentes são igualmente caracterizados (em termos de capacidades e implementações). A informação sobre os módulos de *hardware* e sobre a arquitectura são combinadas para produzir o modelo do sistema. Da análise da aplicação

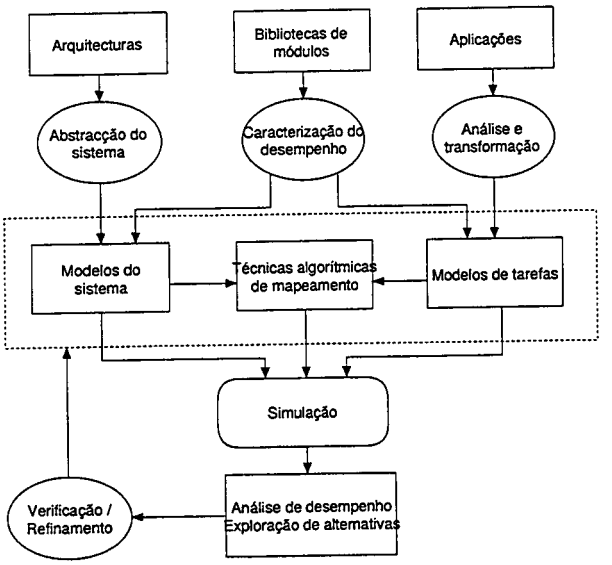


Figura 3.33: Modelo conceptual do sistema de simulação DRIVE.

em termos das funcionalidades disponíveis produz-se um modelo da tarefa, possivelmente na forma de um grafo acíclico de funções F_i em que as arestas denotam a existência de dependências entre tarefas. Algoritmos específicos são usados para mapear o modelo da tarefa no modelo do sistema (mapeamento $F_i \rightarrow C_j$) a fim de se poder proceder à simulação conjunta. Por exemplo, Bondalapati e Prasanna [1999b] descrevem alguns destes algoritmos para a aceleração de cálculos numéricos no interior de ciclos, enquanto Bondalapati e Prasanna [1999c] mostram como mapear alguns tipos de ciclos de um programa em circuitos reconfiguráveis multi-contexto. A simulação permite obter uma indicação do desempenho, bem como explorar diferentes opções por variação dos parâmetros do modelo. Eventualmente pode proceder-se ao refinamento dos modelos ou à alteração dos mapeamentos.

3.4.3 Identificação de módulos dinâmicos

As características específicas dos SDRD levam naturalmente à necessidade de criar algumas ferramentas que não existem nos ambientes de desenvolvimento convencionais, como, por exemplo, o já referido avaliador parcial de McKay e Singh [1998]. Também a tarefa de criar e gerir conjuntos de configurações mutuamente exclusivas mereceu a atenção dos investigadores. O desafio principal consiste em identificar os componentes de cada configuração de maneira a maximizar a utilização dos recursos disponíveis, enquanto se minimiza o tempo e área gastos com a reconfiguração dinâmica.

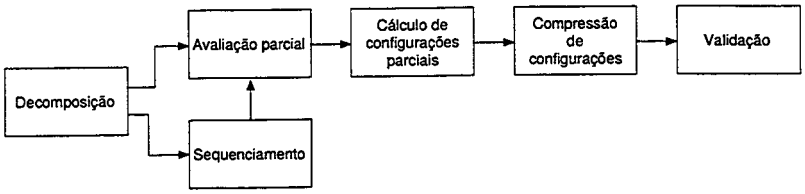


Figura 3.34: Processo de desenvolvimento de um SDRD segundo Luk *et al.* [1997].

Luk *et al.* [1997] apresentam um processo de desenvolvimento de SDRD em 6 passos (figura 3.34): decomposição, sequenciamento, avaliação parcial, cálculo de configurações incrementais, compressão dos dados de configuração e validação. Os primeiros três passos, bem como o último, podem ser aplicados a qualquer dispositivo reconfigurável; o quarto passo é específico de circuitos FPGA capazes de reconfiguração parcial; o quinto passo aplica-se a dispositivos, como o circuito FPGA XC6216, que incorporam mecanismos especiais que permitem a compressão dos dados de configuração.

No primeiro passo procede-se à decomposição do sistema em regiões reconfiguráveis apropriadas. No final desse passo o sistema pode ser representado como uma rede de blocos de controlo interligando as possíveis configurações para cada módulo reconfigurável (segundo o modelo já referido de Luk *et al.* [1996]) e de uma sequência de condições de activação para cada módulo.

No segundo passo, a sequência é usada para determinar que transições entre configurações ocorrerão durante a execução. A ferramenta de sequenciamento de Luk *et al.* [1997] pode gerar um controlador de reconfigurações em *hardware* ou um programa de controlo em linguagem C.

O terceiro passo consiste na utilização de um avaliador parcial que implementa os métodos de Luk *et al.* [1996] (secção 3.4.2, pág. 96). O quarto passo é implementado por um ferramenta que determina as diferenças entre duas configurações sucessivas. Desde que a sequência de activações esteja disponível, esta ferramenta é invocada para calcular os dados de reconfiguração apenas para as transições que efectivamente ocorrem.

Uma versão mais sofisticada desta ferramenta implementa também o passo cinco, aproveitando características específicas da interface de configuração da família XC6200 para criar uma versão mais compacta dos dados de configuração. Por exemplo, esta família permite configurar simultaneamente células idênticas que residam na mesma coluna a partir de uma só instância dos dados de configuração.

O último passo serve para verificar que o comportamento do sistema é o desejado e que são respeitadas as restrições de utilização de recursos é atingido o desempenho pretendido. Luk *et al.* [1997] não apresentam ferramentas específicas para esta tarefa. Contudo, alguns dos métodos de validação descritos na secção 3.4.2 são certamente aplicáveis neste caso.

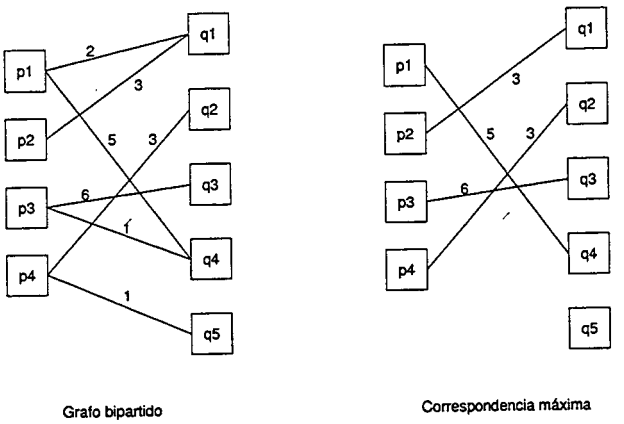


Figura 3.35: Grafo bipartido das configurações P e Q, e a sua correspondência máxima.

O trabalho de Shirazi *et al.* [1998a] completa o anterior ao investigar métodos para automatizar a identificação de regiões configuráveis, levando em conta os elementos comuns de configurações sucessivas. A tarefa de decomposição é dividida em duas etapas, das quais apenas a segunda foi automatizada:

1. identificação dos componentes para reconfiguração e respectivas condições de activação;
2. optimização de configurações sucessivas para obter o ponto de equilíbrio entre tempo de reconfiguração, frequência de reconfiguração e dimensão dos elementos a reconfigurar.

A eficácia do segundo passo depende dos resultados do primeiro, pelo que pode ser necessário iterar entre os dois para se obter uma boa decomposição.

O tempo de reconfiguração é reduzido quando se evita reconfigurar os componentes comuns a duas configurações sucessivas, ou quando se procede à reconfiguração entre componentes similares (por exemplo, entre um somador e um subtractor). O método de Shirazi *et al.* [1998a] desenvolve-se em três etapas:

1. representar os componentes do uma configuração como nós de um grafo bipartido, cujas arestas representam as correspondências possíveis entre componentes de cada configuração; o "peso" de cada aresta representa a "força" da correspondência respectiva;
2. calcular a melhor correspondência global;
3. inserir os componentes virtuais de Luk *et al.* [1996] (referidos na pág. 96) para indicar explicitamente os elementos reconfiguráveis.

O peso de cada aresta é determinado por três factores: tipo, posição no circuito FPGA (se conhecida) e nível hierárquico dos portos de acesso aos componentes. Dois componentes do mesmo tipo levam a uma aresta de peso muito elevado; componentes semelhantes levam a um peso proporcional à sua semelhança (obtidos a partir de uma lista pré-definida manualmente). Depois de feita a determinação da correspondência de peso máximo é criado um novo grafo em que os componentes reconfiguráveis são combinados como definido por Luk *et al.* [1996] para se obter um novo circuito que pode servir de base a mais uma iteração do processo. Shirazi *et al.* [1998a] descrevem várias aplicações desta abordagem a exemplos na área do reconhecimento de padrões e filtros digitais.

Em numerosas situações, o processo de desenvolvimento de SDRD é uma adaptação do processo convencional de desenvolvimento de CIAE ou outros circuitos de arquitectura estática. Estas metodologias podem exigir várias iterações porque não dispõem da capacidade de fornecer estimativas precisas sobre o efeito da reconfiguração nem de re-apreciar decisões arquitecturais já tomadas.

O sistema DYNASTY de Vasilko [1999] propõe uma abordagem diferente do processo de desenvolvimento, com ênfase na manipulação dos aspectos temporais do projecto. Uma sequência típica de etapas de desenvolvimento é a seguinte:

1. concepção e descrição do projecto numa LDH ou por captura esquemática;
2. selecção das partes estáticas do projecto;
3. exploração das alternativas de reconfiguração dinâmica, envolvendo um conjunto de ferramentas dedicadas para o efeito;
4. geração das configurações para a solução adoptada.

Entre as ferramentas usadas no passo 3 contam-se (figura 3.36):

- visualizador de grafo de fluxo (controlo e dados) : fornece informação sobre o comportamento do sistema em fase de implementação.
- visualizador da estrutura hierárquica: apresenta os detalhes sobre todos os elementos do projecto e respectiva relação hierárquica; permite a atribuição de módulos a operadores do grafo de fluxo e às arestas de controlo.
- editor de sequenciamento: permite visualizar e controlar a evolução temporal das subtarefas.
- planeador 3D: permite visualizar e manipular os módulos da implementação, tanto geometricamente sobre a área disponível no circuito FPGA (a duas dimensões), como temporalmente entre duas ou mais configurações (a terceira dimensão).

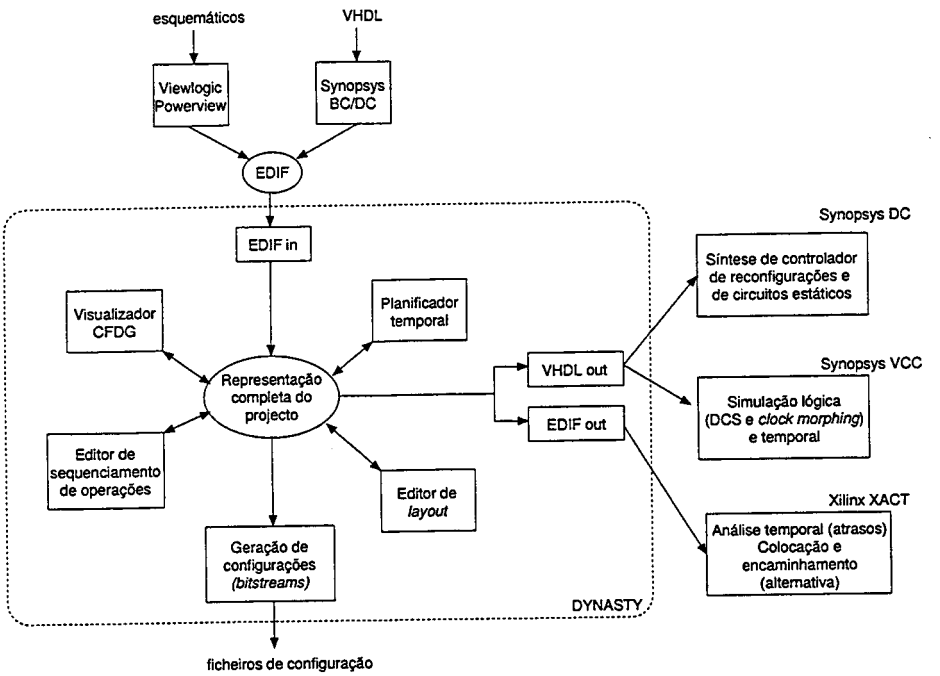


Figura 3.36: O ambiente de desenvolvimento DYNASTY.

- editor de pormenor: permite manipular os elementos físicos tais como recursos de encaminhamento e configuração de células.

A estrutura e parâmetros de cada alternativa de projecto podem ser manipulado directamente através de uma interface gráfica: uma modificação numa das vistas é imediatamente propagada às outras

O sistema pode ainda gerar modelos de simulação VHDL do projecto segundo as metodologias DCS ou *Clock Morphing*, já referidas anteriormente. Também estão disponíveis interfaces para outras ferramentas, incluindo ferramentas de colocação e encaminhamento. A implementação do sistema descrita por Vasilko [1999] suporta a família XC6200.

3.4.4 Gestão de recursos em tempo de execução

Frequentemente, a reconfiguração dinâmica é controlada por *software* que decide quando reconfigurar a infra-estrutura física. Nas situações mais simples, o programa de controlo limita-se a seleccionar a configuração apropriada e transmiti-la para o circuito FPGA.

Aplicações mais complexas necessitam de sistemas de apoio correspondentemente mais

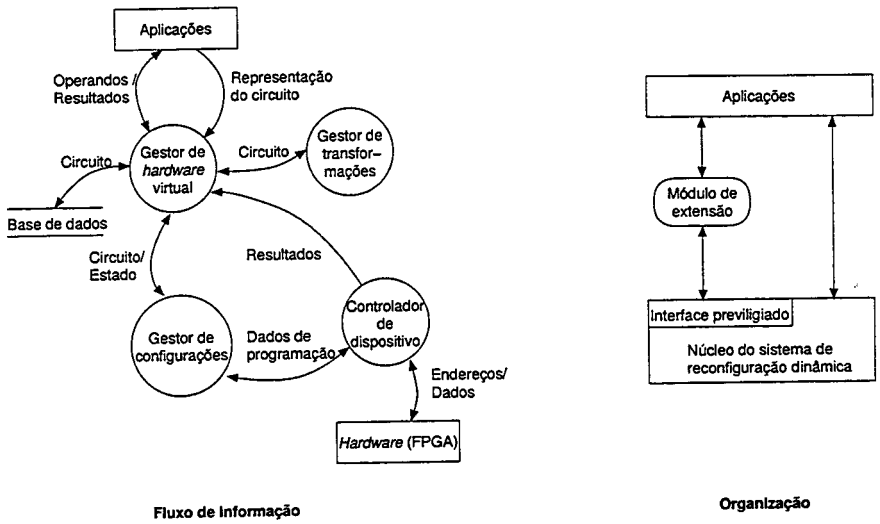


Figura 3.37: O sistema de apoio à execução RAGE.

sofisticados. O objectivo do sistema de Burns *et al.* [1997] é ter uma interface de alto nível à disposição das aplicações que desejem efectuar tarefas complexas de configuração. O sistema proposto pretende satisfazer os seguintes requisitos obtidos por análise de vários sistemas já implementados:

- possibilidade de reconfigurar circuitos FPGA e aceder a recursos da infra-estrutura de *hardware* sem comunicar directamente com o controlador de dispositivos *(device driver)*;
- possibilidade de reservar uma zona do circuito FPGA para outras utilizações;
- capacidade de transformar um circuito (por exemplo, modificar a sua orientação ou posição);
- representação simbólica (alto nível) do circuito que permita efectuar modificações da sua estrutura, bem como a capacidade de transformar essa representação em dados de configuração;
- uma representação do circuito FPGA com informação arquitectural correctamente representada.

A figura 3.37 ilustra o sistema RAGE de apoio à execução proposto por Burns *et al.* [1997]. Um gestor de *hardware* virtual coordenada as operações dos outros elementos e fornece a interface principal entre a aplicação e o sistema de apoio à execução. As chamadas ao gestor

de *hardware* virtual permitem tornar um módulo residente, comunicar com ele e libertar os seus recursos após a utilização. O gestor mantém um mapa da utilização do circuito FPGA, embora o mapeamento desses recursos no dispositivo físico seja responsabilidade de outro elemento da arquitectura, o gestor de configurações. Este elemento é responsável por configurar o circuito FPGA, efectuar a reconfiguração parcial de módulos residentes e passar dados da aplicação para o circuito.

O controlador de dispositivos constitui a base do sistema sobre a qual os outros elementos constroem a respectiva funcionalidade. Entre a funcionalidade implementada por este elemento básico contam-se a leitura e escrita de dados de configuração, o acesso a memória, o tratamento de interrupções, o controlo do sinal de relógio e a activação passo-a-passo do circuito.

O gestor de transformações modifica os módulos (por rotação ou translação) de maneira a permitir a sua colocação nas posições desejadas. Para tal é preciso ter em atenção os pormenores específicos da arquitectura de cada dispositivo, podendo mesmo ser necessário refazer parte do encaminhamento interno dos módulos.

O sistema inclui ainda uma interface privilegiada para módulos de extensão (figura 3.37) que acrescentam funcionalidade adicional, por exemplo, um módulo de especialização que efectue propagação de constantes.

A aplicação geral de reconfiguração dinâmica pode levar à necessidade de resolver dinamicamente problemas de encaminhamento. Naturalmente que não é possível determinar o encaminhamento em tempo de execução para situações arbitrárias. Contudo, Brebner e Donlin [1998] investigam algumas situações particulares em que esse objectivo pode ser atingido, nomeadamente utilizando estruturas genéricas de encaminhamento (p. ex. matrizes de encaminhamento) que podem ser rapidamente reconfiguradas para fornecer a conectividade desejada.

Shirazi *et al.* [1998b] propõem um sistema de gestão de reconfigurações baseadas em módulos relocáveis e implementável em *hardware*. O gestor de configuração é constituído por um monitor, um configurador e uma memória de configuração. O monitor mantém informação sobre o estado do sistema, incluindo tipo e posição dos módulos presentes no circuito FPGA. Quando são verificadas as condições para a mudança de configuração, o monitor activa o configurador, que por sua vez instala o novo módulo no circuito FPGA a partir da memória de configuração. Esta última é constituída por três componentes: uma directoria de configurações, um repositório de dados e um "agente de transformação", capaz de produzir versões de alguns módulos a partir de módulos-padrão pré-definidos.

Em alguns casos os sistemas de gestão de configurações estão disponíveis como bibliotecas de rotinas. O trabalho de McGregor *et al.* [1998] é um exemplo desta abordagem, em que uma interface de *software* orientada aos objectos é usada para simplificar o controlo e gestão de aplicações reconfiguráveis. A interface é constituída por três camadas (figura 3.38). O nível inferior é constituído pelo controlador de dispositivos que assegura o acesso à infra-estrutura

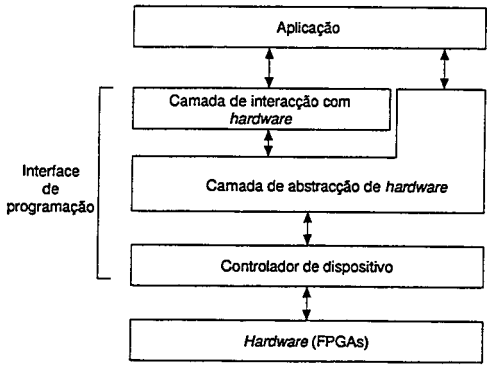


Figura 3.38: Organização da interface de controlo para operações de reconfiguração de McGregor *et al.* [1998].

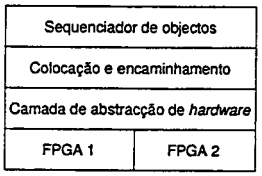


Figura 3.39: Sistema de apoio à execução ACEruntime.

de *hardware*. A camada de abstracção de *hardware* ocupa uma posição intermédia, sendo responsável pela inicialização de *hardware*, controlo do sinal de relógio (execução ciclo-a-ciclo), e pela gestão de toda a informação simbólica associada ao sistema. O último nível, a camada de interacção, é responsável por sequenciamento e controlo de reconfiguração, gestão de dados de configurações e transferências de blocos de dados. Esta camada também é responsável pela reconfiguração dinâmica desencadeada por interrupções, mantendo para o efeito uma tabela de correspondência entre eventos de interrupção e dados de reconfiguração.

Estas bibliotecas de rotinas foram usadas na implementação de uma ferramenta de projecto interactiva, que permite controlar e observar o estado da infra-estrutura de *hardware*, bem como as diferentes operações de reconfiguração [McGregor *et al.*, 1998].

Outro exemplo de gestão em tempo de execução é proporcionado pelo sistema ACEcard descrito por Ban [1998]. O núcleo do sistema é uma placa com dois circuitos XC6264 e um microprocessador microSPARC-IIep. O sistema, que suporta reconfiguração dinâmica, é programado em linguagem Java através de uma biblioteca de classes que permite fazer corresponder objectos em Java a módulos residentes em *hardware*. O sistema de gestão em tempo de execução, cuja arquitectura está ilustrada na figura 3.39, corre no microprocessador do

sistema e é composto por três camadas:

Abstracção de dispositivos Esta camada isola as restantes da base de *hardware*, fornecendo uma interface de leitura/escrita e gerindo a informação básica sobre os circuitos disponíveis, suas dimensões e recursos;

Colocação e encaminhamento Esta camada gere o espaço disponível nos circuitos FPGA, decidindo onde colocar cada módulo e como efectuar a sua interligação;

Sequenciação de objectos Esta camada traduz os pedidos de alto nível gerados pelo código em Java para operações de baixo nível da camada de colocação e encaminhamento; em particular, esta camada determina quando é que um módulo deve estar presente nos circuitos FPGA e gere os detalhes da troca de módulos.

Este sistema tem sido usado em processamento de imagens [Ban, 1998].

O trabalho de Fleischmann *et al.* [1999] descreve outro gestor de reconfigurações em tempo de execução, desta feita no contexto de um sistema de co-projecto *hardware/software* baseado na linguagem Java. Este gestor sequencia funções para execução em *software* (na máquina virtual de Java) ou em *hardware* (numa plataforma reconfigurável dinamicamente). Neste último caso, o gestor verifica se a configuração apropriada já está instalada, e, caso contrário, procede à reconfiguração total ou parcial de um circuito FPGA disponível.

Jean *et al.* [1999] descrevem um sistema de gestão de recursos que implementa reconfiguração dinâmica sobre uma infra-estrutura baseada em circuitos FPGA convencionais. A infra-estrutura contém até 16 módulos reconfiguráveis, cada um com dois circuitos XC4020E, 2 MB de memória dinâmica e 256 KB de memória estática. Em cada instante, o computador hospedeiro executará diversos programas cuja operação é acelerada por um subconjunto de módulos reconfiguráveis, sendo que a utilização desses módulos é feita à medida das necessidades (e das possibilidades). O sistema de gestão deve garantir a partilha da infra-estrutura e a alocação dinâmica de recursos (i.e., de módulos configuráveis a programas em execução). Jean *et al.* [1999] descrevem a implementação em *software* de um gestor de recursos concorrente *(multi-threaded)* capaz de configurar módulos dinamicamente e especulativamente (i.e., antes de o programa requerer a sua utilização). A configuração especulativa (ou por antecipação) permite ter um módulo devidamente configurado assim que a aplicação necessita, não sendo o desempenho desta prejudicado pelo tempo de configuração.

Para planear a sua acção o gestor dispõe de um grafo de fluxo que descreve as operações que cada aplicação executa na infra-estrutura reconfigurável. Cada nó do grafo representa as tarefas executadas nos circuitos FPGA e cada aresta o fluxo de controlo do programa. Com esta informação, o gestor aloca e liberta módulos configuráveis de forma que novos nós possam ser instanciados enquanto outros se encontram em execução. Cada nó usa um subconjunto de módulos (até 16) para a sua implementação.

O gestor foi usado com sucesso numa aplicação de processamento de imagens em tempo real (reconhecimento de alvos em imagens do espectro infra-vermelho) [Jean *et al.*, 2000] e em codificação de vídeo segundo a norma MPEG-2.

O sistema RACE *(Reconfigurable and Adaptive Computing Environment)* de Smith e Bhatia [1996] é outro exemplo de um sistema com circuitos FPGA convencionais que suporta reconfiguração dinâmica. Neste caso, a reconfiguração é invocada explicitamente por rotinas de *software* que descarregam os dados de configuração e esperam pelos resultados. O sistema é suportado por uma biblioteca de componentes que facilitam a produção dos programas.

3.4.5 Síntese de alto nível

Conforme mostram as últimas secções, muitos ambientes de desenvolvimento aproveitam ferramentas tradicionais, acrescentando programas adicionais para lidar com os aspectos particulares de reconfiguração dinâmica. Geralmente os novos programas simplificam os aspectos mais básicos do desenvolvimento de SDRD, deixando ao utilizador o tratamento manual dos aspectos de nível mais alto. Alguns sistemas de compilação de linguagens de alto nível constituem uma excepção, uma vez que isolam completamente o utilizador dos aspectos de reconfiguração dinâmica.

Com o tempo têm vindo a surgir mais esforços orientados para apoiar o processo de desenvolvimento de SDRD a mais alto nível. Um exemplo importante é o sistema CRUSADE [Dave, 1999], usado na síntese conjunta *hardware/software* de sistemas *embedded* reconfiguráveis dinamicamente.

CRUSADE opera com especificações de sistemas em termos de grafos acíclicos de tarefas e gera uma arquitectura heterogénea (incluindo circuitos reconfiguráveis dinamicamente, CIAE e processadores) que verifica todas as restrições de tempo real e minimiza o custo total de *hardware*. O sistema é capaz de identificar os grupos de tarefas a serem implementados em SDRD e sintetizar as correspondentes interfaces de programação.

A síntese de arquitecturas reconfiguráveis dinamicamente envolve a resolução de três tipos de problemas:

Gestão de atrasos O atraso através de um circuito FPGA depende da colocação e encaminhamento dos respectivos módulos. Como a atribuição de tarefas a suportes de implementação precede a avaliação dos tempos de atraso, é necessário desenvolver técnicas de gestão de atrasos que garantam que as restrições temporais são respeitadas. Neste caso, optou-se por restringir a ocupação dos circuitos FPGA e a utilização dos pinos.

Gestão de reconfigurações É necessário identificar os grupos de tarefas a implementar em SDRD de maneira a minimizar o número de reconfigurações, bem como o tempo de cada reconfiguração. O sistema CRUSADE identifica tarefas mutuamente exclusivas no tempo (i.e., aquelas que podem ser atribuídas ao mesmo suporte de implementação).

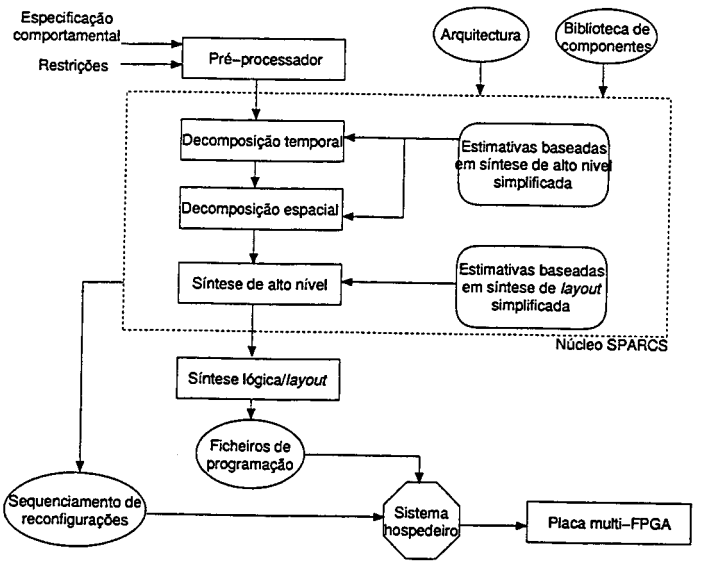


Figura 3.40: O sistema de síntese SPARCS.

Síntese de interfaces de controlo A reconfiguração de circuitos FPGA é feita sob o controlo de interfaces geradas especificamente para a tarefa. O objectivo é minimizar o custo do sistema sem exceder os limites de tempo especificados (já que, em geral, métodos mais lentos de configuração—por exemplo, via interface série—necessitam de menos recursos).

CRUSADE usa um algoritmo heurístico e iterativo, que sucessivamente refina o sequenciamento de tarefas, bem como a sua atribuição aos circuitos reconfiguráveis, na tentativa de obter uma arquitectura de custo total mínimo que satisfaça todas as restrições temporais. Dave [1999] apresenta resultados desta abordagem para numerosos exemplos realistas e com elevado número de tarefas. A redução de custos obtida varia entre 30% e 56%, o que ilustra bem o potencial dos SDRD.

Rath e Li [1998] esboçam uma abordagem à síntese de alto nível de unidades de controlo reconfiguráveis dinamicamente, que se baseia numa representação tabular designada por *Timed Decision Table*, que pode ser derivada automaticamente de descrições em LDH de alto nível. A partir dessa descrição é criado um grafo de controlo *(control-flow graph)* que é de seguida dividido em componentes conexas, que representam as configurações individuais. A abordagem é similar à decomposição de máquinas de estados esboçada anteriormente (secção 3.3.3, pág. 82).

Outro esforço ambicioso na área da síntese de alto nível é descrito por Kaul *et al.* [1998]. O

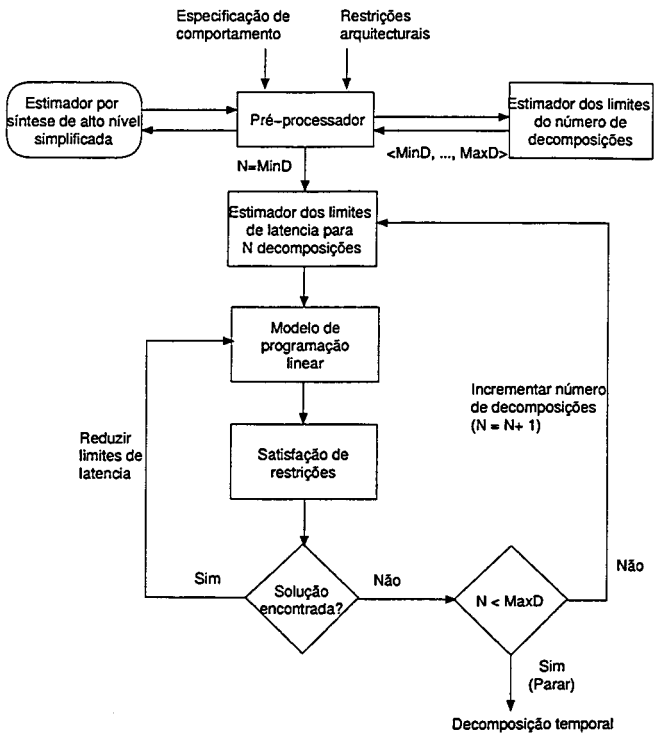


Figura 3.41: Processo de decomposição temporal usado no sistema SPARCS.

sistema SPARCS *(Synthesis and Partitioning for Adaptive Reconfigurable Computers)* é constituído por um conjunto de ferramentas que permitem efectuar a decomposição temporal e espacial de sistemas digitais de forma integrada com as tarefas de síntese (figura 3.40). O sistema aceita especificações em VHDL (ao nível do comportamento, das transferências entre registos (RTL) ou das portas lógicas) e, após decomposição espacial e temporal, bem como síntese de alto e baixo nível, produz múltiplas configurações bem como o respectivo sequenciamento.

O sistema SPARCS está orientado para sistemas multi-FPGA com memórias locais, uma memória global partilhada e intercomunicação por matriz de encaminhamento. O seu principal objectivo é transformar uma representação hierárquica de alto nível do comportamento desejado numa implementação baseada em reconfiguração dinâmica. O sistema SPARCS contém ferramentas para várias tarefas:

Decomposição temporal Esta ferramenta recebe uma descrição das diversas tarefas a executar (grafo de tarefas) e das suas inter-relações e procura obter uma decomposição final

que minimize a latência e satisfaça as restrições de área impostas pelos circuitos disponíveis (fig. 3.41). O método adoptado consiste em explorar exaustivamente as alternativas possíveis. Para tal, a ferramenta começa por usar um sintetizador de alto nível para estimar as características das implementações das diferentes tarefas. Com base nessas estimativas, determina em seguida um intervalo para o número de decomposições temporais a explorar. Esse intervalo é explorado de tal modo que, para um dado número de decomposições, se procede ao refinamento iterativo da estimativa do tempo de latência associado, verificando sempre se é possível respeitar as restrições do problema (através da resolução de um programa linear).

Decomposição espacial Esta ferramenta usa um algoritmo genético para repartir a implementação de cada partição temporal pelos diferentes componentes disponíveis. Também aqui um sintetizador de alto nível é usado para obter estimativas dos custos de cada decomposição espacial.

Síntese de alto nível A síntese de alto nível está a cargo de uma ferramenta já existente, o sistema DSS *(Distributed Synthesis System)*. Este sistema é capaz de operar em dois modos: o mais leve é usado para produzir estimativas de alto nível e é usado pelas ferramentas anteriores; o modo mais pesado é usado para produzir descrições ao nível da operações entre registos (RTL) a partir de descrições comportamentais. Esta ferramenta também é usada para sintetizar o controlador que implementa o sequenciamento. O controlador está organizado como uma conjunto de máquinas de estados finitas síncronas que comunicam entre si.

Projecto final As últimas fases do projecto, incluindo a colocação e encaminhamento, bem como a geração dos dados de configuração, é feita por ferramentas comerciais.

3.4.6 Algoritmos de decomposição temporal e sequenciamento

O principal problema adicional introduzido pela reconfiguração dinâmica consiste em especificar e sintetizar as sucessivas "etapas" temporais que vão partilhar a mesma infra-estrutura digital. No cenário ideal, o sistema digital completo é definido sem atender às limitações impostas pelas capacidades dos circuitos FPGA usados; apenas numa fase posterior se procede à sua implementação por desdobramento no tempo de maneira a não serem necessários, em cada instante, mais recursos que os efectivamente disponíveis.

Este problema está relacionado com dois outros, muito estudados na literatura técnica [de Micheli, 1994; Sait e Youssef, 1995; Gajski *et al.*, 1992] e também relacionados entre si: a decomposição de circuitos *(partitioning)* e o sequenciamento de operações *(scheduling)*. No caso da decomposição procura-se uma repartição equilibrada dos componentes por diversos conjuntos. Um exemplo de decomposição é a distribuição da implementação de um sistema digital por vários circuitos FPGA. De maneira análoga, a decomposição temporal de circuitos

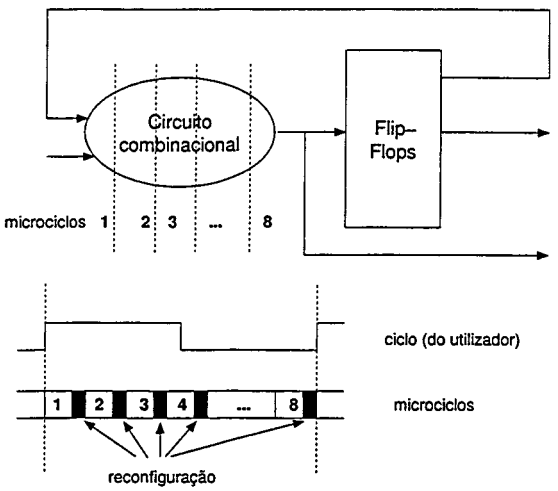


Figura 3.42: Modelo simples de desdobramento temporal para circuitos FPGA com múltiplos contextos.

procura repartir as diferentes operações pelas diferentes etapas de processamento, geralmente satisfazendo diversas restrições no número total de etapas ou na complexidade de implementação de cada uma.

O objectivo do sequenciamento de operações é determinar a ordem total de execução destas por forma a minimizar o tempo total, respeitando simultaneamente restrições de precedência e de compatibilidade entre as operações. É uma tarefa frequente em sistemas de síntese de alto nível [de Micheli, 1994; Gajski *et al.*, 1992]. No caso dos SDRD, o objectivo do sequenciamento é determinar a ordem de reconfiguração das partições por forma a obter um tempo total mínimo, entrando em conta com os tempos de reconfiguração, transferência de dados e execução de cada etapa.

Os problemas de decomposição e sequenciamento surgem frequentemente combinados no caso dos SDRD (geralmente sob a designação genérica de *decomposição temporal*) e adquirem características próprias consoante o cenário concreto da sua aplicação. Em alguns casos, o problema põe-se para sistemas digitais descritos ao nível da porta ou bloco lógicos. Por exemplo, o sistema já pode ter sido mapeado para um circuito FPGA de dimensão infinita e o que se pretende em seguida é distribuir a sua implementação em etapas sucessivas. Exemplos desta abordagem incluem [Trimberger, 1998; Chang e Marek-Sadowska, 1999; Liu e Wong, 1999]. Em alternativa, o problema pode ser abordado no contexto de síntese de alto nível, em que se procura sequenciar um conjunto de tarefas que serão sintetizadas posteriormente [Kaul e Vemuri, 1998].

Uma primeira linha de actividade centra-se no aproveitamento de circuitos FPGA multi-

contexto. Trimberger [1998] aborda o problema de decompor um sistema para caber no circuito FPGA multi-contexto descrito por [Trimberger *et al.*, 1997] (secção 3.2.1, pág. 71). O modelo de operação subjacente está ilustrado na figura 3.42. De acordo com este modelo, um ciclo de relógio é decomposto em vários microciclos, durante os quais se procede à avaliação de parte do circuito original (se possível, apenas um andar combinacional). Entre cada micro-ciclo, o circuito FPGA é reconfigurado. Por ciclo do relógio principal o sistema executa uma sequência completa de microciclos. O circuito FPGA contém registos que permitem guardar os valores lógicos dos sinais combinacionais entre microciclos, bem como os valores dos *flip-flops* entre ciclos do relógio principal. No início de cada micro-ciclo, os valores de saída de todos os blocos lógicos são guardados em micro-registos. De seguida procede-se ao carregamento de uma nova configuração. Os valores dos micro-registos propagam-se através dos circuitos combinacionais, determinando o estado de outros micro-registos. Num circuito correctamente sequenciado, todos os elementos são avaliados uma vez por ciclo do relógio principal.

Partindo de um projecto já mapeado em blocos lógicos virtuais (com tabelas de memória e *flip-flops*), que excedem o número de blocos reais disponíveis, pretende-se decompor a avaliação do circuito em microciclos, de tal maneira que as saídas primárias sejam idênticas às produzidas por um circuito FPGA sem reconfiguração. Esta questão pode ser abordada como um problema de sequenciamento com diversos tipos de soluções. O sequenciamento ASAP *(as-soon-as-possible)* ou ALAP *(as-late-as-possible)* permite obter soluções no menor número de microciclos possível [Gajski *et al.*, 1992]. Contudo, o número de blocos lógicos utilizados por micro-ciclo não é equilibrado: o primeiro método tende a concentrar a utilização de blocos lógicos nos primeiros microciclos, enquanto o segundo tende a concentrá-la nos microciclos finais. No contexto presente, é importante reduzir a ocupação máxima em cada micro-ciclo, porque tal permite utilizar circuitos com menos blocos lógicos, bem como simplificar as operações de encaminhamento.

A solução adoptada por Trimberger [1998] consiste em determinar os sequenciamentos ASAP e ALAP, e aplicar em seguida o algoritmo de sequenciamento por fila de prioridades *(list scheduling)* com uma medida de qualidade que inclui o número de microciclos (que não deve exceder o determinado pelo método ALAP) e favorece o agrupamento de interligações no interior de um micro-ciclo (para reduzir a comunicação entre microciclos bem como o número de interligações a replicar em vários microciclos).

Após o sequenciamento por fila de prioridades aplica-se ainda uma heurística de optimização que troca pares de blocos lógicos entre microciclos adjacentes numa tentativa reduzir o número de interligações em cada micro-ciclo.

Os sequenciamentos ASAP e ALAP permitem determinar o percurso crítico do circuito, que é composto por todos os blocos que são atribuídos ao mesmo micro-ciclo pelos dois métodos. Este percurso crítico pode ter um comprimento superior ao número de configurações suportadas pelo circuito FPGA (oito no caso do circuito experimental de Trimberger *et al.* [1997]).

Nesse caso, é necessário "comprimir" o sequenciamento de forma a que mais que um nível do circuito seja implementado por micro-ciclo. A compressão faz naturalmente aumentar a duração de cada micro-ciclo (já que os sinais lógicos devem ser propagados através de vários blocos lógicos).

Trimberger [1998] usa um método heurístico para resolver este problema, combinando os níveis adjacentes que levam ao menor número de elementos no percurso crítico em cada micro-ciclo. Este método é aplicado antes do sequenciamento por fila de prioridades. Liu e Wong [1999] formulam o mesmo problema em termos da determinação do percurso de peso mínimo num grafo com certas características e apresenta um algoritmo para o resolver de forma exacta.

O trabalho de Chang e Marek-Sadowska [1999] constitui outra abordagem ao sequenciamento (ao nível da porta lógica) em circuitos FPGA multi-contexto. Este trabalho distingue-se do anterior por usar um modelo mais genérico do problema e por adoptar a técnica de sequenciamento forçado *(force-directed scheduling—DFS)* [Paulin e Knight, 1989] em lugar de sequenciamento por fila de prioridades.

O modelo de execução usado por Chang e Marek-Sadowska [1999] aplica-se a diferentes circuitos FPGA multi-contexto, incluindo o protótipo de Trimberger *et al.* [1997], Dharma (pág. 71) e DPGA (pág. 68), e suporta de forma natural múltiplos andares em cada etapa. A técnica de sequenciamento forçado procura equilibrar o número de elementos de cada partição (etapa) de maneira a respeitar um limite de tempo global. Em cada iteração considera-se o impacto de inserir um dado elemento em todas as etapas possíveis, escolhendo-se depois a melhor atribuição. A variante usada neste caso inclui explicitamente o custo da comunicação entre etapas, o que permite obter resultados globais sensivelmente superiores aos obtidos pela simples aplicação da técnica básica.

Uma terceira abordagem ao mesmo problema é descrita por Liu e Wong [1998], que o transformam num problema de decomposição de grafos. O modelo representa correctamente as ligações multi-terminal entre os elementos e inclui todas as restrições de precedência. Para dividir o grafo em duas partições equilibradas, o algoritmo efectua repetidos cálculos do fluxo máximo que, dada a forma específica do grafo, determinam cortes mínimos que satisfazem todas as restrições de precedência. A aplicação repetida do algoritmo de bipartição permite obter uma partição equilibrada do circuito em múltiplos conjuntos correspondentes às diversas etapas. Os resultados experimentais indicam melhorias apreciáveis (superiores a 20%) em comparação com o sequenciamento por fila de prioridades.

O problema da decomposição temporal também foi abordado um nível mais abstracto. O trabalho de Purna e Bhatia [1999] apresenta algoritmos para a decomposição de grafos de fluxo de dados. Dado um sistema com uma capacidade fixa e uma tarefa representada por um grafo de fluxo acíclico (que se supõe exceder a capacidade do sistema), os algoritmos permitem decompor o grafo em segmentos que não excedem a capacidade do sistema (figura 3.43). Esses segmentos são depois sequenciados de maneira a preservar as relações de precedência entre

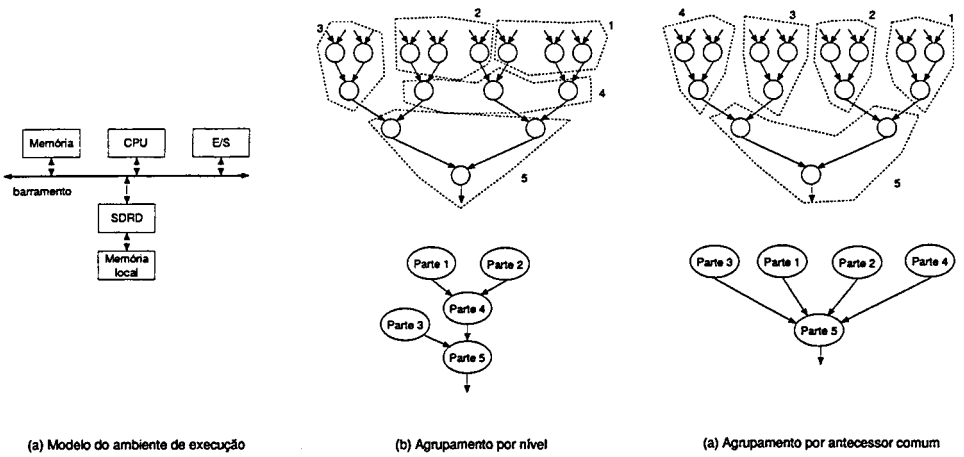


Figura 3.43: Decomposição temporal de grafos de fluxos de dados.

eles.

No cenário descrito por Purna e Bhatia [1999] os segmentos são posteriormente sintetizados e as respectivas configurações armazenadas no computador hospedeiro. Um programa, o sequenciador, envia as configurações para o sistema reconfigurável numa ordem que satisfaça as relações de precedência. O armazenamento dos dados transferidos entre segmentos é feito na memória local do sistema reconfigurável. A cada segmento é associado um controlador (uma máquina de estados finita) que controla a leitura dos dados (na inicio) e a escrita dos resultados (no fim). O controlador é gerado automaticamente após a partição do grafo.

Purna e Bhatia [1999] propõem dois outros algoritmos de decomposição. O primeiro tenta agrupar os nós do grafo segundo o seu nível ASAP, o que preserva o maior grau de paralelismo possível na implementação e permite diminuir o atraso global. Um exemplo da aplicação deste algoritmos está ilustrado na figura 3.43b. A segunda abordagem agrupa nós com antecessores comuns para reduzir o custo da comunicação entre segmentos (o número de arestas que entram ou saem do segmento). A figura 3.43c ilustra a aplicação deste algoritmo. Os resultados experimentais indicam que a segunda abordagem reduz em cerca de 30% os custos de comunicação em comparação com a primeira, mas à custa de uma degradação significativa do atraso global (cerca de 67%).

O sistema NENYA de Cardoso e Neto [1999] converte os fragmentos expressos em código-octeto *(bytecode)* de Java para circuitos a implementar em *hardware* virtual. O sistema extrai as dependências existentes entre blocos básicos (e também no seu interior) e guarda essa informação em vários tipos de grafos, incluindo grafos de fluxo de dados. A partir dessa representação intermédia o compilador de *hardware* efectua a decomposição temporal, resolve

conflitos de acesso a memória e gera código VHDL (ao nível das transferências entre registos) que é posteriormente mapeado em dispositivos reconfiguráveis.

A partição temporal usa uma versão refinada do agrupamento por nível de Purna e Bhatia [1999]. Na abordagem original, os nós a incluir em cada grupo são seleccionados aleatoriamente de entre os nós de um mesmo nível. No caso do sistema NENYA os nós de cada nível são ordenados de acordo com uma política de prioridades que confere primazia aos nós do percurso crítico.

Outra variação do sistema NENYA procura tornar a construção de grupos mais sofisticada. Na abordagem original, uma partição é encerrada logo que o nó seleccionado não cabe nela, passando-se imediatamente à construção de outra partição. No sistema NENYA tenta-se ainda incluir outros nós na partição corrente através de uma pesquisa recursiva dos nós candidatos restantes.

O sistema de síntese de alto nível SPARCS (secção 3.4.5, pág. 3.4.5) inclui entre as suas tarefas a decomposição temporal da representação comportamental. Os seus autores produziram diversos trabalhos em que descrevem algumas abordagens a variantes deste problema [Kaul e Vemuri, 1998, 1999; Kaul *et al.*, 1999]. Todas são centradas na formulação de modelos de programação matemática e combinam a decomposição temporal com outras tarefas de alto nível.

Kaul e Vemuri [1998] apresentam um modelo de programação não linear que combina partição temporal, sequenciamento e selecção e atribuição de unidades funcionais. O objectivo do modelo é minimizar a quantidade de dados transferida entre segmentos temporais para reduzir os custos associados à reconfiguração dinâmica. Para resolver o modelo não linear, o modelo é linearizado de forma a obter-se um modelo de programação linear inteira. A abordagem proposta, embora geral, é de aplicação viável apenas a pequenas instâncias.

Kaul e Vemuri [1999] descrevem ainda outra variante da decomposição temporal. Neste caso o objectivo é proceder à exploração sistemática das alternativas de projecto através de um procedimento iterativo de refinamento que procura minimizar a latência total do projecto final. O núcleo da heurística de pesquisa é formado por um modelo de programação linear inteira, que é resolvido repetidamente para vários pontos do espaço de projecto.

Uma outra variante do problema surge em aplicações de processamento digital de sinal. Neste contexto é frequente o SDRD ser usado no interior de um ciclo (figura 3.44). Para uma implementação de SDRD com N etapas, um ciclo com I iterações executa $N \times I$ reconfigurações. Uma alternativa é amortizar o tempo de reconfiguração executando K iterações por configuração (figura 3.44b). Neste caso o número total de reconfigurações é $N \times (I/K)$, mas é necessário espaço de memória para armazenar os resultados intermédios. Por sua vez a gestão de dados intermédios também pode ser complexa. Por exemplo, pequenas quantidades podem ser guardadas localmente no SDRD, enquanto quantidades maiores podem exigir transferências para a memória do sistema hospedeiro. Kaul *et al.* [1999] apresentam um modelo de programação linear inteira que combina decomposição temporal com desdobramento

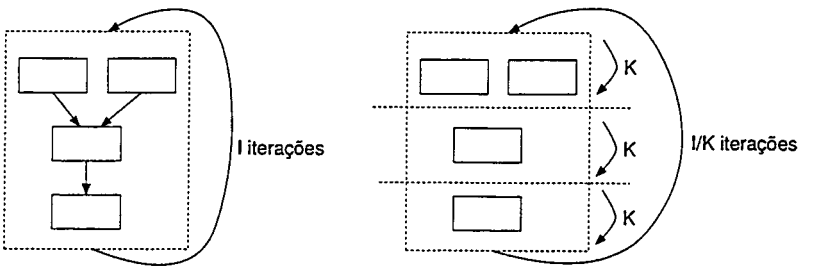


Figura 3.44: Desdobramento de ciclos *(loop fission)*.

de ciclos *(loop fission)*, de maneira a obter a latência mínima tendo em conta o tempo de reconfiguração e de transferências dos dados intermédios.

O estudo experimental deste método aplicado à implementação da transformada de cosseno discreta (usado no algoritmo de compressão da imagens JPEG) permite observar a influência dos diversos factores em jogo. Uma implementação estática necessita de 160 ciclos de relógio a 100 ns. O modelo referido levou a uma implementação com três partições, a primeira usando 68 ciclos a 50 ns, e as outras duas 36 ciclos a 70 ns. Sem o tempo de reconfiguração, a versão dinâmica leva menos 7650 ns que a estática. No SDRD usado neste exemplo o tempo de reconfiguração é relativamente elevado (cerca de 100 ms), pelo que é necessário amortizar o custo de reconfiguração por um número elevado de iterações. A solução produzida pelo modelo permite ainda assim obter uma versão mais vantajosa que a estática; as vantagens são tanto mais nítidas quanto maior for o número de iterações (que depende do tamanho da imagem a tratar). Este exemplo ilustra bem que a reconfiguração dinâmica pode trazer vantagens em termos de desempenho global (mesmo que seja possível uma solução estática) mas apenas se for prestada atenção cuidadosa a todos os factores envolvidos.

3.5 Resumo

O uso de reconfiguração dinâmica para melhorar o desempenho e o aproveitamento de recursos de um sistema digital é uma nova abordagem cujas potencialidades têm vindo a ser exploradas em diversos contextos, conforme este capítulo pretendeu mostrar. A diversidade de maneiras como a reconfiguração dinâmica pode ser usada dificulta a sua categorização. Um critério classificativo simples baseia-se em determinar se a informação necessária à reconfiguração dinâmica é determinada antes ou durante a execução. A informação necessária inclui as configurações e o seu sequenciamento, o que permite distinguir quatro tipos de sistemas, desde os mais simples (configurações e sequenciamento pré-determinados) aos mais dinâmicos (tanto configurações como sequenciamento são determinados durante a execução

da aplicação em função dos dados processados).

Uma questão que se põe naturalmente é saber se a reconfiguração dinâmica traz benefícios a uma dada aplicação. É difícil fazer afirmações gerais porque existe uma influência recíproca entre reconfiguração dinâmica (um método de implementação) e a arquitectura do sistema. Quando existe uma limitação dos recursos lógicos disponíveis devido a outros factores, sejam técnicos ou económicos, a reconfiguração dinâmica pode ser a única maneira de implementar o sistema pretendido. Noutras situações é preciso fazer uma avaliação caso a caso. É importante notar que uma implementação baseada em reconfiguração dinâmica pode apresentar um melhor desempenho que as alternativas, mesmo quando o tempo de reconfiguração não é negligenciável; para isso é preciso que o desempenho durante cada fase seja suficientemente bom para compensar o tempo de reconfiguração. Muitas vezes essas melhorias podem ser obtidas por circuitos que são especificamente adaptados para uma dada fase da execução da aplicação, e que portanto, podem ter melhor desempenho que os circuitos mais gerais que têm de ser usados quando essa adaptação não é possível.

Considerações gerais levam a um critério simples para determinar se a adopção de reconfiguração dinâmica pode ser vantajosa: a razão entre o tempo de configuração e o tempo de execução deve ser inferior à máxima melhoria que pode ser obtida. Esta relação, embora geral e de utilidade na orientação do projectista, pode ser de aplicação difícil em casos concretos. Por um lado pode não ser possível determinar com rigor o tempo de configuração (p.ex., para sequenciamentos ou configurações variáveis); por outro lado é difícil determinar com rigor a máxima melhoria que pode ser obtida para uma dada aplicação, porque tal pode implicar a avaliação de muitas e diferentes arquitecturas alternativas, e ser dependente dos dados a tratar.

Existem vários circuitos FPGA com suporte específico para reconfiguração dinâmica, alguns dos quais mesmo disponíveis comercialmente. Esse suporte assume geralmente uma de duas formas: possibilidade de alternar entre múltiplas configurações armazenadas no circuito reconfigurável (circuitos multi-contexto) ou possibilidade de reconfigurar apenas parte do circuito, mantendo o restante em funcionamento (reconfiguração parcial). Entre os circuitos comerciais a segunda opção é a mais frequente (Xilinx XC6200 e Virtex, Atmel AT6000 e AT40K); a arquitectura multi-contexto é usada apenas nos circuitos FIPSOC.

Os circuitos FPGA referidos têm sido usados em alguns protótipos de SDRD orientados para a investigação aplicada nesta área, a grande maioria dos quais emprega circuitos da família XC6200. Apesar das actividades de investigação, os SDRD ainda não encontraram aplicação comercial relevante.

Muitas estratégias têm sido tentadas para aproveitar a reconfiguração dinâmica, desde a simples comutação periódica de configurações até à construção de configurações em tempo de execução. A própria variabilidade associada à reconfiguração dinâmica implica uma grande diversidade de abordagens, diversidade essa que aumentará ainda mais no futuro à medida que for acumulada experiência nesta área ainda jovem. Por exemplo, a exploração de recon-

figuração dinâmica no contexto do co-projecto *software/hardware* é uma direcção promissora ainda praticamente inexplorada.

A variabilidade temporal dos SDRD vem acrescentar uma dimensão suplementar à já difícil tarefa de projectar sistemas digitais, o que vem conferir ainda mais importância à existência de ferramentas de apoio convenientes. Estas aliás são frequentemente apontadas como o calcanhar de Aquiles dos sistemas reconfiguráveis mais convencionais. Para além de tornar mais difícil tarefas tradicionais como especificação e simulação, a reconfiguração dinâmica vem introduzir novos problemas como a identificação dos módulos dinâmicos ou a decomposição temporal das tarefas e respectivo sequenciamento. Já foram dados alguns passos na resolução destes desafios, geralmente na forma de extensões a ambientes de desenvolvimento tradicionais; contudo muita falta ainda fazer para permitir uma utilização desenvolta deste tipo de sistemas. O trabalho apresentado na segunda parte desta tese pretende ser um contributo para esse fim.

Parte II

Síntese de sistemas reconfiguráveis dinamicamente

Plan to throw one away, you will anyway.

The Mythical Man-Month
FREDERICK P. BROOKS

In theory there is no difference between theory and practice. In practice there is.

YOGGI BERRA

The devil is in the details.

POPULAR

Resumo

Esta parte descreve pormenorizadamente a concepção e o desenvolvimento de um sistema de apoio à criação e utilização de aplicações baseadas em circuitos reconfiguráveis dinamicamente desenvolvido pelo autor e designado por ReDiFlex (REconfiguração DInâmica FLEXível). A exposição evolui dos aspectos mais gerais para os mais particulares (*top-down*), o que não reflecte naturalmente o processo iterativo de refinamento da especificação do sistema nem o processo *bottom-up* de implementação, mas permite obter uma visão unificada do sistema.

O primeiro capítulo desta parte (cap. 4) descreve os objectivos gerais do sistema ReDiFlex, os diferentes cenários de utilização, a sua organização geral e as principais opções de implementação tomadas.

Os restantes capítulos descrevem os principais módulos do sistema. O capítulo 5 trata o modelo funcional usado para especificar a alto nível o comportamento do co-processador reconfigurável dinamicamente e a forma de que se reveste a sua utilização. A implementação do modelo é apresentada em detalhe, incluindo a representação interna da especificação, as operações que esta suporta (e os respectivos algoritmos), bem como a estratégia de mapeamento para o domínio físico.

A representação do modelo físico e o seu tratamento formam o objecto do capítulo 6. Os principais assuntos abordados são: modelo físico do circuito a implementar em *hardware*, geradores de componentes, e algoritmos de colocação e encaminhamento.

O capítulo 7 descreve a geração de configurações a partir do modelo físico e a forma de efectuar o controlo da infra-estrutura reconfigurável (inicialização, controlo do sinal de relógio, transmissão de dados e configurações).

O capítulo 8 descreve o sistema de apoio à execução. Os principais tópicos abordados são: activação e execução (da implementação) do modelo funcional, interface com o sistema de programação hospedeiro, e apoio à depuração e traçagem da execução.

Finalmente, o capítulo 9 apresenta algumas reflexões finais sobre reconfiguração dinâmica, resume o trabalho efectuado e indica algumas perspectivas de actividade futura.

4 Apresentação geral do sistema ReDiFlex

A grande variedade de formas de utilizar a reconfiguração dinâmica e a complexidade intrínseca desta técnica tornam importante a existência de sistemas eficazes de apoio ao desenvolvimento. Estes sistemas devem fornecer ferramentas automáticas de apoio às principais etapas de implementação, promovendo ao mesmo tempo com a interactividade, facilidade de utilização e expansibilidade necessárias ao aproveitamento, exploratório por natureza, de uma técnica de implementação nova e de aplicação pouco sistematizada. A concepção e desenvolvimento do sistema ReDiFlex (REconfiguração DInâmica FLEXível) guiaram-se por estes princípios.

Este capítulo dá uma perspectiva geral do sistema ReDiFlex, incluindo os objectivos que visa, os princípios que nortearam as opções técnicas tomadas ao longo da sua implementação, e a arquitectura resultante. O seu principal objectivo é fornecer o enquadramento técnico global para as descrições mais detalhadas dos capítulos seguintes. O presente capítulo começa pela exposição dos motivos que levaram à construção do sistema e dos objectivos globais que visa. A apresentação das opções gerais de implementação é efectuada na secção seguinte; a análise dessas opções é feita à luz dos objectivos anteriormente estabelecidos. Aos objectivos está subjacente um modelo de utilização, i.e., um conjunto de cenários que descrevem as actividades dos utilizadores do sistema. Conforme se verá, os utilizadores podem assumir vários papéis (ou dito de outra maneira, o sistema pode ter diversos tipos de *clientes*), pelo que é importante estabelecer os principais cenários de utilização. O capítulo encerra com

uma descrição da organização geral do sistema implementado, indicando os seus principais módulos e as respectivas responsabilidades.

4.1 Motivação e objectivos

As técnicas de reconfiguração dinâmica de circuitos FPGA prometem melhorar o desempenho de sistemas digitais dedicados e a gestão dos respectivos recursos. Essa promessa não pode ser realizada por uma abordagem genérica; pelo contrário, a forma de usar a reconfiguração dinâmica depende das características específicas da tarefa a que é aplicada, e o seu sucesso depende crucialmente do aproveitamento dos aspectos particulares de cada caso. Daí resulta uma multiplicação das abordagens possíveis, conforme o capítulo 3 ilustra.

A análise das experiências descritas na literatura técnica faz ressaltar um conjunto de aspectos comuns:

- A aplicabilidade da reconfiguração dinâmica deve ser avaliada caso a caso; frequentemente é necessário avaliar globalmente o sistema a implementar e optar por abordagens menos convencionais para tirar devidamente proveito desta técnica de implementação. A adopção de reconfiguração dinâmica pode levar o projectista a ter de repensar o problema na sua globalidade.

- Ao permitir a alteração no tempo da estrutura lógica de *hardware*, a reconfiguração dinâmica introduz uma dimensão adicional ao projecto; a complexidade acrescida traduz-se numa exigência de ferramentas de apoio, que, contudo, não estão disponíveis porque o domínio de aplicação é muito fragmentado (cf. ponto anterior) e porque a tecnologia é relativamente recente.

- Um número grande de aplicações de reconfiguração dinâmica envolve simultaneamente *hardware* e *software*, necessitando de ambientes de execução relativamente sofisticados (p.ex., para gestão de configurações, desencadeamento das operações de reconfiguração, adaptação a modificações do ambiente operacional). Frequentemente, a integração dos dois domínios é rudimentar, o que naturalmente coloca mais um entrave à utilização desta técnica. É de notar que este problema se põe talvez com mais acuidade para os SDRD, mas é aplicável a quase todos os sistemas reconfiguráveis.

Estes aspectos, aliados às potencialidades da reconfiguração dinâmica, levam naturalmente à investigação de quais as melhores formas de aproveitar esta técnica de implementação e quais as características mais desejáveis para os respectivos ambientes de desenvolvimento e execução. Interessa igualmente avaliar a viabilidade de implementação das abordagens propostas.

É neste contexto geral que foi desenvolvido o sistema ReDiFlex. O principal objectivo deste protótipo é fornecer um ambiente de desenvolvimento e execução de aplicações que usem reconfiguração dinâmica sobre uma infra-estrutura do tipo co-processador com reconfiguração parcial (cf. secções 2.5.2.2 e 3.1). Como a utilização sistemática destes SDRD ainda é incipiente, o protótipo deve permitir a exploração assistida de várias estratégias, pelo que deve ser flexível (para evitar restringir as opções do utilizador) e interactivo, i.e., capaz de indicar rapidamente ao utilizador as consequências das suas opções e disponibilizar o máximo de informação possível numa forma que possa ser manipulada automaticamente pelos programas que o utilizador desenvolva.

Em sistemas que envolvem simultaneamente *hardware* e *software* existe quase sempre uma perspectiva dominante, num ou noutro sentido. No caso do sistema ReDiFlex entendeu-se que seria importante exigir ao utilizador a menor quantidade possível de conhecimentos detalhados de *hardware*. Embora não se querendo esconder a infra-estrutura física, pretendeu-se que o utilizador possa abstrair consideravelmente dos respectivos detalhes, para se poder concentrar na exploração das várias possíveis soluções para a sua aplicação. Por outras palavras, o utilizador deve ter uma visão idealizada e relativamente simples da infra-estrutura física. De facto, o utilizador do protótipo não entra em "contacto" com a arquitectura física do circuito FPGA, mas trabalha apenas com um modelo abstracto cuja estrutura é modificável durante a execução da aplicação. É apenas em termos desse modelo que o utilizador precisa de raciocinar e também é em termos dele que especifica as diversas mutações. O sistema encarrega-se de mapear o modelo abstracto em configurações do circuito FPGA e as suas modificações nas correspondentes reconfigurações. A investigação das características deste modelo de utilização e da forma como ele se combina com os demais aspectos do sistema é outro dos objectivos do trabalho aqui apresentado.

Outra faceta desta perspectiva é a preocupação em proporcionar uma boa integração do modelo abstracto com a linguagem em que é desenvolvida a aplicação, para não perturbar desnecessariamente a unidade conceptual da solução que o utilizador procura desenvolver.

O protótipo implementado constitui um sistema integrado verticalmente, em que as diversas tarefas de apoio ao desenvolvimento de *hardware* (decomposição, sequenciamento, colocação, encaminhamento, geração de configurações) são executadas num mesmo ambiente, garantindo a disponibilidade total da informação obtida a cada passo. Esta organização conceptual tem a vantagem de não impor *a priori* uma divisão que restrinja as opções de implementação. Naturalmente que é preciso assegurar uma organização interna estruturada para manter a complexidade do sistema sob controlo. A maneira mais simples de assegurar flexibilidade e extensibilidade na presença de complexidade consiste na adopção de uma organização por módulos, em que cada um usa os serviços de outros módulos e por sua vez disponibiliza serviços bem definidos aos demais. A extensibilidade do ambiente pode ser avaliada pela facilidade com que podem ser acrescentadas novos módulos ou modificados os existentes (mantendo a interface com os restantes).

Porque a viabilidade das abordagens deve ser validada pela execução num ambiente real representativo e facilmente disponível, o sistema ReDiFlex usa a placa H.O.T. Works (descrita na secção 3.2.2). Esta placa está equipada com um circuito FPGA Xilinx XC6216 e deve ser ligada ao barramento PCI de um computador pessoal. Do ponto de vista lógico, o circuito FPGA é mapeado no espaço de endereçamento do CPU, podendo ser eficientemente acedido para reconfiguração e transferência de dados.

A escolha da infra-estrutura reconfigurável é importante porque determina algumas características do sistema ReDiFlex. Outras opções teriam sido tomadas para um SDRD integrado no CPU ou para grandes sistemas periféricos; de igual modo, o aproveitamento da reconfiguração parcial leva a abordagens diferentes das que seriam requeridas para gerir um dispositivo multi-contexto.

Resumindo, o protótipo ReDiFlex pretende validar um sistema automático de apoio ao uso de reconfiguração dinâmica (reconfiguração parcial de co-processador) em sistemas do tipo IV com sequenciamento variável e configurações dependentes dos dados (cf. classificação da secção 3.1). Os aspectos mais relevantes são:

- definição de um modelo de especificação de um SDRD;
- definição de um modelo de utilização do sistema de desenvolvimento;
- definição de um modelo de interacção entre um CPU convencional (programa de controlo) e um co-processador reconfigurável;
- integração com ambiente de programação e linguagens suportadas;
- extracção automática de toda a informação necessária à operação de um SDRD real a partir de uma especificação de alto-nível;
- ênfase no carácter exploratório da actividade de concepção de um SDRD;
- utilização e aproveitamento de uma infra-estrutura real;
- organização aberta e extensível para servir de suporte a outras abordagens;
- suporte para os vários tipos de utilizadores (criadores de aplicações, implementadores de ferramentas e especificadores de componentes básicos).

Para além disso, este trabalho também visa obter conhecimentos e tirar conclusões que sejam relevantes para desenvolvimentos futuros, tanto ao nível das características dos sistemas de apoio como da própria infra-estrutura física. Pode dar-se o caso das vantagens de uma dada característica ou opção de implementação não serem claras se analisadas isoladamente, mas mesmo assim terem bom aproveitamento no contexto em que são adoptadas e contribuírem para um sistema cujas potencialidades são superiores à da simples soma das partes. O desenvolvimento integral de um sistema tem aqui a vantagem de fornecer um contexto mais completo para a avaliação conjunta das várias características relevantes.

4.2 Opções gerais de implementação

Os objectivos enunciados na secção anterior não determinam, só por si, a estratégia geral de implementação do sistema ReDiFlex. Para orientar a implementação foi necessário tomar algumas decisões de carácter geral que são analisadas nas subsecções seguintes.

Para além das situações analisadas explicitamente a seguir, existem sempre muitas questões de implementação que admitem diferentes soluções. Nesses casos adoptou-se o princípio de privilegiar a a flexibilidade sobre a eficiência para respeitar a motivação inicial de ter um sistema de apoio à exploração das possibilidades abertas pela reconfiguração dinâmica. Estes dois factores não estão necessariamente em conflito: muitas vezes, a flexibilidade de um sistema pode ser usada para o adaptar a casos particulares, que então podem ser tratados muito eficientemente. Aliás é precisamente isso que acontece com o próprio conceito de reconfiguração dinâmica, em que a flexibilidade é usada para adaptar o comportamento da infra-estrutura de *hardware* às necessidades específicas de cada instância de um problema em diferentes instantes.

Outro princípio orientador da implementação foi a preferência por soluções simples em detrimento de abordagens mais sofisticadas. Para cada sub-problema optou-se geralmente pelos algoritmos mais simples que permitissem resolvê-lo sem comprometer os objectivos globais. O resultado prático desta decisão é tornar o sistema mais robusto (já que a simplicidade se traduz directamente numa menor probabilidade de existência de falhas ocultas) e reduzir o esforço de implementação. Desde que a flexibilidade da implementação seja mantida, a adopção das abordagens mais simples não debilita permanentemente o sistema: caso se venha a determinar que a eficiência de um qualquer algoritmo compromete os objectivos gerais é relativamente simples substituí-lo por uma versão mais sofisticada.

4.2.1 Descrição explícita da funcionalidade de *hardware*

Genericamente podemos encontrar duas abordagens à descrição da funcionalidade a implementar no co-processador: implícita ou explícita. No primeiro caso, existe uma descrição de um processo computacional (geralmente numa linguagem de programação de uso geral) e o sistema deve extrair daí a especificação da funcionalidade a implementar [Cardoso e Neto, 1999; Razdan, 1994]. No segundo caso, essa funcionalidade é explicitamente indicada por uma especificação conceptualmente separada.

Como o ênfase neste projecto é colocado na gestão da reconfiguração dinâmica, optou-se por usar uma descrição explícita da computação a ser executada no processador. Desta forma evita-se o problema difícil da extracção da funcionalidade explícita, que, embora importante e interessante, não está no âmbito do objectivos enunciados na secção anterior, pertencendo mais propriamente ao domínio do projecto combinado *hardware/software* (hardware/software *codesign*). Esta decisão corresponde a limitar a implementação do protótipo ao âmbito des-

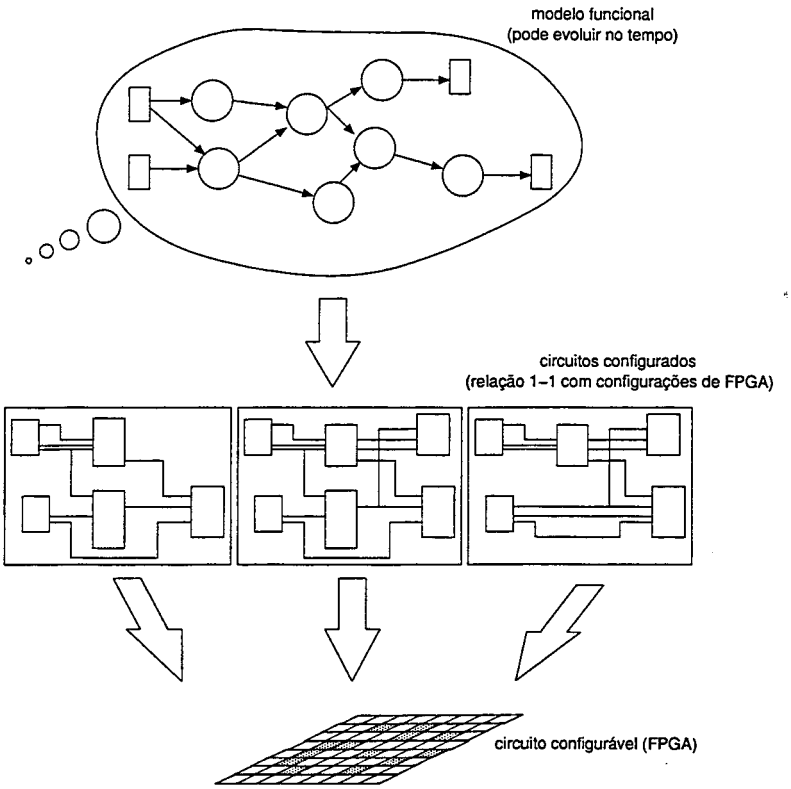


Figura 4.1: Níveis de descrição de *hardware* no sistema ReDiFlex.

ta dissertação; contudo a natureza aberta e expansível do protótipo permite, em princípio, acrescentar novos módulos ao sistema e estendê-lo a outros domínios.

Determinar a sequência das operações de reconfiguração a partir de uma descrição implícita obriga ao emprego de um modelo de execução que se adapte à linguagem de programação, i.e., que respeite a semântica da linguagem. Por exemplo, com linguagens orientadas aos objectos é possível mapear as operações de criação/destruição de objectos em operações de reconfiguração [Cardoso e Neto, 1999; Bellows e Hutchings, 1998].

No modelo usado pelo sistema ReDiFlex optou-se antes por separar o comportamento dinâmico do modelo da descrição da sua funcionalidade. Para isso deixou-se ao utilizador a liberdade (e a complexidade) de especificar explicitamente as mutações do modelo.

No contexto do sistema ReDiFlex é útil distinguir entre o modelo funcional (abstracto e de mais alto nível) e a descrição da "personalidade" que o circuito FPGA assume num dado instante (mais ligada à implementação física), conforme ilustrado na fig. 4.1. O primeiro é

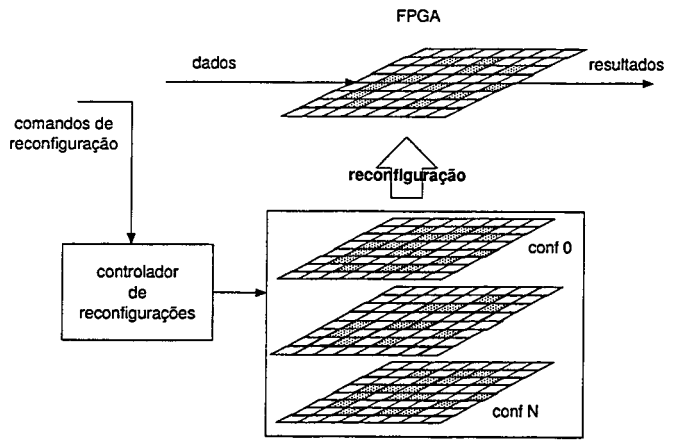


Figura 4.2: Tarefas conceptuais num SDRD.

usado pelo utilizador para especificar as operações de *hardware* e as respectivas mutações. O segundo descreve o circuito digital (o circuito configurado) em termos dos recursos físicos do circuito FPGA (blocos configuráveis, interligações, elementos de memória) e é gerado automaticamente pelas ferramentas do sistema. A cada modelo funcional são associados, por decomposição e síntese da representação física, um ou mais circuitos configurados (a "implementação" do modelo funcional). Mutações do modelo funcional são repercutidas automaticamente em alterações dos circuitos configurados correspondentes.

Para a execução efectiva da computação é necessário derivar as configurações dos correspondentes circuitos configurados e usá-las para estabelecer o comportamento do circuito FPGA. Todo este trabalho é feito automaticamente pelo sistema ReDiFlex.

4.2.2 Modelo de utilização de *hardware*

Num SDRD podemos distinguir conceptualmente dois tipos de operações: processamento de dados e o controlo das operações de reconfiguração (ver fig. 4.2). O processamento de dados é a tarefa primária que se pretende executar no SDRD. No contexto do protótipo ReDiFlex é a forma de proceder a esse processamento que é descrita pelo modelo funcional.

Em princípio existem duas alternativas no que diz respeito à gestão das configurações. Esta pode ser feita em *hardware* ou por um programa de controlo executado pelo processador central. A primeira opção é a única possível para um SDRD sem processador autónomo. Tem ainda a vantagem de tornar a reconfiguração transparente para o resto do ambiente em que o SDRD esteja integrado. A segunda opção é a mais flexível, já que o programa de controlo pode gerir as operações de reconfiguração segundo critérios arbitrariamente complexos e exe-

cutar funções que muito dificilmente poderiam ser implementadas *hardware*, como a geração dinâmica de novas configurações.

Para o sistema ReDiFlex escolheu-se a segunda opção, que, além de ser uma escolha natural para um SDRD tipo co-processador como é a placa H.O.T. Works„ permite garantir a flexibilidade do sistema, em particular a possibilidade de implementar aplicações do tipo IV (cf. pág. 60). Uma outra razão de ordem prática é o facto de o circuito FPGA usado (Xilinx XC6216) ter uma capacidade relativamente pequena, pelo que a inclusão de um controlador de configurações em *hardware* poderia reduzir consideravelmente o espaço disponível para implementar as funções de processamento de dados.

Assim, o sistema ReDiFlex deixa à aplicação a tarefa de activar explicitamente o a implementação do modelo funcional e de indicar as modificações dinâmicas desejadas. É de notar que estas tarefas são executadas de forma simples e sempre em termos do circuito conceptual (ver cap. 5). O sistema encarrega-se de determinar automaticamente como reconfigurar a infra-estrutura física para corresponder às mutações de alto nível e ao sequenciamento dos circuitos configurados resultantes da decomposição do modelo funcional.

4.2.3 Gestão integrada dos recursos físicos

O sistema ReDiFlex necessita de gerar a informação para configurar o co-processador a partir de uma descrição do modelo físico. Em princípio esse trabalho poderia ser efectuado por um programa exterior dedicado especificamente às tarefas de colocação e encaminhamento. Essa solução dificultaria a geração dinâmica de novas configurações e a imposição de restrições estruturais sobre o resultado final. Estas restrições são necessárias para evitar que modificações simples do circuito conceptual se reflictam em modificações extensas dos circuitos configurados (ver cap. 6).

O único programa disponibilizado pela companhia Xilinx para colocação e encaminhamento (designado por Xact6000) requer obrigatoriamente a intervenção do utilizador já que é comandado unicamente por uma interface gráfica. Esse facto constitui uma desvantagem significativa por obrigar o utilizador a trocar de ambiente de trabalho e a familiarizar-se com uma outra ferramenta. Outra desvantagem desta abordagem é que o programa produz a informação de configuração ao nível mais baixo (basicamente uma sequência de *bytes*), o que torna a identificação da organização final do circuito processado muito difícil[1]. Além disso, o programa Xact600 não contém nenhuma provisão especial para aproveitamento da reconfiguração dinâmica nem permite efectuar modificações incrementais a um circuito já processado. A única (e grande) vantagem da sua utilização seria, pois, a de evitar a implementação de algoritmos de colocação e encaminhamento, tarefa sempre delicada e complexa.

A consideração destes factos levou naturalmente à opção de implementar algoritmos para

[1]De facto existe também a possibilidade de aceder a um ficheiro privado dessa ferramenta que contém informação estrutural sobre o circuito; contudo o seu formato é muito complexo e não está documentado oficialmente.

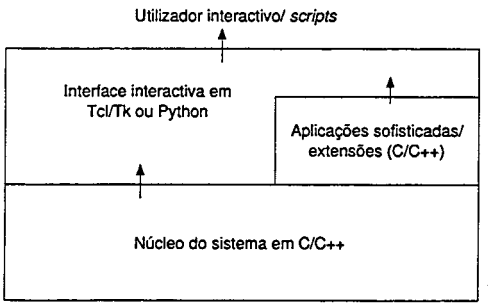


Figura 4.3: Ambiente dual de desenvolvimento.

colocação e encaminhamento no próprio sistema ReDiFlex, o que permite controlar completamente a utilização dos recursos físicos e aproveitar toda a informação recolhida durante as fases de colocação e encaminhamento (que é muito útil quando se pretende proceder a modificações parciais, como é o caso). A maior complexidade da implementação vem largamente compensada pela flexibilidade acrescida e pela maior liberdade de acção.

4.2.4 Ambiente de programação

Como já foi referido anteriormente, o sistema ReDiFlex pretende ser uma ferramenta interactiva. Isto significa que o utilizador deve ser capaz de construir a sua aplicação por pequenas etapas, refazer ou corrigir partes da aplicação e obter informações sobre o estado do sistema de uma forma simples, uniforme e directa. Em particular, é desejável evitar longos ciclos de compilação (para o programa a executar no processador central) ou de geração de configurações (para o co-processador reconfigurável).

Uma possibilidade consiste em criar um ambiente de raiz, eventualmente com possibilidades de programação fornecidas por uma linguagem embebida como Python [Beazley e Van Rossum, 1999] ou Tcl [Ousterhout, 1994]. Um ambiente deste tipo é tipicamente constituído por um conjunto de algoritmos e estruturas de dados implementados em linguagem C ou C++, que depois podem ser combinados da forma que o utilizador desejar através de programas na linguagem hospedeira (que neste contexto são geralmente designados por *scripts*). A situação encontra-se ilustrada na fig. 4.3.

Nesta abordagem existe uma divisão clara entre o núcleo, implementado numa dada linguagem (p.ex. a linguagem C), e os programas do utilizador. Se a separação for feita com cuidado, o utilizador clássico terá às suas ordens um sistema versátil. Contudo, qualquer necessidade que não tenha sido antecipada pelo autor do núcleo, levará à necessidade de reconstruir a ferramenta para incluir os novos serviços. O implementador de ferramentas, por sua vez, actuará num ambiente completamente diferente, no qual não pode beneficiar da interactividade

disponibilizada ao utilizador; em contrapartida terá acesso irrestrito a todo o sistema. Para além disso, existe a necessidade de trabalhar com duas linguagens de programação.

Uma solução melhor, pelo menos no contexto da produção de um protótipo, consiste em usar um ambiente de programação interactivo comum para utilizador clássico e implementador de ferramentas. Foi essa a opção tomada para o sistema ReDiFlex, que foi implementado na linguagem Common Lisp [Steele, 1990], uma linguagem normalizada [ANSI Common Lisp] com excelentes sistemas de desenvolvimento interactivos (tanto comerciais como do domínio público), e que dispõe de um conjunto de características que apoiam explicitamente o desenvolvimento incremental de aplicações. A sua adopção permite usar apenas uma linguagem para implementação do sistema, especificação dos circuitos conceptuais e desenvolvimento de aplicações. Certas tarefas, como a gestão de bibliotecas de geradores de componentes, podem ser feitas com recurso a meios já definidos na própria linguagem.

Como as implementações de Common Lisp dispõe de um ambiente sofisticado de desenvolvimento interactivo, este é directamente aproveitado para servir de interface do sistema ReDiFlex com o utilizador, com a vantagem adicional de fornecer um ambiente de programação sofisticado sem requerer da parte do implementador um dispêndio de esforço numa área (interface homem-máquina) não directamente relacionada com reconfiguração dinâmica. Este tipo de ambiente não é exclusivo da linguagem Common Lisp; a linguagem Smalltalk, p.ex., também está geralmente associada a um ambiente de desenvolvimento interactivo suficientemente sofisticado para suportar sem esforço um programa com as características do sistema ReDiFlex.

O sistema ReDiFlex foi desenvolvido com uma versão de Common Lisp do domínio público designada por CLISP[2]. Esta implementação foi escolhida por estar disponível para sistemas operativos diferentes. O protótipo começou por ser desenvolvido em Windows 95, mas a versão final apenas foi usada sob o sistema operativo GNU/Linux. Contudo a implementação não contém nada de específico a este sistema operativo e, se necessário, poderia ser usada em Windows 98 ou Windows 2000, desde que estejam disponíveis os controladores de dispositivos necessários. A implementação actual pode ser facilmente usada em outros ambientes Common Lisp. A única excepção é o (pequeno) módulo que implementa a interface com o controlador de dispositivo (*device driver*). Como este deve ser invocado por rotinas em linguagem C/C++, são necessários cuidados especiais para o usar a partir de Common Lisp. A maneira de o fazer varia, já que cada implementação de Common Lisp tem as suas próprias extensões para invocar rotinas escritas em outras linguagens.

A principal desvantagem associada ao emprego de Lisp[3] no protótipo do sistema ReDiFlex

[2]O sistema CLISP foi desenvolvido por Bruno Haible e Michael Stoll e está disponível em `http://clisp.cons.org/`

[3]Lisp é a segunda linguagem mais antiga em uso comum actualmente (apenas a linguagem Fortran é mais antiga). É interessante verificar que tanto uma como outra foram evoluindo ao longo dos anos, embora preservando as principais características que presidiram à sua concepção. Common Lisp é, de entre os muitos dialectos de Lisp existentes, um dos poucos que se encontra normalizado oficialmente e, na prática, serve de base a quase todos os modernos sistemas de desenvolvimento disponíveis comercialmente.

é o facto de ser uma linguagem pouco usada, o que pode limitar o universo de utilizadores potenciais. Para o protótipo actual este facto não foi considerado importante o suficiente para negar as vantagens da utilização da linguagem.

4.3 Organização geral

A organização de qualquer sistema de *software* complexo deve seguir o princípio geral de manter a complexidade dentro de limites através da divisão do sistema em módulos bem definidos. A estrutura mais simples obtem-se organizando esses módulos por camadas. Numa implementação ideal cada camada deve usar apenas os serviços providenciados pela camada imediatamente inferior para implementar novos serviços, que, por sua vez, servem de base à camada superior. A implementação de cada módulo deve usar uma interface pré-estabelecida para se manter independente dos detalhes internos dos outros módulos.

Ao promover uma clara separação de responsabilidades e a independência dos detalhes internos de cada módulo (abstracção), esta estratégia torna a implementação mais robusta, porque permite circunscrever o feito de erros da implementação ao módulo em que ocorrem. Este tipo de organização também favorece a flexibilidade de implementação, porque modificações na implementação interna de um módulo não se repercutem no correcto funcionamento dos demais.

A organização por camadas apresenta algumas desvantagens, especialmente durante a fase de concepção e implementação inicial. Em particular, nestas fases pode ser difícil fixar as interfaces entre as camadas, já que a identificação de novos requisitos e necessidades não antecipadas podem exigir reestruturações relativamente profundas. Por outro lado, podem surgir situações em que a implementação de um dado serviço possa ser feita mais eficientemente com recurso directo a outras camadas que não a imediatamente inferior. Muitas vezes a (re)definição de interfaces apropriadas permite contornar esse problema; nos outros casos é necessário optar por uma organização modular mais complexa.

A implementação do sistema ReDiFlex teve duas grandes fases. Durante a concepção usou-se uma abordagem de cima para baixo (*top-down*), partindo dos objectivos gerais e dos requisitos associados para uma definição dos serviços concretos a prestar ao utilizador. De seguida procedeu-se ao refinamento progressivo da especificação, o que permitiu definir os grandes módulos do sistema e as características gerais das respectivas interfaces.

Durante a fase de implementação procedeu-se de baixo para cima (*bottom-up*), implementando primeiro os serviços de baixo nível (acesso à infra-estrutura reconfigurável) e implementando sucessivamente os módulos superiores. Desta forma assegurou-se sempre a possibilidade de testar os serviços à medida que iam sendo implementados. As opções de implementação tomadas durante esta fase puderam ser tomadas à luz das necessidades antecipadas dos módulos superiores. À medida que se prosseguia na implementação, definiam-se os deta-

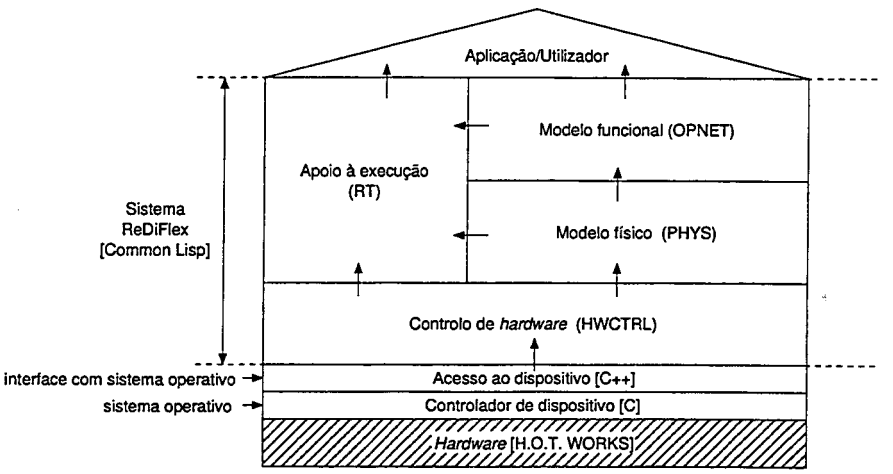


Figura 4.4: Principais módulos do sistema ReDiFlex.

lhes das interfaces entre módulos, sempre de maneira a cumprir os requisitos identificados na primeira fase. Naturalmente foi necessário alterar e re-avaliar as opções tomadas na primeira fase. Na prática estas modificações não levaram a alterações significativas da estrutura inicial, embora tenham originado muitas modificações de pormenor.

Os principais módulos do sistema ReDiFlex estão ilustrados na figura 4.4. Trata-se essencialmente de uma organização por camadas, com a excepção do módulo de apoio à execução. A interface com a aplicação é assegurada por dois módulos, o de apoio à execução (RT) e o de modelização funcional (OPNET). Como a interface está integrada no ambiente interactivo de Lisp, constitui também implicitamente a interface com o utilizador/programador de aplicações. Todos os serviços disponibilizados para as aplicações podem ser usados interactivamente graças à sua integração no ambiente de Lisp. Os serviços disponibilizados através desta interface suportam as seguintes actividades:

Especificação funcional Os modelos funcionais mutáveis que descrevem as operações a implementar em *hardware* podem ser construídos incrementalmente.

Activação e mutação Para cada modelo funcional, o sistema cria uma interface que deve ser usada pela aplicação para passar dados ao circuito e obter os resultados produzidos. Da perspectiva do utilizador/aplicação, tudo se passa como se a implementação em *hardware* fosse uma função da linguagem hospedeira. Também é criada uma interface para permitir a alteração dinâmica do circuito.

Gestão de bibliotecas de componentes Os modelos funcionais são compostos por componentes básicos. Os respectivos geradores estão agrupados em bibliotecas que o utilizador

pode gerir à sua vontade, recorrendo aos serviços do ambiente de programação.

Introspecção O sistema permite que a aplicação ou o utilizador tenham acesso à informação sobre a implementação automática em *hardware* e sobre os recursos usados.

Depuração e traçagem A interface fornece igualmente serviços de apoio à depuração da aplicações e traçagem das operações executados em *hardware*. Estes serviços destinam-se a ser usados interactivamente e não a partir de outras aplicações.

Os serviços disponíveis na interface superior são implementados nos módulos de apoio à execução e de construção de modelos de alto nível. O módulo RT é responsável pela implementação dos mecanismos necessários para suportar a interface de activação e mutação. Em particular, este módulo está encarregado de determinar as operações de reconfiguração necessárias para proceder às mutações exigidas pelo utilizador ou aplicação, bem como de proceder às operações de sequenciamento de configurações. Este módulo é descrito em último lugar (cap. 8) por depender de quase todos os outros.

O módulo OPNET (capítulo 5) é responsável pela gestão dos modelos funcionais, incluindo a construção e verificação da sua representação interna. O módulo também é responsável pelo mapeamento entre o domínio conceptual e a representação da organização física do circuito.

O módulo PHYS (capítulo 6) é responsável pela descrição interna do nível físico do circuito. Este módulo mantém um modelo conceptual da infra-estrutura de *hardware* que lhe permite fazer a gestão de todos os recursos físicos (células, interligações, registos acessíveis, etc.). Para isso inclui geradores de blocos básicos, que, para cada componente a instanciar em *hardware*, produzem a respectiva descrição física (terminais, dimensões físicas, registos internos, etc.) e a informação de configuração. As rotinas de colocação e encaminhamento também pertencem a este módulo e permitem obter toda a informação necessária para configurar a infra-estrutura física a partir da especificação topológica produzida pelo módulo OPNET.

O módulo HWCTRL (capítulo 7) está encarregado de controlar a infra-estrutura física, incluindo:

- inicialização de *hardware* (quer da placa H.O.T. Works, quer do circuito FPGA em particular);
- operações de leitura/escrita (para configuração ou acesso aos elementos de memória internos dos circuitos implementados);
- controlo dos sinais de relógio e de limpeza dos registos;
- descarga das configurações para o circuito FPGA;

O acesso básico à infra-estrutura de *hardware* é implementado por uma fina camada escrita em linguagem C++ e fornecida pelo fabricante (Xilinx). A versão utilizada no sistema ReDiFlex foi adaptada ao sistema operativo Linux por Adam Donlin da Universidade de Edinburgo,

Reino Unido. O acesso dessa rotinas à infra-estrutura física é mediado por um controlador de dispositivo inicialmente produzido por Michael Wirthlin e Peter Bellows de Brigham Young University, USA, alterado por Adam Donlin para suportar a placa H.O.T. Works. A sua utilização no sistema ReDiFlex requereu algumas modificações para eliminar problemas existentes no controlo do sinal de relógio.

4.4 Cenários de utilização

Qualquer sistema de apoio a uma tarefa necessita de ter bem caracterizados os seus utilizadores e os objectivos com que estes o vão usar. O sistema ReDiFlex está orientado para um utilizador que o queira usar na criação de aplicações baseadas em reconfiguração dinâmica, sem ter de se ocupar com os detalhes específicos da infra-estrutura. O utilizador tanto pode querer usar interactivamente os serviços disponibilizados pelo protótipo, como construir aplicações que deles façam uso. Este é o perfil de utilizador "clássico" que se teve em mente ao definir as características das ferramentas de apoio e os serviços a disponibilizar.

No cenário de utilização mais comum do sistema ReDiFlex, este utilizador começa por definir um modelo funcional de alto nível do processamento que o co-processador deve efectuar. Esse modelo é baseado em elementos parametrizáveis, cujas características podem ser modificadas posteriormente durante a execução da aplicação. O sistema gera automaticamente uma implementação associada, o que é um processo complicado, mas totalmente automatizado pelo sistema ReDiFlex. Durante esse processo, o modelo ou a implementação resultante podem ser decompostos para satisfazer as limitações de recursos impostas pelo co-processador. No fim desse processo, são produzidas as funções de interface com a implementação. Para o utilizador, essa funções são exactamente iguais a quaisquer outras de Common Lisp; a diferença reside no facto de a invocação das funções de interface envolver a tratamento da informação pelo co-processador reconfigurável. Todas as tarefas de reconfiguração e transferências de dados são realizadas de forma transparente pela função de interface. (De facto, o sistema ReDiFlex suporta várias políticas diferentes de gestão de *hardware*, pelo que é possível ter várias funções de interface para a mesma implementação.)

Entre as invocações das funções de interface, o utilizador pode modificar os valores dos parâmetros dos elementos do modelo funcional, o que resulta na criação, automática e transparente, de uma nova implementação (em geral, apenas os elementos alterados são reconfigurados, não sendo necessário proceder à reconfiguração completa do circuito). As invocações seguintes das funções de interface usam a nova implementação.

Para efeitos de depuração, o utilizador pode seguir passo a passo a operação das rotinas de gestão de *hardware* invocadas via funções de interface, bem como inspeccionar interactivamente tanto o estado da infra-estrutura reconfigurável como do sub-sistema de gestão de *hardware*.

Os elementos a usar na construção do modelo são representados por objectos regulares de Common Lisp, pelo que os meios do ambiente de programação são usados para proceder à sua gestão. As definições necessárias para a criação destes elementos são assim agrupadas em pacotes (*packages*), o mecanismo normalmente usado em Common Lisp para agrupar definições relacionadas entre si. A gestão das bibliotecas de elementos é feita usando directamente os serviços de Common Lisp para carregar novos pacotes ou remover aqueles que já não são utilizados.

Como veículo de investigação na utilização de reconfiguração dinâmica, o sistema ReDiFlex acomoda ainda outros tipos de utilizador. Em particular, também deve suportar aqueles que pretendem implementar formas alternativas de abordar a reconfiguração dinâmica, de implementar novas ferramentas de apoio ou, simplesmente, de fornecer outros algoritmos para alguma das tarefas já existentes. É para este tipo de utilizador, que poderíamos referenciar genericamente de "implementador de ferramentas", que a flexibilidade e extensibilidade do sistema são mais importantes.

Este segundo tipo de utilizador está naturalmente interessado na organização interna do sistema e nas interfaces entre as suas diversas camadas. Para ele é muito importante que exista uma distribuição clara de responsabilidades entre os diversos módulos e que os protocolos associados às interfaces entre estes sejam respeitados. Deste modo será mais fácil alterar aspectos específicos da implementação (p.ex. para obter melhor desempenho) ou providenciar novas capacidades (p.ex. ligação a sistemas externos de projecto assistido por computador ou generalizar o modo de especificar a funcionalidade dinâmica).

Existe ainda um terceiro tipo de cliente, que pode ser encarado como uma versão mais restrita do "implementador de ferramentas": o criador de bibliotecas de componentes. Como outros sistemas de projecto assistido por computador, o sistema ReDiFlex permite construir circuitos reconfiguráveis dinamicamente a partir de componentes básicos; estes são criadores por geradores de componentes. A interface entre os geradores de componentes e o resto do sistema está claramente definida, de maneira a ser possível acrescentar definições de novos componentes sem modificar o resto do sistema. A definição de novos elementos divide-se conceptualmente em duas etapas relacionadas entre si: (i) a definição dos elementos de alto nível a usar no modelo, e (ii) a definição dos componentes que constituem a respectiva implementação. Para qualquer das tarefas existem regras bem definidas e relativamente simples, que são explicadas nos capítulos 5 e 6. A definição de novos geradores requer apenas um conhecimento reduzido da organização global do sistema, mas exige um conhecimento aprofundado das características da infra-estrutura física. Logo que as definições apropriadas sejam integradas no sistema, o utilizador pode usar imediatamente os novos elementos nos seus modelo.

4.5 Resumo

ReDiFlex é um sistema de apoio ao desenvolvimento interactivo de aplicações que usem reconfiguração dinâmica (co-processador parcialmente reconfigurável) e um veículo para a investigação nesta área. A presente versão oferece ao utilizador um modelo abstracto explícito baseado na especificação funcional do comportamento a implementar na infra-estrutura reconfigurável.

A partir da descrição abstracta o sistema é capaz de gerar automaticamente tudo o que é necessário para implementar a funcionalidade em *hardware*, incluindo decomposição do circuito conceptual, geração da informação de configuração (incluindo para tal as operações de colocação e encaminhamento) e sequenciamento das operações. O sistema também gera uma interface para a aplicação/utilizador que permite usar a implementação do circuito como uma simples função da linguagem hospedeira.

Mutações do modelo abstracto são repercutidas dinamicamente sobre a implementação física. A especificação das mutações é feita explicitamente pelo utilizador que utiliza o circuito, o que permite um controlo arbitrariamente complexo das condições de reconfiguração. O sistema inclui ainda suporte para a depuração de aplicações e traçagem da respectiva execução.

A implementação tem uma organização que contempla as necessidades do "implementador de ferramentas" que queira usar o sistema para experimentar novos modelos de utilização ou alterar os algoritmos usados nas diversas sub-tarefas. Como o protótipo aproveita as capacidades interactivas do ambiente hospedeiro (Common Lisp), a tarefa de acrescentar novos módulos vem substancialmente simplificada.

5 Reconfiguração dinâmica: modelo funcional e ferramentas

A exploração da reconfiguração dinâmica precisa de ser efectuada num contexto que defina as regras de funcionamento e de utilização. Assim, nesse sentido, a actual implementação do sistema ReDiFlex suporta um modelo funcional da especificação do comportamento da infra-estrutura reconfigurável em que as mutações dos componentes obedecem a um conjunto de regras gerais bem definidas. Toda a utilização da infra-estrutura reconfigurável é feita em termos desse modelo.

Cada especificação tem por base uma rede acíclica de operadores parametrizáveis e com estado interno acessível. O modelo suporta a reconfiguração dinâmica ao permitir a modificação em tempo de execução de alguns parâmetros dos operadores; cada novo valor pode dar origem a uma nova implementação da funcionalidade genérica associada ao operador. Para além disso, o sistema suporta contentores, que são operadores que encapsulam . Em qualquer altura, o contentor pode ser reconfigurado para assumir a personalidade de um dos operadores associados. Estes mecanismos permitem uma variação controlada da implementação.

O modelo usado não é o único possível nem sequer o mais geral, mas representa um compromisso interessante entre simplicidade e versatilidade. Em particular, o modelo pode ser mapeado de uma forma efectiva para uma descrição de nível físico do circuito em tempo útil para a utilização interactiva do sistema. Do ponto de vista da flexibilidade, é possível usar o

sistema para criar aplicações do tipo IV (cf. secção 3.1), i.e., com sequenciamento variável de reconfigurações e criação dinâmica de novas configurações.

O capítulo começa com a descrição do modelo funcional (secção 5.1) e da sua utilização. A secção 5.2 descreve a representação interna do modelo, nomeadamente dos operadores (secção 5.2.1) e da sua interligação (secção 5.2.2), bem como dos algoritmos de decomposição usados (secção 5.2.3).

O sistema ReDiFlex extrai automaticamente do modelo funcional uma representação física para o circuito. A secção 5.3 mostra como se efectua o mapeamento entre o modelo funcional e a descrição dos componentes usados na implementação física. O resultado deste processo serve de ponto de partida para o processamento efectuado pelo módulo PHYS (capítulo 6).

5.1 Modelização funcional de circuitos reconfiguráveis dinamicamente

5.1.1 Modelo funcional

A especificação do comportamento da infra-estrutura reconfigurável usada no sistema ReDiFlex é baseada na descrição do fluxo de informação entre operadores (fig. 5.1). Cada operador é caracterizado pelo sua funcionalidade e pelos seus canais de transmissão de dados (unidireccionais), i.e., a sua interface com os demais operadores. O comportamento global é determinado pela interligação dos operadores, que deve respeitar a restrição de ser acíclica. Este modelo constitui uma variante dos grafos de fluxos de dados.

Os operadores são elementos abstractos que não correspondem necessariamente de forma directa a um componente físico. Devem antes ser encarados como especificando um certo comportamento, cuja implementação pode ser efectuada de várias maneiras. Esta distinção é importante porque permite manter o modelo a um nível abstracto mais elevado, deixando a questão da implementação concreta a cargo do sistema.

Em princípio, os operadores podem ter qualquer complexidade que se queira. Contudo, os que são disponibilizados pela biblioteca básica da actual implementação do sistema ReDiFlex são relativamente simples e têm uma correspondência relativamente directa com as respectivas implementações físicas. Como está descrito na secção 5.3 o mecanismo de mapeamento entre os domínios funcional e físico está implementado de forma suficientemente genérica para suportar correspondências complexas.

A restrição a redes acíclicas dirigidas de operadores foi tomada por várias razões:

1. É um modelo conceptualmente simples, já que evita a introdução do conceito de tempo (p. ex., sob a forma de um relógio global) para se obter uma definição operacional. Neste caso a definição é simples: os dados propagam-se pela rede a partir dos pontos de entrada ("fontes" de dados), são processados nos operadores para produzir novos dados

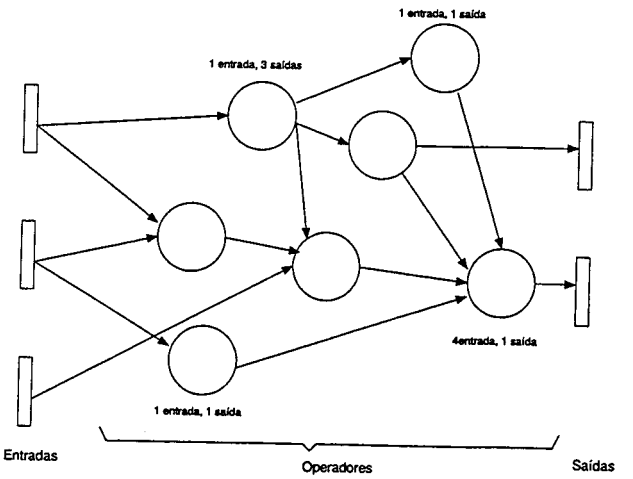


Figura 5.1: Modelo funcional básico.

que alimentam os operadores seguintes. Cada operador reage à modificação dos seus dados de entrada produzindo novos resultados. A propagação de informação termina nas saídas, que, do ponto de vista do modelo, constituem "poços" de dados.

2. O ordenamento de operações implícito na estrutura da rede serve de orientação para a implementação de nível físico. A restrição a circuitos com uma certa estrutura facilita a rápida construção automática dessa implementação, como se deseja para que a interactividade do sistema seja preservada.

3. Em certas situações é necessário decompor a rede acíclica em diversas sub-redes, de forma a que cada sub-rede possa ser totalmente implementada no circuito FPGA subjacente. Também essa tarefa vem simplificada no caso de redes acíclicas, já que se pode proceder de forma a que cada sub-rede seja acíclica e o grafo obtido pelo colapso de cada sub-rede também. Pelo contrário, no caso geral poderia existir retro-alimentação em cada sub-rede, entre sub-redes ou uma combinação das duas situações. As complicações adicionais não estão directamente relacionadas com reconfiguração dinâmica, pelo que foram deixadas de lado nesta implementação.

4. A restrição de aciclicidade é menos drástica do que pode parecer à primeira vista, já que os operadores podem ter qualquer complexidade desejada. Por exemplo, um operador cuja implementação seja uma máquina de estados (que, portanto, tem retro-alimentação interna) é perfeitamente aceitável. A única restrição quanto à complexidade do operador não advém do modelo mas da sua implementação, já que esta deve caber comple-

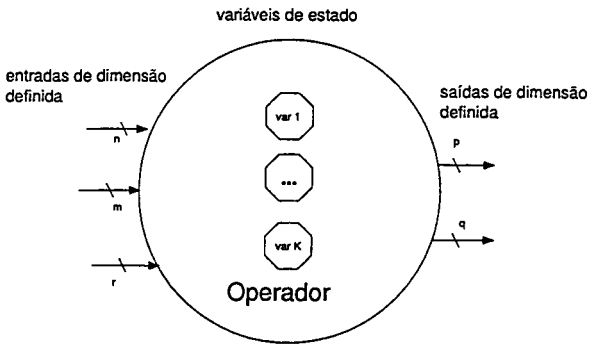


Figura 5.2: Estrutura conceptual de um operador.

tamente no co-processador reconfigurável. Por outras palavras, cada operador deve ser implementado de forma indivisível. Trata-se de uma restrição do sistema ReDiFlex que não é aparente a nível funcional.

Como o objectivo do modelo é descrever sistemas capazes reconfiguração dinâmica, é imprescindível que o modelo seja mutável. No actual modelo usado pelo sistema ReDiFlex as mutações só podem afectar os operadores; as interligações não podem ser modificadas. Esta restrição tem o objectivo de limitar as mutações a mudanças localizadas da implementação física, sem repercussões sobre toda a implementação. Desta forma é possível aproveitar a capacidade de reconfiguração parcial da infra-estrutura para executar as modificações rapidamente. Sem esta restrição, uma modificação pequena no modelo poderia levar a uma implementação física muito diferente que necessitasse de reconfigurações extensas, i.e. a capacidade de reconfiguração parcial não seria aproveitada directamente.

Esta restrição não implica necessariamente que seja difícil fazer modificações da estrutura do modelo. Do ponto de vista do utilizador é simples fazer uma cópia de um modelo e alterar tudo o que desejar, incluindo as interligações. Do ponto de vista do sistema trata-se então de um modelo novo, sem ligação com o modelo original e que dará origem a uma outra implementação física. Sendo considerados dois modelos independentes, o sistema não aproveita eventuais semelhanças; contudo, as modificações globais que é possível obter desta maneira inviabilizariam de qualquer forma o aproveitamento directo da reconfiguração parcial.

No modelo aqui adoptado, o operador constituiu o ponto focal das actividades de mutação, constituindo de facto o aspecto mais complexo de todo o modelo. Cada operador tem canais de entrada e saída de dados, todos unidireccionais e com capacidade fixa, i.e. a informação por eles transmitida tem um número fixo de bits (fig. 5.2). Cada operador tem ainda *parâmetros*, cujos valores permitem caracterizar melhor a funcionalidade de cada instância, e *variáveis de estado* que assumem valores representativos do respectivo estado.

Os parâmetros podem ser divididos em duas classes: os que correspondem a valores iniciais das variáveis de estado e os que determinam a implementação física. Em geral, o utilizador pode modificar o valor dos parâmetros de uma instância em qualquer altura. Se essa modificação tiver como consequência a alteração do valor de uma variável de estado, a modificação é propagada automaticamente para a implementação física, mesmo que o modelo esteja activo nesse momento (i.e. em execução na infra-estrutura de *hardware*).

As modificações de parâmetros que alteram a implementação física são registadas no modelo, mas só têm efeito quando o utilizador activar explicitamente a mutação. As mutações não são feitas automaticamente aquando da alteração dos parâmetros para permitir ao utilizador modificar vários parâmetros antes de ser gerada uma nova implementação.

Nem todas as alterações dos parâmetros que determinam a implementação física são admissíveis. Por exemplo, parâmetros que determinem a capacidade dos canais de informação não são possíveis, já que implicariam uma alteração das características da estrutura de interligações; os valores destes parâmetros só podem ser modificados enquanto não existir uma implementação física da rede de operadores. Por conseguinte, para cada operador, distinguem-se entre parâmetros alteráveis e inalteráveis; os parâmetros que correspondem a valores iniciais pertencem sempre à primeira categoria.

Durante a execução das operações especificadas por um modelo, o utilizador pode aceder ao valor das variáveis de estado de cada operador; isso não é feito através de operações sobre o modelo, mas pelas funções do módulo de apoio à execução (cap. 8) que controlam o tráfego de dados entre processador principal e co-processador reconfigurável.

O modelo usado no sistema ReDiFlex contempla três tipos de operadores:

regulares Os operadores que têm pelo menos uma entrada e uma saída são considerados regulares. A grande maioria dos operadores cai nesta categoria. Um operador pode ter uma ou mais variáveis de estado.

geradores Um operador sem entradas, mas com saídas é um gerador de dados. Um operador desta categoria deve produzir um novo valor à saída de cada vez que o modelo é activado. Um gerador de números aleatórios constitui um exemplo desta categoria.

acumuladores Um operador com entradas, mas sem saídas é um acumulador (ou redutor) porque "reduz" uma sequência de dados de entrada a um único valor (ou, em geral, a um número fixo de valores). Esse valor não é disponibilizado na saída (que é inexistente) mas numa variável de estado. Um exemplo deste tipo de operador é um somador-acumulador.

Para efeitos desta classificação não são considerados os elementos de entrada e saída do modelo, já que estes, conceptualmente, lhe são exteriores. A classificação precedente torna claro que o termo "operador" é usado num sentido lato, mais vasto que o sentido comum. O

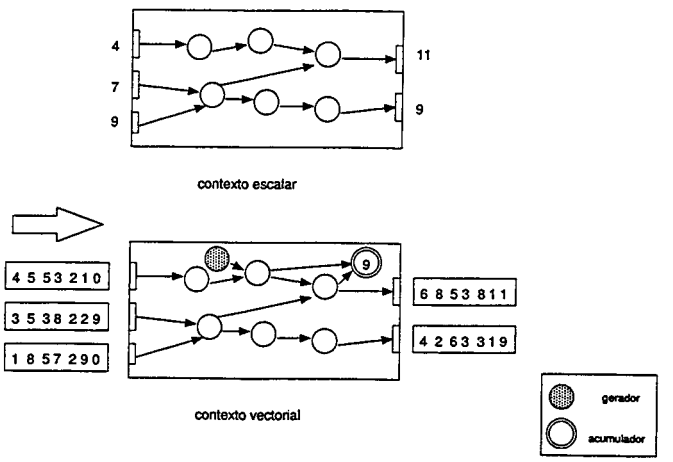


Figura 5.3: Contextos de utilização dos modelos: escalar e vectorial.

termo mais genérico "processo" também poderia ser usado para designar os nós do modelo; como este termo está frequentemente conotado com *software* optou-se pelo termo "operador".

Geradores e acumuladores apenas são úteis quando a implementação do modelo é usada para processar consecutivamente uma sequência de dados. Para distinguir as duas formas de usar um modelo é útil discriminar entre o contexto escalar ou vectorial dessa execução (fig. 5.3). No contexto escalar um conjunto de dados é passado ao co-processador que, após o processamento, retorna um conjunto de dados como resultado. No contexto vectorial, existe uma sequência de conjuntos de dados que vai dar origem a uma sequência de resultados.

Para um modelo que contenha apenas operadores regulares, os resultados obtidos no contexto vectorial são os mesmos que se obtêm por processamento repetido em modo escalar. Contudo, o conhecimento de que se está a operar num contexto vectorial permite ao sistema de apoio à execução utilizar uma forma mais eficiente do escalonamento de configurações, pelo que mesmo para este subconjunto é útil distinguir entre os dois contextos (cf. secção 8.2.2).

Por seu lado, geradores e acumuladores só são realmente úteis em contexto vectorial. Nada impede a sua utilização em contexto escalar, mas nesse caso são equivalentes, respectivamente, a uma entrada e a uma saída (eventualmente precedida de um operador regular).

As tabelas 5.1 e 5.2 apresentam alguns dos operadores do sistema ReDiFlex. Os operadores da primeira tabela são imutáveis, i.e. a respectiva implementação não pode ser modificada dinamicamente após a criação. Mesmo assim, o utilizador pode aproveitar o conhecimento prévio da aplicação para definir circuitos específicos mais eficientes. Por exemplo, para incrementar uma sequência de dados de um valor constante durante a aplicação é mais prático usar o operador `const-adder` que o operador genérico `adder`. A implementação resultante é

Nome	Entradas	Saídas	Parâmetros	Estado	Operação
and-2	A, B	Out	—	—	Out = A ∧ B
xor-2	A, B	Out	—	—	Out = A ⊕ B
selector	A[size], B[size], S	Out[size]	size	—	Out = S ? A : B
comparator	A[size], B[size]	Out	size, cmp-op	—	Out = (A <cmp-op> B) ? 1 : 0
const-comparator	A[size]	Out	size, const, cmp-op	—	Out = (A <cmp-op> const) ? 1 : 0
adder	A[size], B[size]	Out[size]	size	—	Out = A + B
const-adder	In	Out	size, const	—	Out = In + const
const-mult	In	Out	size, const	—	Out = In × const
counter-generator	—	Out[size]	size, dir,init-val*	val[size]	val = val ± 1
circular-shifter	—	Out	size	contents[size]	Out = lsb(contents); rot(contents)
accumulator	In[size]	—	size, acc-size, init-val*	sum[acc-size]	sum = sum + In
cond-accumulator	In[size], Cond	—	size, acc-size, init-val*	sum[acc-size]	if (Cond=1) sum = sum + In
max	In	—	size, init-val*	max-val[size]	max-val=max(max-val, In)
logic-accumulator	In[size]	—	size, log-op, init-val*	acc-val[size]	acc-val=acc-val <log-op> In
cond-counter	In	—	cnt-size, init-val*	count[cnt-size]	if (In=1) incr(count)
event-counter	A[size], B[size]	—	size, cmp-op, cnt-size, init-val*	count[cnt-size]	if (A <cmp-op> B) incr(count)
const-event-counter	A[size]	—	size, cmp-op, cnt-size, lim	count[cnt-size]	if (A <cmp-op> lim) incr(count)

Tabela 5.1: Alguns operadores imutáveis. O significado dos parâmetros é o seguinte: <size>, <cnt-size> e <acc-size> são números de bits; <const>, <lim> e <init-val> são inteiros constantes; <cmp-op> uma operação de comparação; <log-op> uma operação lógica; <dir> pode ser up ou down. O asterisco (*) marca os parâmetros mutáveis.

Nome	Entradas	Saídas	Parâmetros	Estado	Operação
mutable-logic-2	A, B	Out	log-op	—	Out = A <log-op> B
mutable-const-adder	In	Out	size*, const	—	Out = In + const
mutable-const-mult	In	Out	size, const	—	Out = In × const
mutable-comparator	A[size], B[size]	Out	size*, log-op	—	Out = (A <log-op> B) ? 1 : 0
mutable-const-comparator	A[size]	Out	size*, log-op, const	—	Out = (A <log-op> const) ? 1 : 0
mutable-counter-generator	—	Out[size]	size*, dir, init-val	val[size]	val = val ± 1
mutable-event-counter	A[size], B[size]	—	size*, cnt-size*, cmp-op, init-val	count[cnt-size]	if (A <cmp-op> B) incr(count)
mutable-const-event-counter	A[size]	—	size*, cnt-size*, cmp-op, lim	count[cnt-size]	if (A <cmp-op> lim) incr(count)
mutable-logic-accumulator	In[size]	—	size*, log-op, init-val	acc-val[size]	acc-val = acc-val <log-op> In

Tabela 5.2: Alguns operadores mutáveis.O significado dos parâmetros é o seguinte: <size>, <cnt-size> são números de bits; <const>, <lim> e <init-val> são inteiros constantes; <cmp-op> uma operação de comparação; <log-op> uma operação lógica. O asterisco (*) marca os parâmetros *não* mutáveis.

mais pequena e eficaz. Todos os parâmetros alteráveis mencionados nesta tabela estão necessariamente associados a valores iniciais das variáveis de estado dos respectivos operadores. Os valores dos parâmetros não estão restringidos a números inteiros; podem ser qualquer objecto válido de Lisp. Cada operador é responsável por validar o valor dos seus parâmetros. Por exemplo, o operador `comparator` aceita como valor para o parâmetro `comp-op` um dos símbolos `eq`, `neq`, `lt`, `gt`, `lte` e `gte` para indicar a relação desejada entre os dois valores de entrada.

A tabela 5.2 mostra alguns dos operadores mutáveis existentes no protótipo actual. Da perspectiva do utilizador a principal (mas pequena) diferença é que muitos mais parâmetros são modificáveis após a construção da implementação física. É precisamente pela simples alteração dos valores desses parâmetros que se especifica a nova configuração do operador. A modificação da implementação física associada é desencadeada pela invocação da função `mutate`.

Os operadores existentes na biblioteca inicial do sistema foram escolhidos para ilustrar os aspectos dos diferentes cenários de utilização. Por isso, são operadores simples, facilmente associáveis às respectivas implementações físicas. Esta última característica é vantajosa para testar a funcionalidade dos outros módulos. Contudo, deve-se notar que os operadores não estão limitados a funções simples. Por exemplo, é concebível ter um operador que implemente máquinas de estados especificadas em termos dos seus diagramas de transição. A implementação física de tal operador será em geral complexa, mas o seu uso está de acordo com o modelo que tem vindo a ser descrito. (Um operador deste tipo só terá uma utilização vantajosa em contexto vectorial, já que é necessário ter uma sequência de valores à sua entrada para efectuar as transições de estado.) Na escolha dos operadores não houve a preocupação de eliminar redundância lógica. Por exemplo, `const-comparator` e `cond-counter` podem ser usados para obter o mesmo efeito que `const-event-counter`. O objectivo foi simplesmente definir um conjunto de operadores úteis que permitissem avaliar as possibilidades do sistema.

O modelo funcional apresentado restringe a mutação dinâmica aos operadores, i.e., apenas estes podem ser modificados dinamicamente durante a execução. Os operadores básicos apresentados nas tabelas 5.1 e 5.2 preservam a unicidade funcional dos operadores, i.e. não suportam modificações da funcionalidade genérica do operador. Por exemplo, não é possível transformar um somador num comparador. Uma maneira de eliminar esta restrição consiste em criar um novo operador que conjugue as funções desejadas. Esta abordagem teria a desvantagem de levar a uma proliferação de novos operadores (um por cada combinação de operadores básicos que se pretendesse utilizar); além disso, deixaria o utilizador dependente do poder de antecipação do criador de bibliotecas. Por isso é importante fornecer um meio de o utilizador especificar essas novas combinações. No sistema ReDiFlex, essas combinações são modeladas por *contentores*, operadores mutáveis cuja implementação é delegada em outros operadores. A figura 5.4 ilustra este conceito: o contentor é uma "concha" ou "embrulho" (*wrapper*) em torno de outros operadores, dos quais apenas um pode estar presente em cada instante. Os operadores contidos devem ser compatíveis entre si: todos devem ter o mesmo

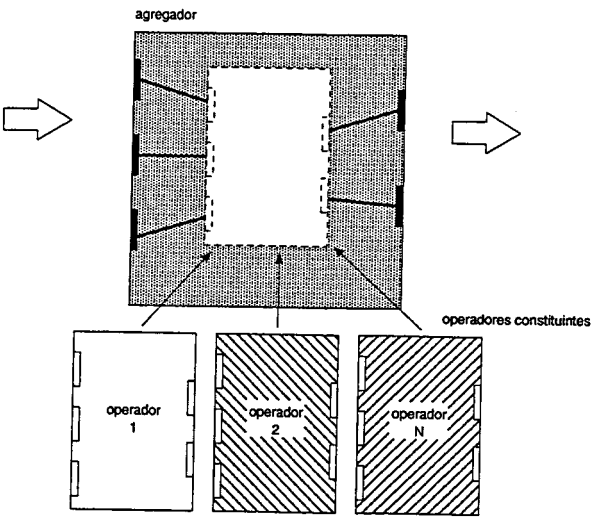


Figura 5.4: Modelo conceptual de operadores contentores.

número de entradas e saída, e para entradas e saídas correspondentes o número de bits deve ser igual. Não é necessário que os operadores contidos tenham as mesmas dimensões físicas.

A especificação de um contentor inclui, para além das especificações dos operadores constituintes, a definição do mapeamento de entradas e saídas entre estes. Na implementação actual do sistema ReDiFlex, os contentores não permitem acesso às variáveis de estado dos operadores (mas permitem a especificação de quaisquer valores iniciais necessários). Um contentor tem ainda um parâmetro modificável especial designado por `personality` que define, em cada instante, qual o operador interno activo, i.e. aquele cuja implementação se encontra dentro da "concha". A modificação da implementação de um contentor faz-se como todas as outras mutações: alteração do parâmetro `identity` e invocação posterior da função `mutate`. Nenhum operador simples (i.e., que não seja um contentor) pode ter um parâmetro chamado `identity`.

5.1.2 Construção e manipulação de modelos

O sistema ReDiFlex permite construir modelos funcionais que especificam o processamento a realizar pelo co-processador de duas maneiras. A mais simples consiste em utilizar a função `define-operator-network`, que cria uma rede de operadores e verifica a sua consistência. O resultado da utilização de `define-network-spec` é um objecto da classe `operator-network-spec`, que, a este nível de abstracção, deve ser encarada como um tipo opaco[1]. A outra maneira de construir uma rede de operadores é por aplicação das funções definidas na tabela 5.3. Estas

[1]Um tipo de dados é considerado *opaco* quando a sua representação interna não está disponível.

Invocação	Operação
`make-new-opnet` *id* ⇒ *opnet-spec*	Nova especificação chamada *id*.
`establish-opnet-input` *opnet-spec spec-list*	Acrescenta nova entrada.
`establish-opnet-output` *opnet-spec spec-list*	Acrescenta nova saída.
`establish-opnet-operator` *opnet-spec spec-list*	Acrescenta novo operador.
`establish-opnet-container` *opnet-spec spec-list*	Acrescenta novo contentor.
`establish-opnet-connection` *opnet-spec spec-list*	Acrescenta nova interligação.
`check-operator-spec-consistency` *opnet-spec*	Verifica a consistência da rede.

Tabela 5.3: Funções para construção e manipulação de modelos funcionais. O parâmetro *spec-list* indica uma lista com os elementos de especificação apropriados (os mesmos que são usados em `define-operator-network`.) As funções que não retornam valores definidos lançam uma excepção se não tiverem sucesso.

funções são usadas internamente para definir `define-operator-network`.

A utilização de `define-operator-network` tem a seguinte forma:

```
(define-operator-network 'rede-exemplo
  ...
  [especificação parte 1]
  ...
  [especificação parte N]
  ...)
```

No corpo de `define-operator-network` podem coexistir código Lisp e especificações dos componentes. Estas últimas distinguem-se por estarem entre parêntesis rectos. O código Lisp pode ser usado para qualquer processamento que se pretenda fazer durante a especificação.

Cada entrada da rede tem a forma

```
[:input <nome> :size <nº de bits>]
```

em que <nome> é um símbolo que identifica a entrada e <nº de bits> é um inteiro positivo. A especificação do tamanho é facultativa; o valor por omissão é 1.

As saídas são definidas de forma similar:

```
[:output <nome> :size <nº de bits>]
```

A instanciação de um operador tem a forma

```
[:operator <nome> <type> <parâmetro 1> <valor 1> ... <parâmetro N> <valor N>]
```

em que <nome> é um símbolo que identifica a ocorrência do operador, <type> é um símbolo que indica o respectivo tipo (p. ex., os tipos indicados nas tabelas 5.1 e 5.2). De seguida

pode surgir um número variável de parâmetros, cada um seguido do respectivo valor. Cada operador de um modelo deve ter um identificador diferente.

A especificação de um contentor tem a forma

```
[:encapsulate <nome> <operador 1> ... <operador N>]
```

em que <nome> é um símbolo que identifica a instância do contentor e <operador N> é a especificação no N-ésimo operador. Essa especificação tem a forma:

```
(<op-id> <type> <terminal-map>
        <parâmetro 1> <val 1> ... <parâmetro N> <val N>)
```

em que <op-id> é identificador do operador constituinte, <type> representa o respectivo tipo, e <terminal-map> mapeia as entradas e saídas do operador constituinte nas do contentor. Cada operador constituinte deve ter um identificador diferente. As entradas de um contentor têm sempre os nomes começados por In seguidos de um número de ordem (1 ... nº de entradas); uma regra similar vale para as saídas, mas usando o prefixo Out. Inicialmente, o operador activo (aquele cuja implementação é colocada dentro da "concha") é o que surge em primeiro lugar na especificação.

O mapeamento de terminais tem a forma

```
((<entrada 1> ... <entrada N>) (<saída 1> ... <saída M>))
```

em que <entrada N> é o nome (uma cadeia de caracteres) da entrada do operador constituinte que deve ser associada à N-ésima entrada do contentor. O significado de <saída N> é similar.

Por último, a definição das ligações entre operadores tem o formato

```
[:connect <fonte> <destino1> ... <destinoN>]
```

em que <fonte> e <destino1>...<destinoN> especificam os terminais (respectivamente, de saída e de entrada) interligados. Cada terminal é especificado por uma lista de três elementos:

```
(<op-id> <terminal-name> <bit-spec>)
```

em que <op-id> é o identificador associado a um operador da rede, <terminal-name> é o nome do terminal (uma cadeia de caracteres) e <bit-spec> especifica que parte dos terminais participa nesta interligação. Esta especificação pode ser omitida (indicando que todos os bits do terminal participam na interligação), ou pode ser um número inteiro positivo p (para designar o p-ésimo bit do terminal) ou ainda um par de números (p . q) que designa todos os bits entre o p-ésimo e o q-ésimo, inclusive.

A possibilidade de especificar a parte de um terminal que participa numa ligação existe apenas para satisfazer as necessidades potenciais da utilização de operadores mais "próximos" de *hardware* e pode ser ignorada na maioria das aplicações.

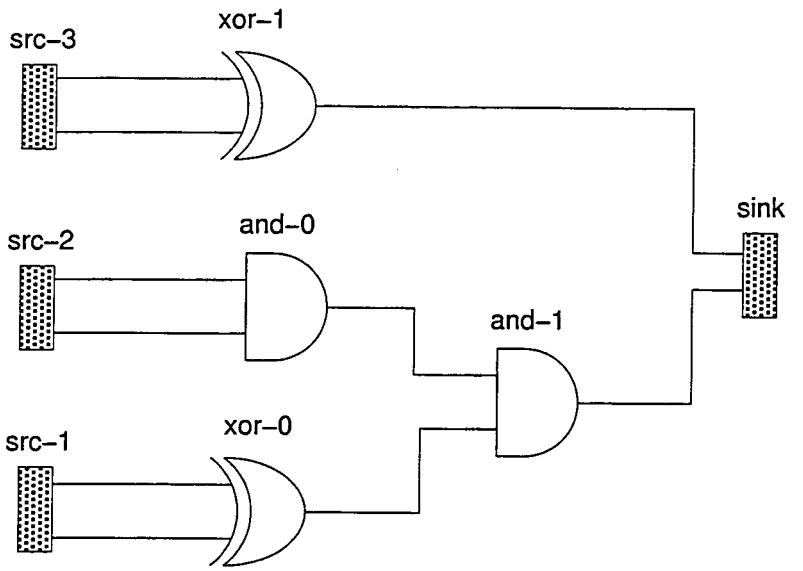


Figura 5.5: Diagrama do circuito `circuit-3`.

Exemplo: Um circuito com operações lógicas simples (fig. 5.5).

```
(define-operator-network circuit-3
     [:input 'src-1 :size 2]
     [:input 'src-2 :size 2]
     [:input 'src-3 :size 2]
     [:output 'sink :size 2]
     [:operator 'and-0 'and2]
     [:operator 'xor-0 'xor2]
     [:operator 'xor-1 'xor2]
     [:operator 'and-1 'and2]
     [:connect '(src-1 "Out" 0) '(xor-0 "A")]
     [:connect '(src-1 "Out" 1) '(xor-0 "B")]
     [:connect '(src-2 "Out" 0) '(and-0 "A")]
     [:connect '(src-2 "Out" 1) '(and-0 "B")]
     [:connect '(src-3 "Out" 0) '(xor-1 "A")]
     [:connect '(src-3 "Out" 1) '(xor-1 "B")]
     [:connect '(and-0 "Out") '(and-1 "B")]
     [:connect '(xor-0 "Out") '(and-1 "A")]
     [:connect '(xor-1 "Out") '(sink "In" 1)]
     [:connect '(and-1 "Out") '(sink "In" 0)]))  ⇒ operator-network-spec
```

Exemplo: Os modelos são representados por objectos de Lisp que, como todos os outros, podem ser retornados por funções ou usados como argumentos. Uma função que cria um modelo com um somador constante.

```
(defun make-simple-adder (nome dim constante)
  (define-operator-network nome
      [:input 'source :size dim]
      [:output 'sink :size dim]
      (format t "A criar um somador constante com ~a bits e constante=~a."
              dim constante)
      [:operator 'adder 'fixed-const-adder :size dim :const constante]
      [:connect '(source "Out") '(adder "In")]
      [:connect '(adder "Out") '(sink "In")]))

(setf modelo1 (make-simple-adder 'somador-1 6 19))
A criar um somador constante com 6 bits e constante=19.

(setf modelo2 (make-simple-adder 'outro-somador 12 1024))
A criar um somador constante com 12 bits e constante=1024.

(setf modelo3 (make-simple-adde 'ainda-outro-somador 4 2))
A criar um somador constante com 4 bits e constante=2.
```

Exemplo: Uma função que cria um modelo com um somador constante, fixo ou mutável.

```
(defun make-adder (nome dim constante mutavel-p)
  (define-operator-network nome
      [:input 'source :size dim]
      [:output 'sink :size dim]
      [:operator 'adder (if mutavel-p 'mutable-const-adder
                                  'fixed-const-adder)  :size dim :const constante]
      [:connect '(source "Out") '(adder "In")]
      [:connect '(adder "Out") '(sink "In")]))

(setf addfixo (make-adder 'somador 6 19 nil))
(setf addmut (make-adder 'somador-mutavel 12 1024 t))
```

Neste caso, a função `make-adder`, para além dos argumentos que especificam o nome, tamanho e a constante, recebe mais um que especifica qual o tipo de somador pretendido.

Exemplo: Definição de um modelo com um contentor (somador constante / operação lógica constante).

```
(setf exemplo-comb
      (define-operator-network 'exemplo
        [:input 'src :size 4]
        [:output 'sink :size 4]
        [:combine 'misc ('adder 'fixed-const-adder (("In") ("Out"))
                                :size 4 :const 2)
                        ('log1 'const-logic-func (("In'') ("Out"))
                                :size 4 :const #b1001 :op 'xor)
                        ('log2 'const-logic-func (("In") ("Out"))
                                :size 4 :const #b1101 :op 'xnor)]
        [:connect '(misc "Out") '(sink "In")]
        [:connect '(src "Out") '(misc "In")]))
```

Após a criação de uma rede, as características dos operadores podem ser modificadas com auxílio da função `set-parameter`, que por sua vez delega a implementação no método `set-param-value` apropriado. A representação interna de um operador pode ser obtida com `get-operator-by-name`. As instâncias de operadores pertencem a subclasses de `operator` (um tipo opaco). A função `mutate` é usada para actualizar a implementação física de um operador. Esta função não deve ser chamada directamente pelo utilizador de alto nível, porque a operação de reconfiguração deve considerar outros factores que estão fora da responsabilidade do operador individual. Para tratar de todos os factores associados à reconfiguração, o sistema de apoio à execução fornece a função `mutate-operator` (sec. 8.2.3). Para simplificar a utilização, `mutate-operator` também aceita uma especificação dos parâmetros a modificar antes de proceder à mutação. A definição desta função utiliza directamente o método `mutate` (depois de obter o objecto que representa o operador a partir do identificador respectivo), conforme mostra o seguinte fragmento:

```
(defun mutate-operator (opnet op-id &rest params)
 ...
 (let ((op (get-operator-by-name op-id (ops opnet))))
      ...
      (funcall #'mutate op params)
      ...))
```

A utilização de `mutate-operator` é muito simples:

Exemplo: Modificação de parâmetros e mutação de operadores (baseado nos exemplos precedentes).

```
(mutate-operator add-mut 'adder :const 8)
```

Exemplo: A modificação de um contentor é feita de maneira semelhante. O parâmetro `:personality` define qual o operador constituinte a ser activado.

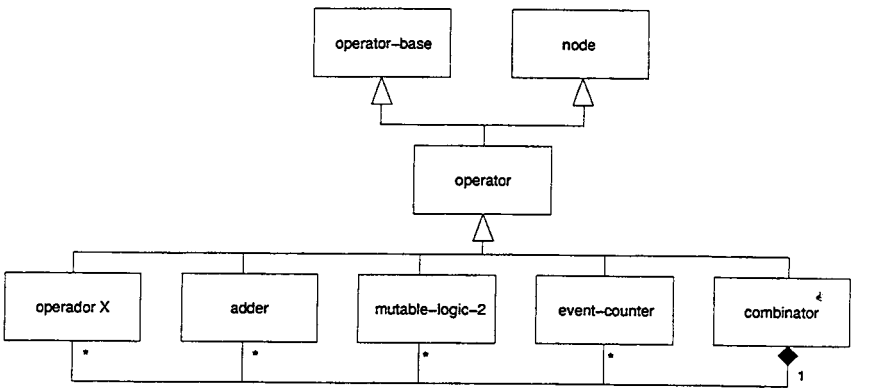


Figura 5.6: Hierarquia das classes que representam operadores.

```
(mutate-operator exemplo-comb 'misc :personality 'log1)
```

A activação da implementação de um modelo é feita como se de uma função normal de Common Lisp se tratasse. Embora a interface de utilização seja simples, o processamento executado por essas funções pode ser bastante complexo e depende das características internas da implementação física. As funções de activação e sequenciamento (que pertencem ao módulo de apoio à execução) são descritas no capítulo 8.

5.2 Representação do modelo funcional

Internamente, o sistema ReDiFlex usa três estruturas de dados principais para representar a informação associada a um modelo funcional. A classe `operator-network-spec` forma a base da representação do modelo completo. A correspondente implementação é representada pela classe `operator-network-impl`. Por sua vez, cada operador usado no modelo é representado por uma classe derivada de `operator`. Para o utilizador do sistema, apenas é importante saber que estas classes existem (porque tem de usar as respectivas instâncias), mas a respectiva estrutura interna é irrelevante. Já o criador de bibliotecas de operadores deve conhecer a estrutura interna da classe `operator`, porque precisa de definir sub-classes desta para representar os novos operadores.

5.2.1 Operadores parametrizados

Cada tipo de operador está associado a uma classe derivada de `operator`. A hierarquia está ilustrada na fig. 5.6. Notar que todos os contentores são representados por uma única subclasse, `op-container`, que delega todas as operações num dos seus operadores constituintes, o

operador "activo". Em consequência, todos os operadores apresentam uma interface uniforme para o resto do sistema.

A definição da classe operator é a seguinte:

```
(defclass operator-base ()
  ((id :reader id :initarg :id :type symbol
       :documentation "The identifier of the operator instance.")
   (inputs :accessor inputs :type op-input-spec
           :documentation "Names and sizes of input ports.")
   (outputs :accessor outputs :type op-output-spec
            :documentation "Names and sizes of output ports.")
   (params :accessor params :type op-param-spec
           :documentation "Named parameters and their ranges.")
   (blk :reader blk :type rblock
        :documentation "The physical block for this instance.")))

(defclass node ()
  ((number-of-predecessors :initform 0 :type pos-fixnum
                           :accessor number-of-predecessors
                           :documentation "How many links connect to this node.")
   (shadow-np :type pos-fixnum :initform 0
              :documentation "Copy of number-of-predecessors.")
   (level :accessor level :initform 0 :type pos-fixnum
          :documentation "The ASAP level of this node.")
   (clock-cycle :accessor clock-cycle :type pos-fixnum :initform 0
                :documentation "The clock cycle of the node .")))

(defclass operator (operator-base node) ())
```

A classe operator é derivada de operator-base e node. A primeira representa as características básicas de um operador, enquanto a segunda acrescenta as características adicionais que advêm da inclusão do operador numa dada rede. Para além de um identificador, cada operador tem uma especificação das suas entradas (inputs) e saídas (outputs), bem como de todos os parâmetros que aceita (params). A representação dos parâmetros inclui, para além do nome, um conjunto de atributos. Cada operador tem ainda associada a representação da correspondente implementação física (blk). Este campo, inicialmente vazio, será preenchido no decurso da criação automática da implementação física.

Os atributos de cada parâmetro são representados por instâncias da classe op-param-attrib:

```
(defclass op-param-attrib ()
  ((value :accessor value :initarg :value
          :documentation "The value for this parameter.")
```

```
(mutable :accessor mutable :initarg :mutable
         :initform nil type boolean
         :documentation "Can value change after freezing the implementation?")
(valid-fn :accessor valid-fn :initarg :valid-fn  :initform nil
          :documentation "Function that validates the parameter value.")
(update-fn :accessor update-fn :initarg :update-fn
           :initform #'default-op-param-update-once-fn
           :documentation "Function that updates the new parameter value.")))
```

O campo `value` especifica o valor do parâmetro; inicialmente encontra-se vazio. O campo booleano `mutable` especifica se o parâmetro pode ser alterado depois de construída a representação física do sistema. Os outros dois campos são funções que o sistema invoca para determinar se um novo valor é aceitável (`valid-fn`) e para efectivar a modificação do valor (`update-fn`). A primeira função é chamada (se existir, o que não é obrigatório) com o novo valor do parâmetro e com o valor actual de todos os outros parâmetros do mesmo operador. Desta forma, a função pode preservar as relações necessárias entre os parâmetros. Por exemplo, um `const-adder` com 6 bits não poderá ter uma constante superior a $2^6 - 1$.

A função `update-fn` é chamada pelo sistema para proceder à modificação do parâmetro (depois de este ser validado por uma chamada eventual a `valid-fn`). Esta função encarrega-se da alteração do valor do parâmetro bem como da propagação das respectivas consequências. Em particular, esta função propaga modificações dos valores iniciais das variáveis de estado até à implementação de *hardware* (se esta já existir na altura da alteração do parâmetro). O sistema já inclui algumas funções típicas para usar como valor de `update-fn`. Por precaução, o valor por omissão é uma função que apenas permite especificar um parâmetro uma vez (`default-op-param-update-once-fn`).

A inclusão de funções de validação/alteração próprias a cada operador evita que durante a concepção do sistema seja necessário prever todas as restrições possíveis ou todas as combinações admissíveis de valores dos parâmetros, bem como arranjar forma de as codificar. A alternativa adoptada é mais versátil e geral.

A classe `operator` herda parte das suas características de `node`. Esta classe representa aqueles aspectos de um operador que lhe advêm da sua inclusão numa rede. Em particular `number-of-predecessors` regista o número de outros operadores que estão ligados às suas entradas. O campo `level` indica a distância do operador às entradas primárias. Se considerarmos as ligações entre nós da rede como estabelecendo uma relação de precedência entre os nós, trata-se de estabelecer um ordenamento entre os operadores que seja compatível com as restrições impostas por essa relação (ordenamento topológico) . O campo `level` indica a posição do seu operador nessa ordem parcial (fig. 5.7). O algoritmo de ordenação topológica é um algoritmo clássico de complexidade polinomial (linear na soma do número de operadores e do número de interligações). Na literatura de síntese de alto nível o algoritmo também é conhecido como ASAP (*as soon as possible*), porque atribui a cada operador o nível mais baixo

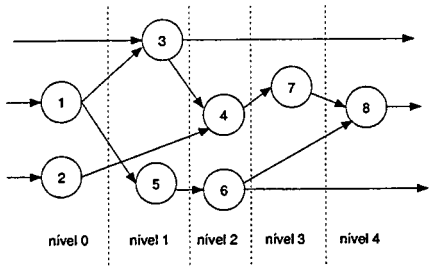


Figura 5.7: Nível de um operador.

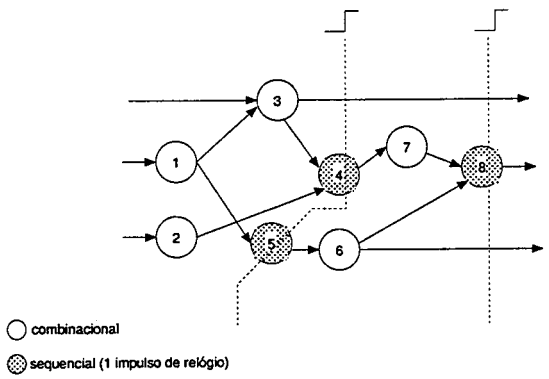


Figura 5.8: Propagação temporal de informação através do modelo.

compatível com as restrições de precedência [de Micheli, 1994]. O resultado final contém o menor número de níveis possível; além disso, cada nível contém o maior número possível de operadores em paralelo.

Conforme se verá mais adiante (secção 5.3 e capítulo 6) o sistema ReDiFlex usa o agrupamento por níveis para guiar a estrutura da implementação física. Para esta ser possível é preciso que sejam respeitadas algumas condições adicionais. Quando tal não acontece é necessário decompor a rede em várias sub-redes. Este tópico é discutido em mais detalhe na secção 5.2.3.

Do ponto de vista da implementação, os operadores usados no sistema ReDiFlex distinguem-se em puramente combinacionais e sequenciais. Na versão actual do sistema, todos os operadores sequenciais necessitam apenas de um ciclo para executarem. A restrição a operadores uni-ciclo permite simplificar alguns detalhes da implementação (nomeadamente, mas não exclusivamente, no sistema de apoio à execução), sem restringir demasiado os operadores utilizáveis. Para efeitos deste protótipo, considerou-se que as simplificações assim obtidas

Método	Operação
initialize-instance	Inicialização do operador, incluindo atributos.
set-param-value	Modificação dos valores de um parâmetro.
check-operator-parameters	Verificar se parâmetros necessários estão definidos e se têm valores coerentes.
calculate-port-size	Determinar o número de bits dos portos do operador.
generate-operator-block	Gera a implementação física do operador.
operator-input-mapping	Posição dos terminais de entrada da implementação física.
operator-output-mapping	Posição dos terminais de saída da implementação física.
convert-parameters	Conversão dos parâmetros do operador em parâmetros do bloco físico.
mutate	Desencadeia a modificação da implementação física.

Tabela 5.4: Métodos definidos para cada operador.

justificam a restrição adoptada.

Para determinar a evolução temporal da rede (em termos de ciclos de relógio), o campo `clock-cycle` regista o ciclo de relógio em que o operador pode executar (fig. 5.8). Todos os dados de entrada de um operador devem estar disponíveis do mesmo ciclo de relógio. Como a evolução temporal não faz parte do modelo funcional (nem tão-pouco qual o tipo de implementação usado para cada operador), o sistema ReDiFlex introduz elementos de atraso na implementação, por forma a garantir esta condição. A determinação do ciclo de relógio em que cada operador deve produzir os seus resultados é efectuado por uma variação do algoritmo usado para calcular os níveis. Neste caso os "níveis" são os ciclos de relógio e só existe mudança de nível quando os dados passam por operadores com implementação sequencial.

Os algoritmos que determinam o nível e o ciclo de relógio para o operador são mais fáceis de implementar se for possível alterar o campo `number-of-predecessors`; para poder recuperar o valor original é guardada um cópia deste valor em `shadow-np`.

A manipulação dos operadores é feita via um conjunto de métodos que devem ser especializados para cada novo operador. A tabela 5.4 apresenta sumariamente estes métodos. Para todos eles existe uma implementação por omissão.

Estes métodos são o único meio que o sub-sistema encarregado de manter o modelo funcional usa. Uma parte desses métodos é usada para efectuar a gestão dos parâmetros, enquanto os outros são usados para gerir a implementação física. A única excepção é `initialize-instance`, que é usada para adaptar o processo de inicialização às necessidades de cada operador.

Todos os operadores usados no sistema ReDiFlex são descendentes, directos ou indirectos, da classe `operator`. Também os contentores são implementados como um operador "virtual" da classe `op-container`. A sua definição é a seguinte:

```
(defclass op-container (operator)
  ((sub-operators :accessor sub-operators
                  :type combinator-op-map
                  :initform (make-empty-combinator-op-map)
```

```
                :documentation "Map id's to simple operators")
  (active-op :accessor active-op
             :type operator
             :documentation "The active operator.")))
```

Esta classe mantém um mapeamento entre identificadores de operadores e os objectos correspondentes (o campo sub-operators) e uma indicação de qual o operador activo (que deve ser um dos especificados no mapeamento). O mapeamento inclui, para além do sub-operador propriamente dito, informação sobre as correspondências de terminais entre o operador virtual (especificadas pelos campos inputs e outputs herdados de operator) e os sub-operadores.

Na sua grande maioria, os métodos de op-container limitam-se a invocar os métodos correspondentes dos operadores constituintes. Por exemplo:

```
(defmethod generate-operator-block ((op op-container))
  (with-all-operators (subop (sub-operators op))
    (generate-operator-block subop)))

(defmethod set-param-value ((opc op-container) param val)
  (set-param-value (active-op opc) param val))
```

No sistema ReDiFlex, cada contentor vai corresponder a um componente físico que implementa o contexto físico para os componentes que implementam os elementos constituintes. Esse componente é guardado no campo blk (herdado da classe operator).

Qualquer operador mutável deve ser capaz de proceder a operações de alteração da sua implementação física. Para esse efeito o sistema ReDiFlex define o método mutate:

```
(defmethod mutate ((op operator) params)
  "Update parameters and change implementation."
  (dolist (p (group params 2))
    (set-param-value op (first p) (second p)))
  (let ((blk-params (convert-parameters op)))
    (funcall #'mutate (blk op) blk-params)))
```

Este método é muito simples porque está construído sobre os métodos da tabela 5.4. Primeiro invoca repetidamente set-param-value para alterar os parâmetros do operador; de seguida converte esses parâmetros para parâmetros do gerador de implementações associado à representação física (campo blk); por fim, invoca o método mutate associado à implementação física. Este último encarrega-se de todas as modificações físicas necessárias. Os detalhes deste processo são descritos na secção 6.2.2.

A classe op-container precisa de uma definição especial do método mutate, que se limita a delegar o trabalho no método associado ao operador activo:

```
(defmethod mutate ((opc op-container) params)
  (mutate (active-op opc) params))
```

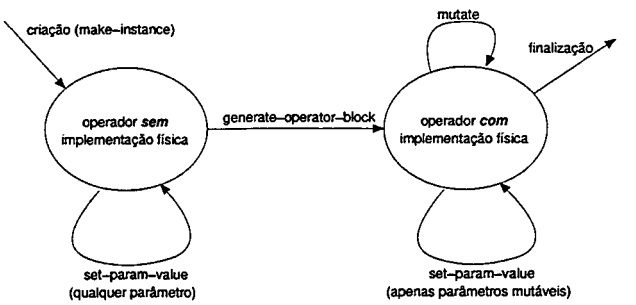


Figura 5.9: Ciclo de vida de um operador.

As instâncias das classes derivadas de `operator` têm um ciclo de vida bem definido (fig. 5.9) que determina as operações que são possíveis. No estado inicial, após a criação, os parâmetros podem ser modificados livremente; nesta fase ainda não existe uma implementação física. Quando essa é gerada (por invocação do método `generate-operator-block`), certos parâmetros passam a estar fixados (a sua modificação corresponderia a modificações dos terminais da implementação). Esta fase é caracterizada pelo facto de o campo `blk` não estar vazio. Caso se trate de um operador mutável é possível modificar dinamicamente a sua implementação (método `mutate`). Parâmetros que correspondam a variáveis de estado do operador podem ser sempre modificados.

Exemplo: A especificação do operador `const-mutable-adder`

Para mostrar o trabalho envolvido na definição de um operador, apresenta-se de seguida a definição completa de `const-mutable-adder`, um operador cujo resultado é a soma da sua única entrada com uma constante. Essa constante pode ser modificada dinamicamente: para cada valor o sistema gera uma nova configuração do somador. Este circuito não é implementado usando um registo e um somador de duas entradas; em vez disso, os circuitos de adição de cada andar são modificados para implementarem a adição com 0 ou 1, conforme o valor correspondente a esse andar da representação binária da constante.

A classe `mutable-const-adder` é derivada directamente de `operator`. O método de inicialização `initialize-instance` define a entrada e saída do operador (respectivamente `"In"` e `"Out"`), bem como os dois parâmetros `:size` e `:const`. O primeiro indica o número de bits do somador (e consequentemente da entrada e da saída); o segundo representa o valor constante. Apenas o segundo parâmetro é mutável, conforme indicado pela chamada à função `op-param-toggle-mutability`. Para cada parâmetro são também definidas as respectivas funções de validação e de alteração. Neste caso particular, o parâmetro `:size` tem de estar entre 0 e 32 e o valor de `:const` deve estar entre 0 e $2^{:const} - 1$.

```
(defclass mutable-const-adder (operator) ())
```

```
(defmethod initialize-instance :after ((instance mutable-const-adder) &rest initargs)
  (declare (ignore initargs))
  (setf (slot-value instance 'inputs) (make-default-op-input-spec "In")
        (slot-value instance 'outputs) (make-default-op-output-spec "Out")
        (slot-value instance 'params) (make-default-op-param-spec :size :const))
  (op-param-toggle-mutability (slot-value instance 'params) :const)
  ;; tratar parâmetro :size
  (set-op-param-valid-fn (slot-value instance 'params) :size
                         #'(lambda (new-val param-spec)
                             (declare (ignore param-spec))
                             (and (numberp new-val) (plusp new-val) (<= new-val 32))))
  (set-op-param-update-fn (slot-value instance 'params) :size
                          #'(lambda (new-val attrib param-spec blk)
                              (declare (ignore blk))
                              (setf (value attrib) new-val)
                              (when (op-param-bound-p param-spec :const)
                                (let ((c (get-op-param-val param-spec :const)))
                                  (when (>= c (expt 2 new-val))
                                    (set-op-param-val-no-check param-spec
                                                               :const
                                                               (1- (expt 2 new-val)))))))))
  ;; tratar parâmetro :const
  (set-op-param-valid-fn (slot-value instance 'params) :const
                         #'(lambda (new-val param-spec)
                             (when (and (numberp new-val) (plusp new-val))
                               (if (op-param-bound-p param-spec :size)
                                   (< new-val (expt 2 (get-op-param-val param-spec :size)))
                                   (< new-val (expt 2 32))))))
  (set-op-param-update-fn (slot-value instance 'params) :const
                          #'(lambda (new-val attrib param-spec blk)
                              (declare (ignore blk))
                              (when (not (op-param-bound-p param-spec :size))
                                (set-op-param-val-no-check param-spec
                                                           :size
                                                           (integer-length new-val)))
                              (setf (value attrib) new-val)))
  instance)
```

O método check-operator-parameters é muito simples, porque só tem de verificar que :size está especificado, já que o respectivo valor deve ser especificado antes da geração da implementação. O mesmo não acontece com :const; caso não tenha um valor, o gerador usa

um valor por omissão.

```
(defmethod check-operator-parameters ((op mutable-const-adder))
  "Returns T if parameter values are coherent."
  (unless (param-bound-p op :size)
    (error "Parameter :size of MUTABLE-CONST-ADDER must be initialized."))
  t)
```

O tamanho dos portos de um `mutable-const-adder` é simplesmente igual ao valor de `:size`.

```
(defmethod calculate-port-size ((op mutable-const-adder))
  (let ((size  (get-op-param-val (params op) :size)))
    (set-op-output-size (outputs op) "Out" size)
    (set-op-input-size (inputs op) "In" size)))
```

O mapeamento entre operador e implementação é definido, para cada tipo de operador, pelo método `generate-operator-block`, especializado para o operador em questão. Desta forma deixa-se ao implementador de operadores uma vasta amplitude de selecção do melhor tipo de implementação.

Neste caso, a construção da representação física é deixada a cargo de um gerador chamado `gen-mutable-const-adder`, cujos argumentos são derivados dos parâmetros do operador. Um gerador é uma função de Lisp encarregada de criar a representação física de um dado componente. Estas funções pertencem ao módulo PHYS e são descritas no capítulo 6. Neste caso a tarefa de `generate-operator-block` é simples, porque existe um gerador precisamente para este tipo de componentes. Noutros casos, o método `generate-operator-block` pode, por exemplo, escolher entre diferentes geradores, talvez baseado nas combinações de valores dos parâmetros.

```
(defmethod generate-operator-block ((op mutable-const-adder))
  "Generate the CONST-ADDER physical block."
  (let* ((x (gen-mutable-const-adder (get-param-value op :size)
                                     (get-param-value-default op :const 0))))
    (when (null x)
      (error "Failed to generate block for MUTABLE-CONST-ADDER operator."))
    (setf (slot-value op 'blk) x)))
```

O mapeamento entre entre as entradas do operador e os terminais do bloco físico é muito simples neste caso (como aliás na maior parte dos casos) e é definido pelos métodos `operator-input-mapping` e `operator-output-mapping`. Neste caso as funções limitam-se a verificar a correcção dos argumentos com que são invocadas e a retornar a posição do terminal desejado.

```
(defmethod operator-input-mapping ((op mutable-const-adder) port-name bit-idx)
  (unless (string= "In" port-name)
```

```
    (error "Unknown input port ~s for a MUTABLE-CONST-ADDER operator." port-name))
  (unless (and (not (minusp bit-idx))
               (< bit-idx (get-input-size op "In")))
    (error "Bit index out of range for input port."))
  bit-idx)

(defmethod operator-output-mapping ((op mutable-const-adder) port-name bit-idx)
  (unless (string= "Out" port-name)
    (error "Unknown output port ~s for an MUTABLE-CONST-ADDER operator." port-name))
  (unless (and (not (minusp bit-idx))
               (< bit-idx (get-output-size op "Out")))
    (error "Bit index out of range for output port."))
  bit-idx)
```

Este operador é implementado por um bloco reconfigurável dinamicamente. A configuração desse bloco varia em função da constante que deve ser adicionada. Para estas situações é necessário definir um método que converte os parâmetros do operador nos parâmetros do bloco reconfigurável. Neste caso a conversão é directa, porque o bloco reconfigurável tem apenas um parâmetro que até tem o mesmo nome. Em geral, o método `convert-parameters` pode fazer conversões tão complexas quanto necessário.

```
(defmethod convert-parameters ((op mutable-const-adder))
  (list (cons :const (get-op-param-val (params op) :const))))
```

Este exemplo mostra que a definição de um novo operador pode ser bastante fácil, embora o enquadramento seja muito flexível (já que os métodos a definir podem ser arbitrariamente complexos). Neste caso, a simplicidade advêm de o operador ter poucos parâmetros e um mapeamento directo num bloco reconfigurável. Para operadores mais abstractos, estes métodos (em particular `generate-operator-block`) podem ser muito mais sofisticados.

O exemplo também ilustra como o processamento de um modelo funcional está dividido em duas partes; o código ilustrado acima define um novo tipo de operador, mas pressupõe a existência de uma função geradora. Caso esta não exista (o que acontecerá na maior parte das vezes) será ainda necessário defini-la; mas essa tarefa está conceptualmente separada da definição do operador a nível funcional.

5.2.2 Redes de operadores

A representação das redes de operadores é feita pelas classes `operator-network-spec` e `operator-network-impl`, que representam respectivamente a especificação criada pelo utilizador e a sua implementação física.

A classe `operator-networ-spec` representa um modelo não decomposto e tem a seguinte definição:

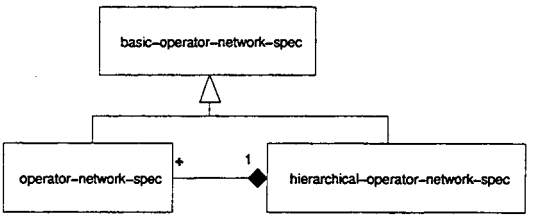


Figura 5.10: Relação entre as classes que representam a especificação.

```
(defclass basic-operator-network-spec ()
  ((id :reader id :initarg :id :type symbol
       :documentation "Network identifier.")
   (ops :accessor ops :type op-name-binding
        :initform (make-empty-op-name-binding)
        :documentation "Binding between names and operators.")
   (connections :accessor connections :type op-connection-set
                :initform (make-empty-op-connection-set)
                :documentation "Set of port connections that build the network.")
   (global-inputs :accessor global-inputs :type list
                  :initform nil
                  :documentation "List of all explicit global input names.")
   (global-outputs :accessor global-outputs
                   :type list :initform nil
                   :documentation "List of all explicit global output
                   names.")))

(defclass operator-network-spec (basic-operator-network-spec)
  ((levels :accessor levels :type (vector cons)
           :documentation "Elements of each structural level.")))

(defclass hierarchical-operator-network-spec (basic-operator-network-spec)
  ((sub-networks :accessor sub-networks :type (vector operator-network-spec)
                 :documentation "Subnetworks in scheduling order")
   (input-maps :accessor input-maps :type (vector subnetwork-input-mapping)
       :documentation "Input sources specifications for each subnetwork")
   (output-map :accessor output-map :type subnetwork-output-mapping
               :documentation "Output sources specification for global output" )))
```

A classe basic-operator-network-spec contém toda a informação necessária para representar uma rede de operadores; operator-network-spec acrescenta-lhe alguma informação derivada da estrutura da rede e necessária para processamento posterior. A classe hierarchi-

`cal-operator-netork-spec` contém a informação necessária para representar uma rede decomposta em sub-redes (v. subsecção seguinte). A relação entre as três classes está ilustrada na figura 5.10.

Uma instância de `operator-network-spec` contém a identificação do modelo (`id`), todos os operadores usados (incluindo operadores especiais que representam as entradas/saídas), as ligações entre operadores (`connections`), bem como os identificadores associados aos operados que representam entradas (`global-inputs`) e saídas (`global-outputs`) primárias do modelo (aquelas que são especificadas pelas cláusulas `:input` e `:output` nas definições de modelos com `define-operator-network`). Esta informação, que representa a totalidade do modelo, também está contida na descrição de uma rede composta.

A especificação de uma rede simples (não composta) inclui ainda a atribuição de operadores a níveis (campo `levels`). Conforme descrito na secção anterior, cada operador também tem um campo que especifica o seu nível; todos os operadores do mesmo nível devem estar na mesma posição do vector `levels`.

A especificação de uma rede decomposta contém um vector de subredes (`sub-networks`), cada uma das quais é uma rede simples. Contém ainda um vector que especifica, para cada sub-rede, a proveniência dos valores a colocar nas suas entradas (`input-maps`). Esses valores podem provir das entradas primárias ou das saídas de outras sub-redes. Também é necessário especificar como é que os resultados das sub-redes são mapeados para as saídas primárias (`output-map`).

5.2.3 Decomposição do modelo funcional

Em geral, a implementação de um dado modelo funcional pode não caber na infra-estrutura reconfigurável (neste caso, um circuito FPGA XC6216, conforme já foi mencionado). A opção mais óbvia num sistema que suporte reconfiguração é decompor o modelo em vários sub-modelos, cada um capaz de ser completamente implementado no circuito FPGA. No caso do sistema ReDiFlex, a decomposição pode ser feita tanto a nível do modelo funcional como da implementação física.

Para se poder apreciar convenientemente a situação é preciso conhecer algumas das características do modelo físico. Este modelo segue de perto a estrutura de níveis especificada pelo modelo funcional (fig. 5.11): as implementações de cada operador (designadas no seguimento por "blocos") são alinhadas em colunas ("fatias") que por sua vez são colocadas consecutivamente no circuito FPGA. Cada bloco tem as suas entradas do lado esquerdo e as saídas do lado direito; consequentemente todo o fluxo de dados é feito da esquerda para a direita. Esta organização física foi escolhida por permitir uma mapeamento directo e rápido do modelo funcional. A estrutura por coluna também se adapta bem a algumas restrições da interface de acesso do circuito XC6216; em particular, o acesso ao conteúdo dos elementos de memória do circuito está organizado por colunas.

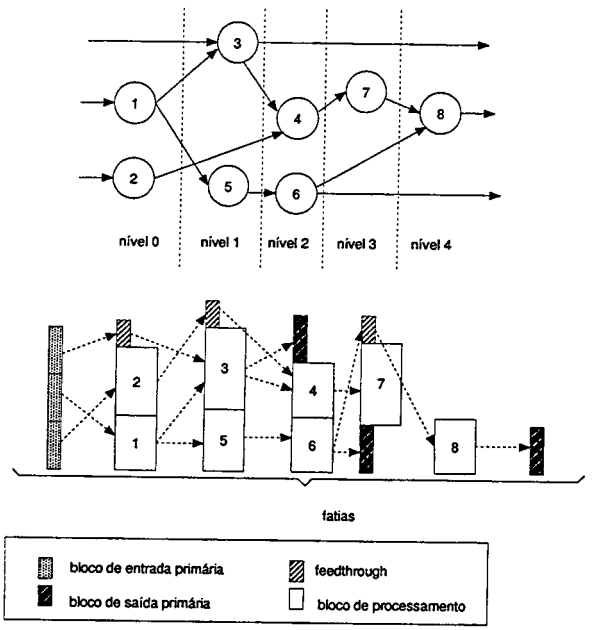


Figura 5.11: Transformação do modelo funcional num modelo físico.

Neste esquema podem surgir dois problemas: a implementação pode ser demasiado larga ou demasiado alta. O primeiro problema pode ser resolvido puramente no domínio físico (fig. 5.12): a implementação é partida em grupos de fatias (designados no código e na figura por *clusters*), por inserção de registos especiais, que permitem o acesso ao interior da implementação. Para executar um modelo decomposto desta maneira, o sistema de apoio à execução executa o primeiro grupo de fatias, obtém os valores da última fatia do grupo por acesso a um registo especial especialmente inserido na implementação para este efeito, reconfigura o circuito FPGA com o segundo grupo, inicializando o registo especial de acesso com os valores produzidos pelo primeiro grupo. Esta operação pode ser feita para todos os grupos em que a implementação inicial seja dividida.

Este método de decomposição, que se baseia nas características particulares da implementação do modelo físico adoptada no sistema ReDiFlex, pode ser aplicado após as fases de colocação e encaminhamento, o que significa que pode usar as dimensões exactas da implementação física (incluindo o espaço ocupado pelas interligações). Trata-se de um método simples e rápido de implementar, mas apenas pode ser aplicado quando cada fatia cabe na infra-estrutura reconfigurável.

O problema das fatias demasiado altas não pode ser resolvido tão facilmente no domínio físico. Uma possibilidade consistiria em partir uma fatia em duas (ou mais) fatias que res-

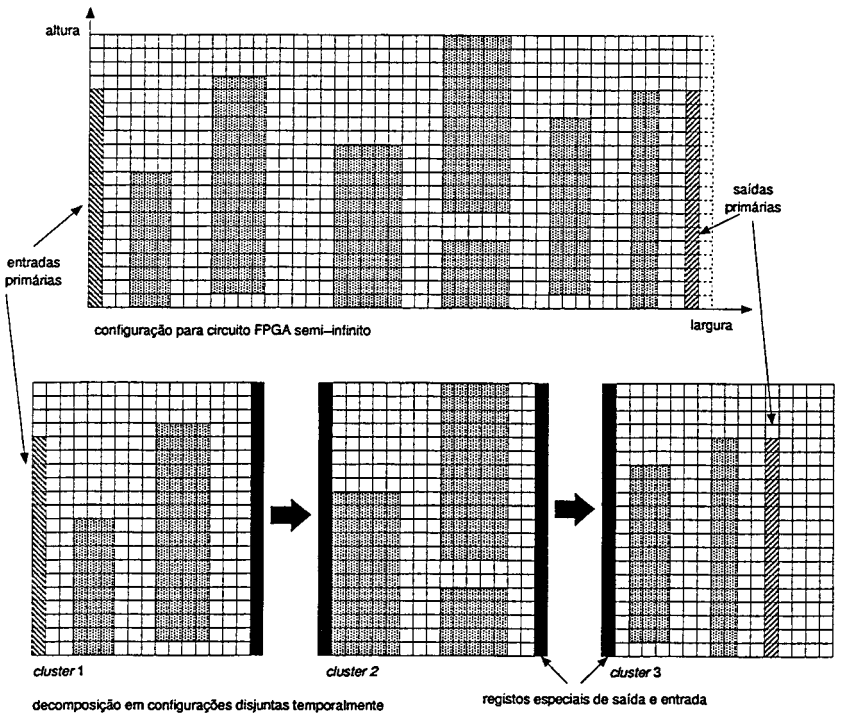


Figura 5.12: Princípio de decomposição no domínio físico.

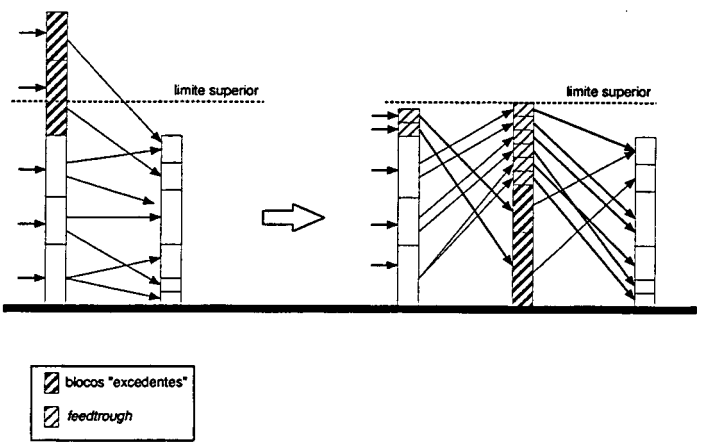


Figura 5.13: Problemas com a decomposição de fatias altas.

peitassem a altura desejada (fig. 5.13). Contudo o resultado obtido não é bom, porque pode aumentar substancialmente o número interligação entre fatias não consecutivas; cada uma destas interligações ocupa espaço nas fatias que atravessa. Como consequência as fatias resultantes da decomposição perderiam parte da sua altura "útil" para assegurar a comunicação entre as fatias precedentes e subsequentes.

Para resolver este problema, pode decompor-se o modelo funcional num conjunto de sub-modelos cuja implementação já não apresente o mesmo problema. Como cada sub-modelo dá origem a uma reconfiguração do circuito FPGA, interessa ter o mínimo número de sub-modelos, em que cada um aproveita o melhor possível a totalidade da infra-estrutura. Para tal o sistema ReDiFlex inclui um algoritmo que decompõe o modelo funcional num número mínimo de sub-modelos que respeitam as duas restrições seguintes:

1. A implementação de um sub-modelo não dá origem a fatias de altura superior à capacidade da infra-estrutura reconfigurável. Esta restrição é obrigatória.

2. A implementação de um sub-modelo inclui apenas um grupo de fatias. Esta restrição visa assegurar que os sub-modelos sejam de tamanho similar, e que portanto, o esforço de reconfiguração esteja repartido por todos. Esta restrição é opcional. Se a implementação não couber no circuito FPGA, é sempre possível aplicar posteriormente a decomposição a nível físico.

A decomposição do modelo funcional só é obrigatória se a primeira restrição não for satisfeita. Como o processo de mapeamento funcional→físico é muito simples, também é fácil determinar quando tal caso ocorre. Para que a primeira restrição seja verificada é necessário que, para todos os níveis (que são mapeados em fatias físicas), a soma das alturas (das implementações) dos operadores mais o espaço ocupado pelas ligações inter-níveis que atravessam o nível em questão, não seja superior à altura do circuito FPGA (altura e largura são medidos em termos do número de células básicas da arquitectura XC6200). Cada interligação que atravesse um nível (i.e. não tenha origem nem destino nesse nível) ocupa uma célula por bit. Todas as interligações com origem no mesmo terminal só devem ser contadas uma vez.

Quando o sistema ReDiFlex verifica que um modelo não satisfaz esta restrição, activa automaticamente a decomposição do modelo funcional antes de gerar o modelo físico. A decomposição funcional também pode ser desencadeada explicitamente, p. ex., para efeitos de teste.

O comprimento da implementação, necessária para avaliar a segunda restrição é mais difícil de determinar. De facto, apenas após as fases de colocação e encaminhamento é que estão disponíveis os valores correctos. Por consequência, durante a decomposição funcional é necessário estimar o comprimento (da implementação) de um modelo. Sendo L_f a soma da largura das fatias de um grupo, o sistema calcula a largura total estimada através da fórmula $L_T = L_f + \alpha(n-1)$, em que α é um número real positivo que representa o acréscimo de área

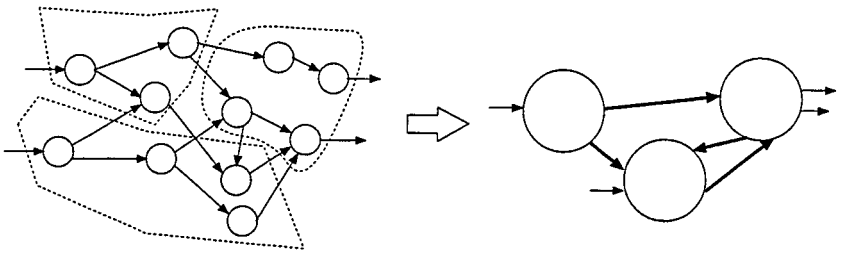


Figura 5.14: Decomposição com resultados cíclico e acíclico.

estimado devido às interligações e n é número de fatias do grupo (logo $n-1$ é o número de zonas de encaminhamento entre fatias).

O factor α deve ser estimado empiricamente. Sempre que o sistema ReDiFlex efectua uma decomposição física, notifica o utilizar dos tamanhos dos diversos grupos resultantes, o que permite uma avaliação rápida do efeito de α, cujo valor pode ser modificado antes de se repetir a decomposição funcional, se desejado. Por omissão, o sistema não itera a decomposição funcional se esta resultar em sub-modelos demasiado grandes, limitando-se a recorrer à decomposição física para ajustar a implementação dos sub-modelos à dimensão do circuito FPGA.

Uma formulação do problema da decomposição (temporal) de redes acíclicas foi apresentada por Kaul e Vemuri [1998] e Purna e Bhatia [1999]. O problema tem muitas semelhanças com o problema de escalonamento em síntese de alto nível. Dada uma rede acíclica de operadores, o objectivo é encontrar uma decomposição em k sub-redes de forma que:

1. o tamanho da implementação de cada sub-subrede é inferior à capacidade do circuito FPGA;

2. existe uma relação de precedência (acíclica) entre as sub-redes.

Conforme notam Purna e Bhatia [1999] a primeira restrição não implica a segunda. A figura 5.14 mostra uma decomposição de uma rede acíclica que resulta numa relação cíclica entra as diferentes sub-redes.

Kaul e Vemuri [1998] modelam o problema de decomposição temporal como um problema de programação não-linear binária. Na prática, a resolução do problema por este método não é feita em tempo aceitável (o algoritmo é de complexidade exponencial), pelo que diversas heurísticas alternativas são propostas por Purna e Bhatia [1999]. O sistema ReDiFlex inclui uma implementação da heurística baseada em níveis descrita por esses autores, com uma melhoria introduzida por Cardoso e Neto [1999] no sistema NENYA. A heurística original terminava a construção de uma sub-rede logo que fosse encontrado um operador que não coubesse; a versão de Cardoso e Neto [1999] tenta aproveitar os operadores restantes do mesmo nível.

O algoritmo de decomposição de redes aplica-se a uma rede acíclica $\mathcal{R}$ com N operadores op_i^l (i-ésimo operador do nível l). O objectivo é obter um conjunto $\mathcal{P}$ de sub-redes, cada uma das quais cabe num circuito FPGA de largura L_{fpga} e altura A_{fpga}. Altura e largura são medidos em número de blocos lógicos. Para o circuito XC6216 temos $A_{fpga} = L_{fpga} = 64$.

1. *Inicializar.* No início, não existe nenhuma sub-rede $\mathcal{P} = \emptyset$. Seja $r = \emptyset$ a sub-rede em construção e $l = 0$ o nível corrente.

2. *Incluir operadores de nível l.* Para cada operador op^l ainda não atribuído a uma sub-rede determinar se acrescentar o operador a r viola as restrições de tamanho; em caso negativo incluir o operador na sub-rede $r = r \cup \{op\}$.

3. *Mudar de nível.* Se todos os operadores de nível l estão distribuídos, fazer $l = l + 1$. Se existirem mais níveis voltar ao passo 2, senão terminar.

4. *Construir nova sub-rede.* O nível l contém operadores que não cabem em r, pelo que é necessário construir uma nova sub-rede. Fazer $\mathcal{P} = \mathcal{P} \cup \{r\}, r = \emptyset$ e voltar ao passo 2.

Durante a execução do passo 2 é preciso determinar se o novo operador pode ser acrescentado à sub-rede r, o que se traduz em verificar se as restrições de altura e comprimento são respeitadas. É importante notar aqui que o novo operador op é acrescentado ao mesmo nível de r que todos os outros operadores do seu nível l. Seja m o nível de r em que são colocados os operadores de nível l de $\mathcal{R}$. Então a soma das alturas de todos os operadores de nível m mais o espaço ocupado pelas ligações inter-nível que atravessam m não deve exceder A_{fpga}.

Quanto ao comprimento, é necessário calcular o comprimento de todas as fatias de r (que é a largura do componente mais largo de cada fatia de r), incluindo uma eventual alteração da largura da m-ésima fatia causada pela inclusão de op. Calculada a soma das larguras, pode-se estimar a largura total (L^* da implementação de r se incluísse op. Para incluir op em r é necessário que a largura estimada L^* verifique a condição $L^* \leq L_{fpga} - 2$. O termo 2 representa o espaço ocupado pelos circuitos de entrada e saída.

O sistema ReDiFlex usa exclusivamente o nível ASAP, porque se entendeu que a versatilidade assim obtida era suficiente para o actual protótipo. Contudo, deve notar-se que é possível obter, em alguns casos, resultados melhores usando outras atribuições de operadores a níveis. O algoritmo usado fornece apenas uma possível ordem total a partir das restrições de precedências implícitas na rede de operadores. A atribuição de níveis pode ser mais sofisticada se levar em conta a mobilidade de cada operador, i.e. todos os níveis possíveis para um dado operador. A mobilidade de cada operador pode ser determinada calculando o respectivo nível ALAP (*as late as possible*), i.e. o maior nível em que o operador pode ser colocado sem aumentar o número total de níveis (que é igual ao maior nível obtido pelo método ASAP). Para cada operador, a mobilidade é definida como a diferença entre os seus níveis ALAP e ASAP [de Micheli, 1994]. Explorando a mobilidade dos operadores pode ser possível encontrar em

alguns casos uma atribuição de níveis que evite a necessidade de decomposição. A mobilidade também pode ser usada para orientar a escolha dos operadores a processar durante a decomposição descrita acima, conforme descrevem Cardoso e Neto [1999]. Por fim, note-se que, como indicam estes autores, o algoritmo de decomposição usado no sistema ReDiFlex também pode ser aplicado com base no ordenamento ALAP .

No sistema ReDiFlex, a interface para o processamento do modelo funcional é a função `prepare-opnet-spec-for-implementation`:

```
(defun prepare-opnet-spec-for-implementation (op-spec)
  "Prepare the data structure OP-SPEC for the implementation phase."
  (declare (type operator-network-spec op-spec))
  (assign-operator-levels op-spec)
  (fill-in-feedthroughs op-spec)
  (for-each-op-in-binding #'(lambda (x)
                              (generate-operator-block x)
                              (revert-pred-count x)) ; !!!
                          (ops op-spec))
  (match-opnet-delays op-spec)
  (maybe-decompose-opnet-spec op-spec))
```

Esta função calcula os níveis do modelo (`assign-operator-levels`) e acrescenta operadores internos que representam explicitamente a passagem de dados através de um nível (`fill-in-feedthroughs`). Seguidamente gera a representação física de cada bloco (`generate-operator-block`), o que permite determinar as suas dimensões e tipo (combinacional ou sequencial). Com essa informação é possível determinar o ciclo de relógio de cada operador e, se necessário, acrescentar atrasos aos percurso de dados para equilibrar os tempos de chegada dos sinais às entradas de cada operador (`match-opnet-delays`) (também aqui são usados operadores internos para representar esses atrasos). Finalmente, a função `maybe-decompose-opnet-spec` avalia a necessidade de decomposição do modelo funcional, e, caso necessário, efectua essa mesma decomposição. Esta função retorna ao modelo definitivo, que pode ser do tipo `operator-network-spec` (no caso de não existir decomposição) ou `hierarchical-operator-network-spec` (no caso contrário). Este é também o resultado retornado por `prepare-opnet-for-implementation`.

O comportamento da função `maybe-decompose-opnet-spec` é controlado pela variável dinâmica `*decomposition-mode*` a que o utilizador pode atribuir os seguintes valores legais:

`nil` Não é efectuada a decomposição do modelo funcional. Se esta é mesmo necessária, `maybe-decompose-opnet-spec` gera um erro. A função retorna sempre um valor do tipo `operator-network-spec`.

`t` A decomposição do modelo funcional é efectuada se necessário. Este é o valor por omissão.

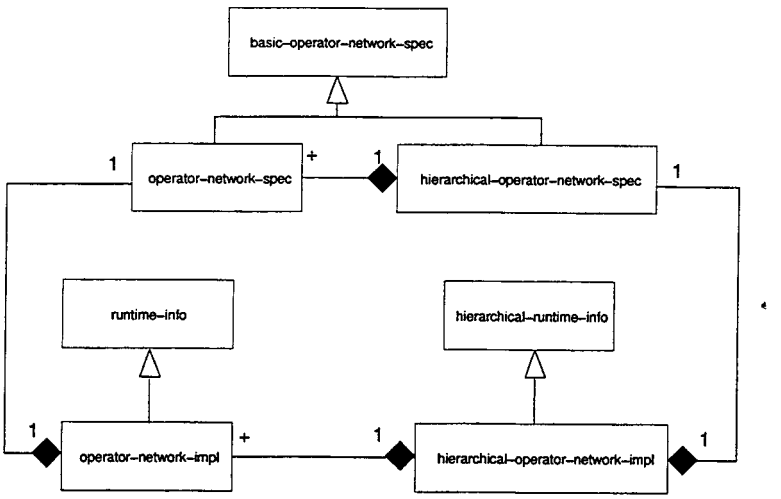


Figura 5.15: Relação entre as estruturas de dados usadas para representar especificação e implementação.

`:force` A decomposição do modelo funcional é sempre efectuada. A função `maybe-decompose-opnet-spec` produz sempre um objecto pertencente à classe `hierarchical-operator-network-spec`.

Outra variável dinâmica que influencia a decomposição é `*alpha*`, que representa o factor com que as interligações contam para a estimativa do tamanho da implementação.

Convém notar que o método `generate-operator-block`, que conceptualmente, gera toda a informação necessária para implementar o operador, na realidade se limita a determinar as características físicas da implementação (dimensões, posições dos terminais de entrada e saída, etc.), mas não cria ainda a informação necessária para configurar a infra-estrutura de *hardware*. Essa informação é gerada unicamente quando necessária, isto é, na altura de configurar o circuito FPGA (cf. capítulo 7).

5.3 Construção do modelo físico

Baseado no modelo funcional, o sistema ReDiFlex constrói uma representação interna da especificação apropriada. Essa representação é levemente diferente para redes simples e para redes compostas (fig. 5.15). Para uma rede simples, a implementação é uma instância da classe `operator-network-impl`:

```
(defclass runtime-info ()
  ((mapped :accessor mapped :type boolean :initform nil
```

```
        :documentation "True if the network is currently mapped to hardware.")
   (active-cluster :accessor active-cluster
                   :type pos-fixnum :initform 0
                   :documentation "If MAPPED is true, ACTIVE-CLUSTER specifies the subcluster.")))

(defclass operator-network-impl (runtime-info)
  ((global-spec :accessor global-spec :initarg :global-spec :type operator-network-spec
                :documentation "The toplevel specification of the network.")
   (split-level :accessor split-level :type (simple-array pos-fixnum)
                :documentation "The level at which a cluster starts.")
   (clusters :accessor clusters :type (simple-array scluster (*))
             :documentation "The cluster that implements each spec.")
   (cluster-ticks :accessor cluster-ticks :type (simple-array pos-fixnum (*))
                  :documentation "Number of clock ticks for this cluster.")
   (clock-ticks :accessor clock-ticks :type pos-fixnum
                :documentation "Total number of clock ticks that the implementation needs.'')
   (setups :accessor setups
           :type (simple-array design-setup (*))
           :documentation "The internal registers of each cluster.")
   (data-io-itf :accessor data-io-itf
                :type data-io-map
                :documentation "Functional data interface spec.")))
```

A implementação inclui informação necessária para a execução (classe base `runtime-info`) bem como o modelo físico associado. Os detalhes do modelo físico são descritos na secção 6.1. Para perceber a definição desta classe, basta saber que cada configuração da infra-estrutura de *hardware* é representada por um grupo de implementações de operadores, designados por *clusters*. A implementação de uma rede inclui uma referência ao modelo que lhe serviu de especificação (`global-spec`), o conjunto de grupos de operadores (`clusters`), a correspondência entre níveis do modelo e grupos (`split-level`) e o número de ciclos de relógio que cada grupo necessita. O campo `setups` especifica a maneira de aceder às variáveis de estado (que correspondem a registos em *hardware*) de cada grupo de operadores. O número de ciclos de relógio necessários para propagar os dados através de cada grupo é especificado pelo vector `cluster-ticks`; a sua soma total é guardada em `clock-ticks`. O campo `data-io-itf` contém a informação necessária para o sistema de apoio à execução efectuar a transferência de dados da aplicação para o co-processador.

A função `make-opnet-impl` é usada para gerar a implementação. De notar que nesta fase, os grupos de operadores apenas contém informação sobre os blocos físicos que implementam os operadores e respectivo agrupamento por fatias, bem como a indicação dos terminais a ligar (*netlist*). A geração da representação física completa é da responsabilidade do módulo PHYS (capítulo 6). A função `make-opnet-impl` executa consecutivamente as seguintes operações:

1. Criação de grupos (*clusters*). Cada grupo é constituído por fatias de blocos (que implementam os operadores). Cada fatia corresponde a um nível do modelo funcional. A exacta colocação de cada bloco no grupo ainda não está determinada, mas será determinada durante a fase de colocação (secção 6.3.1).

2. Criação das ligações entre terminais dos operadores. Nesta fase, as ligações são apenas especificadas pelos terminais envolvidos. A especificação completa será determinada na fase de encaminhamento (secção 6.3.2).

3. Executar colocação e encaminhamento do grupo, supondo que o circuito FPGA é infinitamente largo.

4. Se a implementação do grupo for demasiado grande, fazer a decomposição da representação física.

5. Especificar a interface com o modelo baseada na informação física entretanto disponível. Como essa informação é usada pelo sistema de apoio à execução, a sua descrição detalhada está incluída no capítulo 8.

A informação associada à implementação de uma rede composta (ou hierárquica) é ligeiramente diferente (fig.5.15). A classe `hierarchical-operator-network-impl` contém uma referência para a especificação e um vector com as implementações de cada sub-modelo. Como cada sub-modelo é sempre simples, a implementação respectiva é do tipo já referido `operator-network-impl`. A geração da implementação é feita por uma versão de `make-op-net-impl` que se limita a invocar o método do mesmo nome para cada sub-modelo e a guardar o respectivo resultado no vector `subnetwork-impls`. O campo `data-io-itf` contém a informação necessária para o sistema de apoio à execução construir a função de interface e gerir a transferência de informação entre as várias sub-redes.

```
(defclass hierachical-runtime-info ()
  ((active-subnetwork :accessor active-subnetwork :type pos-fixnum
                      :initform 0
                      :documentation "The mapped subnetwork.")))

(defclass hierarchical-operator-network-impl (hierachical-runtime-info)
  ((global-spec :accessor global-spec :initarg :global-spec
                :type hierachical-operator-network-spec
                :documentation "The toplevel specification of the network.")
   (subnetwork-impls :accessor subnetwork-impls :type (vector operator-network-impl)
                     :documentation "Subnetwork implementations in scheduling order.")
   (data-io-itf :accessor data-io-itf :type toplevel-data-io-map
                :documentation "Functional data interface spec.")))
```

Um modelo tenha várias implementações diferentes, p. ex., como resultado de decomposição com diferentes valores de `*alpha*`.

As duas classes descritas incluem ainda alguns campos que se destinam a ser usados durante a activação das respectivas implementações. No caso de uma rede simples, usa-se o campo `mapped` para registar o facto a implementação estar em uso; nesse caso o campo `active-cluster` contém o índice do grupo cuja configuração está no circuito FPGA. (Uma implementação terá mais que um grupo se tiver havido uma decomposição ao nível físico.) As redes decompostas contém o campo `active-subnetwork` para registar o índice da sub-rede a que pertence o grupo activo. Tanto `active-cluster` como `active-subnetwork` apenas são válidos se `mapped` tiver o valor verdadeiro. Estes campos são usados pelo sistema de apoio à execução para registar o estado (activo/inactivo) de cada implementação.

5.4 Resumo

O sistema ReDiFlex usa um modelo funcional de alto nível para especificar o comportamento da infra-estrutura reconfigurável. O modelo é baseado numa rede acíclica de operadores e é bastante semelhante aos tradicionais diagramas de fluxo de dados. Cada operador contém um número bem definido de entradas e saídas. As ligações entre operadores são unidireccionais e têm uma capacidade bem definida (em bits), que deve corresponder à dimensão das entradas e saídas a que estão associadas. Um operador pode ter parâmetros que especificam aspectos particulares da sua funcionalidade; também é possível aceder a informação sobre o seu estado interno (para implementações sequenciais). Além de operadores regulares, o modelo inclui operadores sem entradas (geradores) ou sem saídas (acumuladores). Estes dois últimos tipos de operadores só têm utilidade de o modelo for usado num contexto vectorial, i.e., numa situação em que a sua implementação seja alimentada sem interrupção por uma sequência de valores de entrada.

O modelo suporta a reconfiguração dinâmica ao permitir a modificação em tempo de execução de alguns parâmetros dos operadores; cada novo valor pode dar origem a uma nova implementação da funcionalidade genérica associada ao operador. Para além disso, o sistema suporta contentores, que são operadores que combinam a funcionalidade de vários outros operadores compatíveis. Em qualquer altura, o contentor pode ser reconfigurado para assumir a personalidade de um dos elementos constituintes. Estes mecanismos permitem uma variação controlada da implementação.

As mutações da implementação são desencadeadas directamente pelo utilizador/aplicação, o que permite fazer depender as reconfigurações de critérios arbitrariamente complexos.

A definição de modelos pode ser efectuada por chamada a funções apropriadas ou usando a instrução `define-operator-network`, que permite misturar a definição da rede com código Lisp arbitrário. Qualquer elemento da especificação (valores dos parâmetros, tipos dos ope-

radores, etc,) pode ser substituído por código Lisp que produza a informação desejada. Os modelos são representados por objectos de Lisp (da classe `operator-network-spec`), que podem ser manipulados como quaisquer outros objectos do sistema (usados como argumento ou resultado de funções, copiados, etc.).

Dado um modelo, o sistema gera uma implementação física baseada directamente na estrutura do modelo funcional. Em geral, a implementação de um modelo pode exceder a capacidade da infra-estrutura de *hardware*. A capacidade de reconfiguração é então usada para dar a ilusão de uma infra-estrutura muito maior (*hardware* virtual), multiplexando a infra-estrutura existente entre as implementações de partes do modelo global. A decomposição necessária pode ser feita sobre o próprio modelo funcional ou, em algumas situações, já sobre a implementação física. A decomposição funcional usa um método heurístico rápido, inicialmente descrito por Kaul e Vemuri [1998] e também adoptado por Cardoso e Neto [1999]. O sistema procede automaticamente à decomposição funcional sempre que necessário. A função `prepare-opnet-for-impl` é o principal ponto de acesso a todo este mecanismo.

O mapeamento entre o domínio funcional e o físico segue de perto a estrutura de precedências imposta pelo modelo funcional, o que facilita a tarefa de estimar a influência do modelo funcional sobre a implementação, o que é importante quando se procede à decomposição. Por outro lado, a implementação física adquire assim uma estrutura que torna as tarefas de colocação e encaminhamento mais simples e rápidas. A função `make-opnet-impl` encarrega-se de todas as operações necessárias para criar a implementação de um modelo funcional a partir da especificação.

6 Nível físico: modelo e ferramentas

Definido o modelo funcional do comportamento da infra-estrutura reconfigurável, é necessário determinar quais os recursos físicos a usar (blocos lógicos, recursos de encaminhamento, sinais globais) e, se necessário, decompor a implementação física de forma a respeitar as dimensões físicas impostas pelo circuito FPGA. As rotinas responsáveis por estas tarefas pertencem ao módulo PHYS (cf. fig. 4.4). As rotinas deste módulo não fazem o processamento físico de circuitos arbitrários; estão antes particularmente adaptadas para circuitos cuja estrutura seja facilmente derivável do modelo funcional. Um objectivo importante deste módulo é produzir toda a informação física necessária de uma forma rápida, que seja compatível com a utilização interactiva do sistema. Consequentemente a abordagem escolhida privilegia a rapidez na obtenção dos resultados sobre a qualidade destes em termos do aproveitamento da área.

O modelo funcional é mapeado directamente num modelo físico de estrutura simples, em que a cada operador corresponde um componente físico. Por sua vez estes são agrupados em fatias verticais. Cada componente tem um formato rectangular, com terminais de entrada e saída de lados opostos. Componentes de uma fatia são alimentados por sinais provenientes da fatia imediatamente anterior (à esquerda) e, por sua vez, alimentam componentes da fatia imediatamente seguinte (à direita).

O sistema ReDiFlex dispõe de ferramentas automáticas para efectuar a colocação dos componentes e fazer o encaminhamento das respectivas ligações. O principal objectivo é ter uma

implementação com altura inferior ao limite imposto pela arquitectura do circuito FPGA; acessoriamente pretende-se que, no seu todo, o grupo ocupe a menor área possível. Se um grupo de fatias necessitar de uma área superior à do circuito FPGA, é automaticamente decomposto em sub-grupos de dimensões apropriadas.

O capítulo começa com uma descrição do modelo físico, incluindo uma descrição da arquitectura geral dos circuitos gerados e das restrições a que os componentes estão sujeitos. Como decorre do modelo funcional usado, os componentes (implementações dos operadores) são um ponto fulcral do modelo físico, pelo que são analisados detalhadamente na secção 6.2; são analisados tanto os componentes simples como os que implementam contentores. Também é descrita a forma como informação necessária para a operação global do circuito é coligida a partir da informação de cada componente, em particular no que respeita ao uso de sinais globais (sinais de relógio e de inicialização) e ao acesso a registos internos dos componentes. A secção 6.3 descreve as abordagens usadas no processamento do modelo físico, nomeadamente os algoritmos usados para a colocação dos componentes e encaminhamento das interligações.

6.1 Modelo físico

O módulo PHYS cria a representação física detalhada da implementação de uma rede de operadores. Conforme descrito na secção 5.3, o módulo OPNET mapeia cada operador num

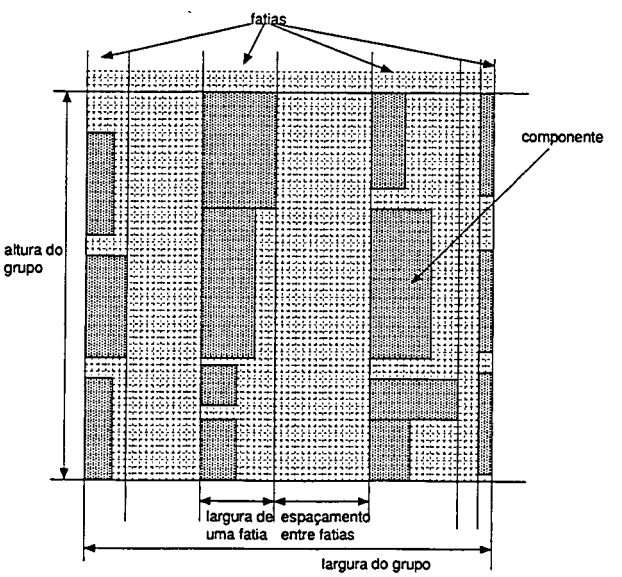


Figura 6.1: Organização de um grupo de componentes do modelo físico.

componente físico e gera uma lista das interligações entre terminais físicos. A implementação de um modelo deve respeitar um conjunto de condições e restrições, que no seu conjunto definem o modelo conceptual do circuito físico.

O mapeamento funcional→físico é baseado em redes de operadores simples. O mapeamento de redes hierárquicas (i.e., aquelas que foram sujeitas ao processo da decomposição funcional) é reduzido ao mapeamento independente das respectivas sub-redes, que são sempre simples. Conceptualmente, uma rede é mapeada num grupo de componentes e respectivas interligações, dotado de uma estrutura particular. Estes grupos ou agregados (*clusters*) de componentes são a entidade principal a ser submetida aos diversos algoritmos de processamento descritos subsequentemente.

Um grupo é composto por fatias verticais de componentes (fig 6.1). Cada componente é representado por um rectângulo (de lados paralelos aos limites do circuito FPGA). Para efeitos de colocação e encaminhamento, o interior dos componentes é inacessível. Cada componente tem os terminais de entrada do seu lado esquerdo e os de saída à direita. Não existem terminais bidireccionais. Todas as interligações são entre fatias verticais adjacentes. As entradas e saídas do modelo funcional são representadas por componentes especiais, que implementam registos a que o processador principal tem acesso directo para leitura ou escrita. Todos os componentes são síncronos, controlados pelo mesmo sinal de relógio. (Existe uma excepção importante a esta última regra, explicada na página 190.) O sinal de inicialização (*clear*) também é o mesmo para todos os componentes.

No sistema ReDiFlex optou-se por restringir a estrutura física da implementação a este arranjo pelas simplificações que introduz a vários níveis. Em particular:

- O mapeamento funcional→físico é directo, o que se traduz na simplicidade e rapidez da respectiva implementação. Mais importante ainda, a determinação da necessidade de proceder à decomposição funcional (que depende da capacidade de determinar se a implementação cabe na infra-estrutura) pode ser feita segundo critérios precisos (sec. 5.2.3).

- Esta estrutura permite usar para a colocação e o encaminhamento algoritmos simples e ainda assim obter um tempo de execução compatível com o objectivo de se obter um sistema de desenvolvimento interactivo.

- A estrutura está bem adaptada à organização do acesso aos recursos do circuito XC6216 durante a execução de aplicações. Embora a organização dos blocos lógicos e da rede de interligações programáveis geralmente não privilegie uma dada orientação, existem duas características que favorecem uma organização por colunas:

  1. Os sinais globais (sinais de relógio e inicialização) são comuns a sub-conjuntos de blocos lógicos arranjados por colunas. Mais precisamente, cada quatro células de

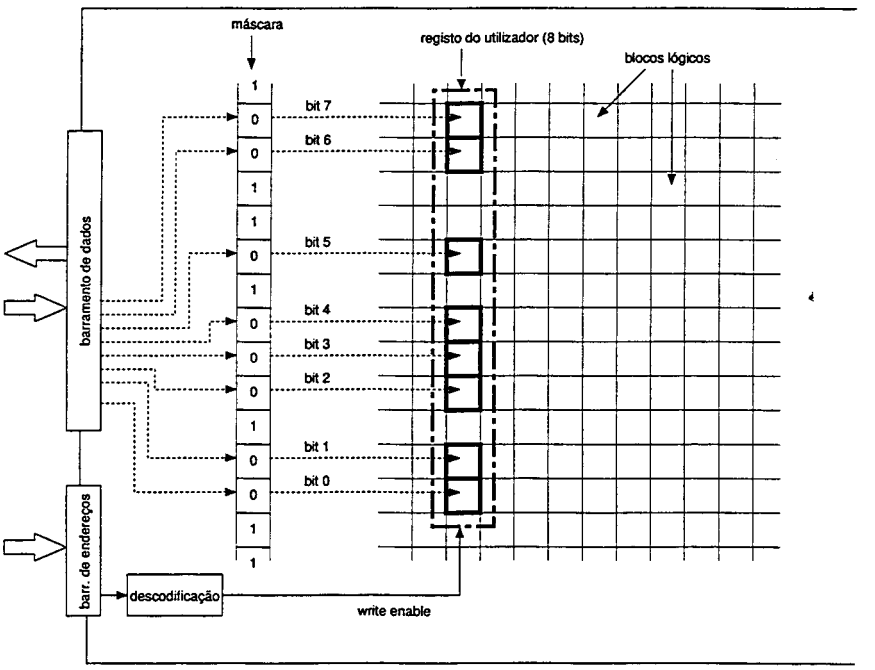


Figura 6.2: Acesso a um registo do utilizador segundo Churcher *et al.* [1995].

uma coluna partilham o mesmo sinal de relógio; de forma semelhante, dezasseis blocos consecutivos de uma coluna partilham o mesmo sinal de inicialização.

2. É possível aceder simultaneamente, a partir da interface externa do circuito FPGA, ao conteúdo de *flip-flops* de células situadas na mesma coluna. Consequentemente é vantajoso orientar verticalmente os registos internos de maneira a serem directamente acessíveis do exterior, o que, por sua vez, privilegia uma orientação horizontal para a propagação da informação.

O mapeamento funcional→físico, embora directo, deve ter em atenção alguns factores. Em geral, cada nível de um modelo funcional dá origem a uma fatia do modelo físico. Para impor a condição de todas as ligações serem entre componentes de níveis sucessivos, é necessário introduzir no modelo funcional elementos adicionais (designados por *feedthroughs*) (pág. 179), cuja finalidade consiste unicamente em efectuar a transferência de sinais através de uma fatia. O modelo funcional não depende explicitamente do tempo; o funcionamento temporal correcto do modelo físico é garantido antes de este ser gerado, se necessário através da inclusão de operadores de atraso ao nível funcional (pág. 179).

A transferência de dados entre processador principal e co-processador é feita pelo pro-

cessador através de instruções de leitura/escrita que permitem aceder aos registos internos do circuito implementado na infra-estrutura reconfigurável. A interface FastMap [Churcher *et al.*, 1995] dos elementos da família XC6200 permite especificar através de uma máscara que *flip-flops* de uma dada coluna pertencem a um registo acessível (*user register*) do circuito implementado (fig. 6.2; ver também a pág. 66). Um subsequente acesso com essa coluna como endereço permite ler ou escrever o valor do registo. Assim, os componentes do modelo físico que implementam as entradas e saídas do modelo funcional são simplesmente registos de escrita ou leitura, respectivamente. As variáveis de estado dos operadores correspondem fisicamente a registos de leitura/escrita que o definidor dos operadores entendeu tornar explícitos a nível funcional.

Durante a operação do circuito implementado em FPGA é necessário manter o controlo rigoroso do sinal de relógio. Em particular, o acesso aos registos internos para leitura deve ser feito quando estes têm valores válidos, o que acontece apenas em instantes bem definidos; de igual forma, a inicialização dos mesmos registos deve ser feita nos instantes apropriados. O controlo do sinal de relógio é particularmente delicado em contexto vectorial, caso em que a gestão eficiente da execução requer que a implementação opere em modo *pipelined*. (Estes aspectos são abordados com mais pormenor no capítulo 8.) Surge aqui um delicado problema de implementação, devido ao facto de o acesso a registos interferir directamente com o controlo do sinal de relógio: cada acesso implica obrigatoriamente a execução de 2 (para leitura) ou 3 (para escrita) ciclos do relógio global GCLK. Como o número e momento exacto dos acessos não são, em geral, antecipáveis (p. ex, quando se pretende examinar o conteúdo dos registos para fazer a depuração de uma aplicação), este comportamento do circuito de interface introduz uma séria complicação.

Existem diversas possibilidades de contornar o problema, que geralmente envolve a introdução de sinais de *enable* em todos os componentes sequenciais e a utilização de circuitos condicionadores do sinal de relógio. Para aplicações concretas pode ser possível adoptar técnicas *ad hoc* menos gerais.

O sistema ReDiFlex usa uma outra abordagem, em que quase todos os componentes são controlados por um sinal de relógio alternativo (G1). Neste caso é preciso garantir que os sinais G1 e GCLK não entram em conflito. Para que um acesso de leitura decorra sem problemas, é necessário que as saídas dos *flip-flops* estejam estáveis no flanco ascendente de GCLK, i.e. não deve ocorrer uma modificação das saída que viole as restrições de *setup time* e *hold time* em relação ao flanco activo de GCLK usado para capturar/modificar o conteúdo dos registos. No sistema ReDiFlex, essa condição é garantida pelo simples expediente de apenas se efectuarem acessos de leitura com o sinal G1 desactivado. Nestas condições, os circuitos de controlo residentes na placa H.O.T. Works geram automaticamente os impulsos de GCLK necessários, quando detecta um acesso.

O acesso para escrita é mais complicado, porque é necessário que os *flip-flops* sejam sensíveis ao sinal GCLK para efectuarem a captura do novo valor. Este problema é resolvido de

duas maneiras diferentes. Em primeiro lugar, os componentes que implementam as entradas do modelo são sempre activados pelo sinal GCLK; neste caso não existe a possibilidade de o seu conteúdo ser alterado por ciclos de relógio adicionais; por sua vez o restante circuito não é alterado porque está configurado para reagir apenas a G1. O acesso para escrita a registos internos dos outros componentes é mais complicado: o componente é reconfigurado temporariamente para reagir a GCLK, o multiplexador de protecção do conteúdo é configurado devidamente (multiplexador RP da fig. 6.7), o valor do registo é alterado, e, finalmente, a configuração inicial é reposta. Com esta abordagem o acesso para escrita a registos de componentes é mais oneroso que o acesso para leitura. Na implementação actual, o acesso para escrita é usado apenas para estabelecer os valores iniciais, o que faz com que seja uma operação mais rara que o acesso para leitura.

Para simplificar a implementação, o relógio de controlo é estabelecido por fatia; assim a primeira fatia é controlada por GCLK enquanto todas as outras são controladas por G1. Para isso, a primeira fatia deve ser composta exclusivamente por registos de entrada; quaisquer geradores (que logicamente pertencem também ao primeiro nível ASAP) são passados para o segundo nível (para serem mapeados na segunda fatia). Como caso especial, um modelo sem entradas primárias (i.e. apenas alimentado por geradores) não dá origem a nenhuma fatia sensível a GCLK.

Todo o problema levantado pelo comportamento da interface de acesso aos registos internos é um exemplo da distância que pode existir entre um modelo conceptual simples e a sua implementação na presença das restrições efectivamente encontradas. Também é um bom exemplo para a última epígrafe da página 129.

O conjunto de grupos de componentes definido pelo mapeamento funcional→ físico é processado para se obter a especificação exacta da colocação de cada componente e das respectivas interligações. Esses algoritmos consideram que o circuito FPGA tem largura infinita. Naturalmente, pode suceder que a implementação resultante ultrapasse as dimensões do circuito físico, pelo que pode ser necessário decompor a implementação segundo o processo descrito na secção 5.2.3. O processo de decomposição física que pode ser formulado mais formalmente da seguinte forma:

1. *Inicialização.* Seja k o número do sub-grupo $\mathcal{S}_k = \emptyset$ em construção. De início $k = 1$. Seja $i = 1$ o contador das fatias processadas. A largura do circuito FPGA é constante L_{fpga}.

2. *Preenchimento.* Preencher $\mathcal{S}_k$ com todas as fatias possíveis. A fatia i é incluída em $\mathcal{S}_k$ e i incrementado se for verificada uma das seguintes condições:

   - (i) a fatia i é a última e a sua inclusão não leva a largura de $\mathcal{S}_k$ a exceder L_{fpga}; neste caso este é o último sub-grupo e a decomposição está terminada.
   - (ii) a inclusão da fatia i e da zona de interligações que a liga à próxima fatia não leva a largura de $\mathcal{S}_k$ a exceder $L_{fpga} - 1$ (para deixar espaço para o registo usado

para capturar os valores que deveriam ir para a fatia seguinte).

3. *Inserir elementos de entrada/saída.* Acrescentar a S_k um registo de altura igual à altura do circuito FPGA; este registo serve para capturar os valores que originalmente entrariam para próxima fatia. O sub-grupo k está completo. Incrementar k e começar um novo grupo $S_k = \{Ireg\}$, cujo primeiro elemento é um registo de entrada de altura igual ao registo de saída acrescentado ao sub-grupo precedente. Voltar ao passo 2.

Para utilização correcta de um circuito decomposto desta maneira, o sistema de apoio à execução deve ler o valor do registo extra de saída de cada sub-grupo e copiar esse valor para o registo extra de entrada do sub-grupo seguinte. Todas estas operações são efectuadas automaticamente, sem intervenção do utilizador.

A representação interna de um grupo é feita por instâncias da classe `scluster`:

```
(defclass scluster ()
  ((slices :accessor slices :initarg :slices :type (vector slice)
           :documentation "The slices of the cluster.")
   (slicenets :accessor slicenets :initarg :slicenets :type (vector snetlist)
              :documentation "Vector with the netlist for each slice.")
   (sliceroutes :accessor sliceroutes :initarg :sliceroutes :type (vector routing-region)
                :documentation "Vector with the routes of the inter-slice regions")
   (vd-factor :accessor vd-factor  :type pos-fixnum :initform 1 :initarg :vd-factor
              :documentation "Cluster mobility in the y direction.")
   (hd-factor :accessor hd-factor :type pos-fixnum :initform 1 :initarg :hd-factor
              :documentation "Cluster mobility in the x direction.")
   (width :accessor width :initarg :width :initform 1 :type pos-fixnum
          :documentation "Width of cluster.")
   (height :accessor height :initarg :height :initform 1 :type pos-fixnum
           :documentation "Height of cluster.")
   (place-x :accessor place-x :initarg :place-x :initform 0 :type pos-fixnum
            :documentation "X-coordinate of cluster origin.")
   (place-y :accessor place-y :initarg :place-y :initform 0 :type pos-fixnum
            :documentation "Y-coordinate of cluster origin.")
   (placed :accessor placed :initarg :placed :initform nil :type boolean
           :documentation "Flag indicating if the cluster is placed.")))
```

Cada instância desta classe tem um vector de fatias (`slices`), cada uma das quais é um conjunto de componentes. As ligações entre componentes estão representadas no vector `slicenets`: cada elemento do vector representa todas as ligações entre a correspondente fatia (elemento de `slices`) e a seguinte. À entrada para o módulo PHYS apenas estes dois campos estão preenchidos; os outros são preenchidos à medida que as diversa etapas de processamento são efectuadas.

Após a fase de encaminhamento, o vector `sliceroutes` contém a informação que permite realizar fisicamente as ligações especificadas pelos elementos correspondentes de `slicenets`.

Os campos `width` e `height` contêm as dimensões (em número de blocos lógicos) do rectângulo envolvente da implementação. O processamento de um grupo de componentes é feito de forma independente do local em que será colocado; apenas quando é necessário gerar os dados para programação do circuito FPGA é que é necessário saber exactamente a posição do grupo. Nessa altura, o campo `placed` assume o valor "verdadeiro", e os campos `place-x` e `place-y` contêm as coordenadas do canto inferior esquerdo do rectângulo envolvente.

A implementação dos componentes pode impor restrições aos locais em que estes podem ser colocados. Componentes que apenas usem interligações locais podem ser colocados em qualquer local; componentes que usem internamente recursos de interligação partilhados entre grupos de blocos lógicos devem ser colocados de maneira a que esses recursos possam ser usados. No caso concreto do circuito XC6216, tal significa que as suas coordenadas verticais ou horizontais (ou ambas) sejam múltiplas de 4, de 16 ou de 64, dependendo do tipo de recurso que usam (conforme descrito na página 39). Estas restrições propagam-se ao grupo que contém esses componentes e são registadas nos campos `hd-factor` e `vd-factor`, em que "hd" ("vd") é uma abreviatura de *horizontal displacement* (*vertical displacement*). O valor destes campos pode ser 1 (sem restrições), 4, 16 ou 64.

A classe `scluster` é usada na definição de `operator-network-impl` para representar a implementação de cada grupo. Não existe nenhuma classe para representar sub-grupos; a decomposição de um grupo é simplesmente representada por um vector de instâncias de `scluster` (pág. 181).

6.2 Componentes e geradores

Da mesma forma que operadores são os objectos mais complexos do modelo funcional, também componentes têm um papel primordial no modelo físico. Em particular, os componentes constituem uma parte fundamental do mecanismo de reconfiguração dinâmica. No sistema ReDiFlex, a informação necessária para esta tarefa está encapsulada nos componentes e nos métodos que lhes estão associados.

6.2.1 Representação de componentes

A representação interna de componentes distingue essencialmente três situações: componentes imutáveis (i.e., que não suportam reconfiguração dinâmica), componentes mutáveis e componentes contentores. Os dois últimos constituem extensões do primeiro. A relação entre as diversas classes está ilustrada na figura 6.3 e é descrita a seguir.

A classe `sblock` (*structural block*) agrupa a informação geral sobre componentes, tanto os que têm realidade física como os virtuais. Estes últimos são apenas usados durante a fase de

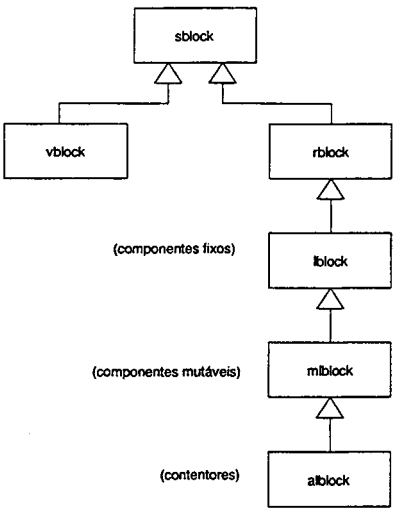


Figura 6.3: Classes de componentes.

colocação (secção 6.3.1).

```
(defclass sblock ()
  ((width :accessor width :initform 1 :type pos-fixnum :initarg :width
          :documentation "The width of the block in number of cells.")
   (height :accessor height :initform 1 :type pos-fixnum :initarg :height
           :documentation "The height of the block in number of cells.")
   (place-x :accessor place-x :initform 0 :type pos-fixnum :initarg :place-x
            :documentation "Horizontal coordinate of placed block.")
   (place-y :accessor place-y :initform 0 :type pos-fixnum :initarg :place-y
            :documentation "Vertical coordinate of placed block.")
   (placed :accessor placed :initform nil :type boolean :initarg :placed
           :documentation "Flag indicating whether sblock is placed")
   (hd-factor :initform 1 :accessor hd-factor :type pos-fixnum :initarg :hd-factor
              :documentation "Allowed x-coordinates are multiple of this factor")
   (vd-factor :initform 1 :accessor vd-factor :type pos-fixnum :initarg :vd-factor
              :documentation "Allowed y-coordinates are multiple of this factor")))
```

A classe `sblock` contém a informação geométrica associada a um bloco; os seus campos têm o mesmo significado que os campos correspondentes de `scluster`. Uma instância de `sblock` pode estar num de dois estados, conforme o valor do campo `placed`. Este valor é "verdadeiro" se o bloco tiver uma posição fixa na infra-estrutura reconfigurável; nesse caso, `place-x` e `place-y` contêm as coordenadas absolutas do componente. Enquanto este ainda não estiver

definitivamente colocado, os campos mencionados contêm as coordenadas relativas ao canto inferior esquerdo do grupo a que o componente pertence.

```
(defclass rblock (sblock)
  ((id :accessor id :initarg :id
       :type string :initform (default-rblock-id)
       :documentation "Name of the block.")
   (type :accessor type :initarg :type
        :type string
        :documentation "Name of the component class.")))

(defclass vblock (sblock) ())
```

Um componente com existência física (classe `rblock`) tem um identificador (`id`) e um indicador do tipo de componente (p. ex.,somador ou comparador). O indicador de tipo é usado apenas como etiqueta informativa. O algoritmo de colocação recorre a componentes "virtuais" para representar espaços livres entre componentes de uma fatia. Esses componentes são do tipo `vblock`.

Os componentes imutáveis são representados pela classe `lblock` (*logical block*):

```
(defclass lblock (rblock)
  ((inputs :accessor inputs :type (simple-vector cons) :initarg :inputs
           :documentation "Array of inputs in relative position.")
   (outputs :accessor outputs :initform nil
            :type (simple-vector pos-fixnum) :initarg :outputs
            :documentation "Array of outputs in relative position.")
   (clocked :accessor clocked :initform nil :type boolean :initarg :clocked
            :documentation "True if block uses the global clock.")
   (clearable :accessor clearable :initform nil :type boolean :initarg :clearable
              :documentation "True if block uses the global clear.")
   (active-low-terminals :accessor active-low-terminal :type simple-bit-vector
                         :documentation "Mark terminal as active low.")
   (register-list :accessor register-list :type cons :initarg :register-list :initform nil
                  :documentation "List of available user registers.")
   (core-builder-fn :accessor core-builder-fn :type function
                    :initarg :core-builder-fn :initform #'make-lblock-core
                    :documentation "The function that builds the appropriate core")
   (core :accessor core :type cell-plane :initarg :core
         :documentation "cell array implementing the desired functionality")
   (clock-cells :accessor clock-cells :initform nil :type list :initarg :clock-cells
                :documentation "Absolute cell positions where CLK must be activated")
   (clear-cells :accessor clear-cells :initform nil :type list :initarg :clear-cells
                :documentation "Absolute cell positions where CLR  must be activated")))
```

Os campos `inputs` e `outputs` especificam respectivamente as entradas e saídas do componente. Uma entrada lógica pode ter vários terminais físicos. As saídas têm sempre apenas um terminal físico.

Os campos `clocked` e `clearable` indicam se o componente deve ser ligado aos sinais globais de relógio e inicialização. Na versão actual, todos os componentes sequenciais, i.e. aqueles com `clocked` a "verdadeiro", necessitam apenas de um ciclo de relógio para executar a sua função.

O vector `active-low-terminals` indica quais os terminais físicos que são activos baixo. A possibilidade de designar certos terminais como activos ao valor lógico "0" permite produzir componentes mais compactos, já que a inversão pode ser compensada modificando a polaridade das interligações.

O campo `register-list` contém a especificação de todos os registos acessíveis do componente; a lista está sempre vazia para componentes combinacionais. A especificação da posição dos registos é feita relativamente ao canto inferior esquerdo do componente.

O conteúdo dos campos precedentes está fixado desde a criação do componente. Inicialmente este não contém nenhuma informação sobre a configuração do interior do componente. Essa informação, que é guardada no campo `core`, só é gerada depois de o componente ser definitivamente colocado, porque a informação de configuração depende dos recursos que devem ser configurados, que, por sua vez, variam com a posição das células configuráveis. Embora todos os blocos lógicos do circuito FPGA tenham os mesmos recursos lógicos, os seus recursos de interligação variam com a posição em que se situam na grelha do circuito FPGA, conforme visto na primeira parte (pág. 39).

A conteúdo do campo `core` é gerado quando necessário usando a função `core-builder-fn`. Em geral, não é necessário chamar explicitamente esta função; qualquer acesso ao campo `core` resulta na invocação automática de `core-builder-fn` conforme especificado pelo seguinte fragmento de código:

```
1  (defmethod core :around ((lb lblock))
2    "Create core if necessary and possible."
3    (if (slot-boundp lb 'core)
4        (call-next-method)
5        (if (placed lb)
6            (multiple-value-bind (c lst1 lst2) (funcall (core-builder-fn lb)
7                                                        (width lb) (height lb)
8                                                        (place-x lb) (place-y lb))
9              (unless c
10               (error "Implicit core generation failed."))
11             (setf (core lb) c
12                   (clock-cells lb) lst1
13                   (clear-cells lb) lst2)
```

```
14              c)
15          (error "LBLOCK is unplaced: cannot access core"))))
```

Quando é efectuado uma acesso ao campo core, o método do mesmo nome verifica se a informação de configuração já foi gerada; nesse caso o acesso prossegue normalmente (linha 4). No caso contrário, verifica-se o o componente já está colocado e, em caso afirmativo, invoca-se a função core-builder-fn (linha 6). A função retorna a informação de configuração a guardar em core, bem como os dados para os campos clock-cells e clear-cells. Estes dois últimos itens de informação são necessários, porque, dependendo da posição em que o componente for colocado, a activação dos sinais globais de relógio e inicialização pode ter de ser feita em células exteriores ao componente. Para ilustrar a necessidade desta informação, considere-se o sinal global de inicialização, que é partilhada por grupos de 16 blocos lógicos de cada coluna. Por exemplo, na primeira coluna do circuito XC6216, o bloco lógico 15 controla a distribuição do sinal de inicialização aos blocos 0-15. Este bloco lógico terá de ser adequadamente configurado caso um componente que use este sinal global ocupar os blocos 7–11, por exemplo. Os campos clear-cells e clock-cells contêm a informação necessária para o sistema, no momento de configurar a infra-estrutura, activar a distribuição dos sinais globais fora dos limites dos componentes. Qualquer activação necessária no interior do componente deve ser incluída na informação guardada no campo core.

Deve-se notar que um componente imutável não precisa de registar os parâmetros usados para a sua construção, já que a sua configuração (i.e. o conteúdo do campo core nunca é alterada depois de gerada. A função core-builder-fn particular a cada instância já inclui o conhecimento desses parâmetros. No caso dos componentes mutáveis, o valor dos parâmetros que podem influenciar a reconfiguração devem ficar registados no objecto que representa o componente. Para tal, o sistema ReDiFlex usa a classe mlblock, uma extensão simples de lblock.

```
(defclass mlblock (lblock)
  ((fixed-params :accessor fixed-params :initarg :fixed-params
                 :initform nil :type cons
                 :documentation "Fixed parameters and their values.")
   (modifiable-params :accessor modifiable-params :initarg :modifiable-params
                      :initform nil :type cons
                      :documentation "Modifiable parameter name and default spec.")
   (reconf-fn :accessor reconf-fn :initarg :reconf-fn :type function
              :documentation "Function that rebuilds the core of the block.")))
```

Para além da informação já contida em lblock, esta classe tem ainda a especificação dos parâmetros que influenciam as operações de reconfiguração. Os parâmetros estão divididos em duas categorias, os imutáveis (fixed-params) e os modificáveis (modifiable-params). A presença dos primeiros é necessária, porque a geração da nova configuração pode depender

da relação entre valores dos parâmetros fixos e modificáveis. O campo `reconf-fn` contém a função encarregada de refazer a configuração (o valor do campo `core`) para satisfazer os valores actuais dos parâmetros. A função `reconf-fn` pode usar qualquer método apropriado para gerar a nova configuração: em particular, tanto pode gerar uma configuração nova como alterar a configuração existente. A função apenas pode modificar o conteúdo do campo `core`; os outros campos (incluindo a especificação de entradas e saídas, registos e sinais globais) não podem ser alterados.

A implementação física de contentores fica a cargo da classe `alblock`:

```
(defclass alblock (mlblock)
  ((members :accessor members :initarg members
            :type block-set :initform (make-empty-block-set)
            :documentation "Set of rblocks that belong to this aggregate.")
   (current-blk :accessor current-blk :type lblock
                :documentation "The current block.")))
```

Esta classe implementa a "concha" que envolve a implementação dos componentes constituintes (fig. 5.4). A classe inclui o conjunto de componentes constituintes (`members`, bem como o componente activo (`current-blk`). A especificação geométrica deste componente deve permitir acomodar os sub-componentes no seu interior (um de cada vez). Em particular todos os componentes partilham os mesmos terminais de entrada e saída, bem como a mesma especificação de sinais globais. Esta última restrição não é limitativa: a especificação dos sinais globais é simplesmente a união das especificações respectivas dos componentes constituintes. Tudo o que é necessário assegurar é que os sinais globais estejam disponíveis em qualquer bloco lógico em que sejam usados; se estiverem acessíveis em blocos lógicos onde não sejam usados, não têm qualquer influência sobre o funcionamento lógico do bloco[1]. Pela mesma razão não há problema em activar os sinais de relógio ou inicialização para um componente combinacional: este simplesmente não é sensível ao valor do sinal globais. Consequentemente, componentes combinacionais e sequenciais podem fazer parte do mesmo contentor.

O facto de os componentes constituintes terem necessariamente idêntico número de entradas e saídas não implica que estas devam ter a mesma localização, nem que o número de terminais físicos (por oposição ao número de entradas e saídas lógicas) deva ser igual.

Os contentores têm apenas um parâmetro modificável, designado por `:personality`, tal como nos operadores correspondentes. Os contentores usam todos a mesma implementação das funções `core-builder-fn` e `reconf-fn`. A primeira gera a informação geométrica de todos os sub-componentes para determinar a constituição da "concha" envolvente, criando uma configuração "vazia" para o seu interior. A sua invocação, que é ligeiramente diferente das

[1]Pode haver um aumento do consumo de potência porque existirão comutações desnecessárias dos *flip-flops*. No contexto actual tal facto não constitui um problema.

funções equivalentes das outras classes, é controlada por uma versão do método core já descrito adaptada à nova situação. Esse método termina efectuando uma primeira chamada à função `reconf-fn` para "colocar" o primeiro sub-componente no centro da concha. Esta função também é chamada quando é preciso modificar o interior da "concha" na sequência de uma modificação do parâmetro `:personality`. Esta tarefa envolve copiar a configuração do sub-componente para o interior da "concha" e estabelecer as ligações entre os terminais do sub-componente e da concha. Esta última tarefa é levada a cabo por uma versão restrita do algoritmo de encaminhamento da secção 6.3.2.

Cada instância da classe `op-container` (pág. 166) tem no seu campo `blk` uma referência para um objecto da classe `alblock`. Por seu lado o campo `sub-operators` da classe `op-container` tem referências aos operadores constituintes, que por sua vez têm uma referência ao respectivo componente (o conteúdo do campo `blk` dos sub-operadores); esses componentes são os mesmos objectos que são referenciados pelo campo `members` de `alblock`.

Consolidação dos sinais globais

Após as fases de colocação e encaminhamento, é necessário ainda coligir a informação associada aos sinais globais de relógio e inicialização. O sistema consolida toda essa informação individual (já que as mesmas células externas podem ser referenciadas em componentes diferentes) e guarda a informação assim obtida em campos apropriados da estrutura que representa cada fatia (a classe `slice`). A função `adjust-global-cluster-signals` é responsável por esta operação.

Da mesma forma, é necessário processar a informação sobre os registos internos de cada componente, com dois objectivos:

- Obter uma interface uniforme para os registos internos de cada componente de um grupo;
- Adaptar a informação sobre cada registo (que cada componente especifica em termos relativos) à posição definitiva em que ficou colocado o componente. Isto é necessário, porque o acesso se faz em termos da posição absoluta do registo; também a máscara de selecção dos bits que compõem o registo deve ser adaptada à nova posição.

Esta tarefa é executada pela função `make-cluster-interface`. A informação gerada é guardada na posição do vector `setups` de `operator-networ-impl` (pág. 181) correspondente ao grupo em processamento, e posteriormente usada pelas rotinas do módulo de apoio à execução para efectuar o acesso aos referidos registos.

6.2.2 Geração de componentes

No sistema ReDiFlex, a tarefa de geração de componentes está repartida por três funções:

- **Criação.** A primeira tarefa da geração de um componente consiste em construir uma instância apropriada da classe `lblock` ou derivadas. Essa tarefa é realizada por uma função especial para cada tipo de componente. Essa função é posteriormente invocada por `generate-operator-block` (pág. 168).

- **Geração de configuração.** A geração da configuração inicial de um componente é efectuada pela função referenciada pelo campo `core-builder-fn`.

- **Reconfiguração.** Para componentes mutáveis, a função do campo `reconf-fn` é responsável por transformar a informação de configuração de maneira a corresponder aos actuais valores dos parâmetros do componente. Esta operação só se aplica a instâncias da classe `mlblock` ou derivadas desta.

A definição destas funções é a segunda parte da tarefa do "criador de bibliotecas" (a primeira é definir os operadores funcionais apropriados, ver pág. 171). As funções `core-builder-fn` e `reconf-fn` são as únicas que trabalham directamente com a representação dos recursos existentes no circuito FPGA (sec. 7.2.1). A definição do método `mutate` associado a componentes reconfiguráveis é feita precisamente à custa desta última função:

```
1  (defmethod mutate ((blk mlblock) new-params)
2    "Generates a new version of the core."
3    (when (and (slot-boundp blk 'core)
4               (slot-boundp blk 'reconf-fn))
5      (dolist (p new-params)
6        (let ((p-attribs (assoc (car p) (modifiable-params blk) :test #'eq)))
7          (unless (and p-attribs
8                       (funcall (mparam-val-fn (cdr p-attribs)) (cdr p) blk))
9            (return-from mutate nil))))
10     (dolist (p new-params)
11       (set-param-value blk (car p) (cdr p)))
12     (funcall (reconf-fn blk) blk)))
```

A função começa por verificar o estado do componente (linha 3 e seguinte), para depois processar os novos parâmetros (linha 5 e seguintes) e, caso estes sejam válidos, proceder à respectiva actualização (linha 10 e seguinte). Por fim é chamada a função de reconfiguração do componente (linha 12).

A título de ilustração, apresentam-se de seguida as funções usadas para implementação dos operadores `const-adder` e `mutable-const-adder`. A estrutura de um somador constante está ilustrada na figura 6.4. Trata-se de um somador do tipo *carry-adder* em que cada andar é especializado para o valor do bit correspondente da constante. Para além disso, em alguns caso é possível determinar que o transporte do andar anterior é nulo, pelo que ao todo se

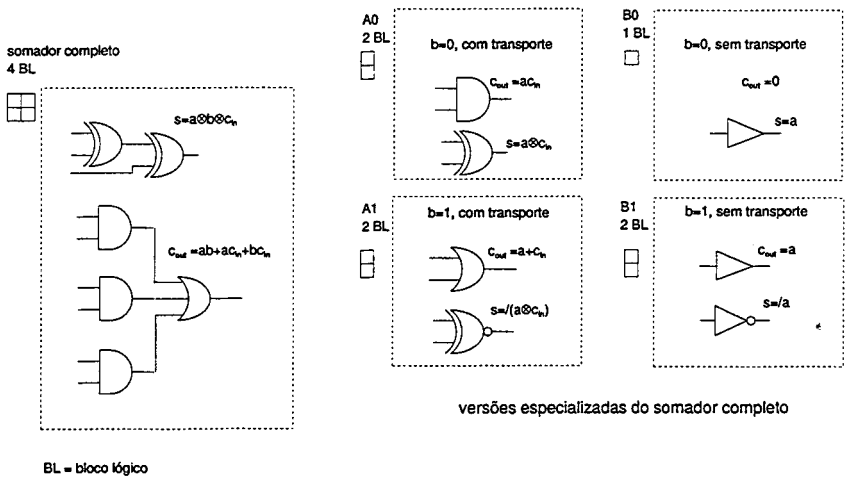


Figura 6.4: Estrutura interna de um somador constante.

podem efectuar quatro especializações do andar genérico. As versões especializadas de cada andar não têm todas o mesmo tamanho.

A construção de um somador constante é efectuada pela função `gen-fixed-const-adder`:

```
(defun gen-fixed-const-adder (size const)
  "Generate the configuration for a adder to constant with SIZE bits."
  (check-type size (integer 1 32))
  (unless (and (or (plusp const) (zerop const))
               (<= (integer-length const) size))
    (error "Constant ~a does not fit in ~a bits." const size))
  (let* ((number-of-0-lsb (if (zerop const) size (first-bit-1 const)))
         (hi (if (zerop const) size (+ number-of-0-lsb
                                       (* 2 (- size number-of-0-lsb)))))
         (lb (make-instance 'lblock  :width 1 :height hi
                            :type "fixed-const-adder"
                            :inputs (make-array size
                                                :element-type 'cons)
                            :outputs (make-array size
                                                 :element-type 'pos-fixnum)
                            :core-builder-fn
                            #'(lambda (w h x y)
                                (fill-fc-adder w h x y size const number-of-0-lsb)))))
    ;; Terminal information
    (let ((i 0))
```

```
21      (dotimes (j size)
22        (setf (svref (inputs lb) j) (if (< i number-of-0-msb) (list i) (list i  (1+ i))))
23        (setf (svref (outputs lb) j) i)
24        (incf i (if (< i number-of-0-msb) 1 2))))
25    lb))
```

O código nas linhas 3–6 verifica se os argumentos da funções têm valores correctos; em particular não é possível construir somadores com mais de 32 bits, já que não se poderia garantir que caberiam no circuito FPGA.

A altura total do componente depende da constante: se os bits menos significativos da da constante forem iguais a zero, é possível usar uma especialização que ocupa apenas um bloco lógico. A linha 8 determina a altura exacta do componentes.

As linhas 10 e seguintes servem para criar uma instância da classe `lblock` devidamente inicializada. A linha 17 mostra a definição da função de configuração; neste caso é simplesmente a invocação da função `fill-fc-adder` com parâmetros adequados. A definição mostra como se podem encapsular os parâmetros necessários à criação da configuração (`size`, `const` e `number-of-0-lsb`) na chamada à função auxiliar (o que explica porque não é necessário prever a representação explícita dos parâmetros num campo da classe `lblock`).

Para o circuito FPGA usado, o somador constante tem menor área que um somador regular do mesmo tipo por duas factores:

1. A largura de cada andar é menor: o somador constante tem apenas um bloco lógico de largura, enquanto um somador regular tem duas.

2. Dependendo do valor da constante, o somador constante pode ser mais baixo que o regular.

A função `fill-fc-adder` é que faz o verdadeiro trabalho de configuração:

```
1  (defun fill-fc-adder (width height x y size const nb0msb)
2    "Fill in for the first time the block's core."
3    (declare (type (integer 1 63) width height) (type pos-fixnum x y)
4             (type (integer 1 32) size nb0msb))
5    (unless (= 1 width)
6      (error "Size mismatch in FILL-FC-ADDER."))
7    (let ((core (make-lblock-core width height x y))
8          (i 0))
9      (dotimes (j size)
10       (if (< j nb0msb)
11           (let ((c (aref core 0 i)))
12             (prog-cell-B0 c)
13             (incf i))
```

```
14          (let ((c0 (aref core 0 i))
15                (c1 (aref core 0 (1+ i))))
16            (if (= j nb0msb)
17                (prog-cells-B1 c0 c1)
18                (if (logbitp j const)
19                    (prog-cells-A1 c0 c1)
20                    (prog-cells-A0 c0 c1)))
21            (incf i 2))))
22    (values core nil nil)))
```

As linhas iniciais (até à linha 6) servem para verificar a consistência dos argumentos. A linha 7 cria uma configuração de base (equivalente à configuração de *hardware* após inicialização). O ciclo iniciado na linha 9 gera a configuração dos diferentes andares (a começar pelo menos significativo), recorrendo às funções auxiliares `prog-cell-B0` (linha 12), `prog-cell-B1` (linha 17), `prog-cell-A1` (linha 19) e `prog-cell-A0` (linha 20). A última linha retorna a configuração gerada; como é um elemento combinacional não retorna nenhuma informação sobre os sinais globais (valores `nil` na linha 22).

As rotinas auxiliares são todas parecidas. Como exemplo, apresenta-se uma das mais complexas, `prog-cells-A0`, que configura dois blocos lógicos para implementar a especialização A0 da figura 6.4.

```
1  (defun prog-cells-A0 (c0 c1)
2    (declare (type cell c0 c1))
3    (setf (cs c0) (mux-index 'functional-cell 'cs '(not c))
4          (cs c1) (mux-index 'functional-cell 'cs '(not c)))
5    (setf (x1 c0) (mux-index 'cell 'x1 '(not e))
6          (x2 c0) (mux-index 'cell 'x2 '(not n))
7          (x3 c0) (mux-index 'cell 'x3 '(not n))
8          (y2 c0) (mux-index  'functional-cell 'y2 'x2)
9          (y3 c0) (mux-index  'functional-cell 'y3 '(not x3))
10         (e c0) (mux-index 'cell 'e '(not f))
11         (n c0) (mux-index 'cell 'n '(not n)))
12   (when (typep c0 'cell-n4)
13     (setf (alt-north-mux c0) (mux-index 'cell-n4 'alt-north-mux '(not n))))
14   (setf (x1 c1) (mux-index 'cell 'x1 '(not e))
15         (x2 c1) (mux-index 'cell 'x2 '(not e))
16         (x3 c1) (mux-index 'cell 'x3 '(not n))
17         (y2 c1) (mux-index 'functional-cell 'y2 'x2)
18         (y3 c1) (mux-index 'functional-cell 'y3 '(not x3))
19         (n c1) (mux-index 'cell 'n '(not f)))
20   (when (typep c1 'cell-n4)
21     (setf (alt-north-mux c1) (mux-index 'cell-n4 'alt-north-mux '(not f)))))
```

A implementação é conceptualmente muito simples. O código da linha 3 desactiva os *flip-flops*. De seguida, configura-se o bloco que implementa a soma (linha 5 e seguintes) e a passagem do transporte do andar anterior para o bloco seguinte, onde é gerado o novo sinal de transporte; por fim, configura-se o bloco que gera o transporte (linha 14 e seguintes) para o próximo andar.

A função geradora de um `mutable-const-adder` tem uma estrutura semelhante. Neste caso, não se pode optimizar a altura do componente, porque a alteração da constante poderia levar à modificação de geometria do componente; por consequência, todos os andares são implementados por dois blocos lógicos.

```
1  (defun gen-mutable-const-adder (size const)
2    "Generate the configuration for a adder to constant with SIZE bits."
3    (check-type size (integer 1 32))
4    (unless (and (or (plusp const) (zerop const))
5                 (<= (integer-length const) size))
6      (error "Constant ~a does not fit in ~a bits." const size))
7    (let* ((lb (make-instance 'mlblock  :width 1 :height (* 2 size)
8                              :type "mutable-const-adder"
9                              :inputs (make-array size
10                                                 :element-type 'cons)
11                             :outputs (make-array size
12                                                  :element-type 'pos-fixnum)
13                             :core-builder-fn
14                             #'(lambda (w h x y) (fill-mc-adder w h x y size const))
15                             :fixed-params (list (cons :size size))
16                             :modifiable-params (list (cons :const (cons const
17                                                                    #'valid-const-mca)))
18                             :reconf-fn #'reconf-mc-adder)))
19     ;;  Terminais
20     (dotimes (j size)
21       (setf (svref (inputs lb) j) (list (* 2 j) (1+ (* 2 j))))
22       (setf (svref (outputs lb) j) (* 2 j)))
23     lb))
```

A organização de `gen-mutable-const-adder` é similar à de `gen-fixed-const-adder`, excepto que o componente criado é da classe `mlblock` (linha 7). A função `fill-mc-adder` (linha 13) é muito semelhante à função `fill-fc-adder` usada para o somador constante imutável, excepto que a especialização B0 é implementada com dois blocos lógicos. A função de reconfiguração `reconf-mc-adder` simplesmente gera uma nova configuração apropriada:

```
1  (defun reconf-mc-adder (blk)
2    "Update configuration core for mutable constant adder."
```

```
3   (let ((core (core blk))
4         (i 0)
5         (const (get-param-value blk :const))
6         (size (get-param-value blk :size)))
7     (let ((c0 (aref core 0 i))
8           (c1 (aref core 0 (1+ i))))
9       (if (logbitp 0 const)
10          (prog-cells-B1 c0 c1)
11          (prog-cells-B0 c0 c1))
12      (incf i 2)
13      (dotimes (j (1- size))
14        (let ((c0 (aref core 0 i))
15              (c1 (aref core 0 (1+ i))))
16          (if (logbitp (1+ j) const)
17              (prog-cells-A1 c0 c1)
18              (prog-cells-A0 c0 c1)))
19        (incf i 2))
20      t)))
```

A função `reconf-mc-adder` trata o primeiro andar especialmente (linha 7 e seguintes), porque não existe transporte de um andar anterior. Os andares seguintes assumem que pode existir sempre transporte (linha 13 e seguintes), já que não é possível aproveitar as situações especiais, porque envolveriam modificações da geometria do componente.

6.3 Processamento do modelo físico

Dado um objecto da classe `scluster` com os vectores `slices` e `slicenets` devidamente preenchidos, o processamento do modelo físico é feito em três fases: colocação parcial dos componentes, encaminhamento e colocação final, e determinação da informação final global sobre os sinais de relógio/inicialização e registos internos. O último passo é relativamente directo e já foi descrito na página 198. Os outros dois formam o âmago do processamento do módulo físico.

6.3.1 Colocação de componentes

A determinação da posição dos componente é feita em duas fases. Na primeira, determina-se a posição dos componentes na fatia; na segunda, determina-se a posição da fatia. A segunda fase é feita simultaneamente com o encaminhamento.

Para a colocação dos componentes no interior da fatia é usada um método heurístico, enquanto o encaminhamento é feito por pesquisa exaustiva com `backtracking`. Em qualquer

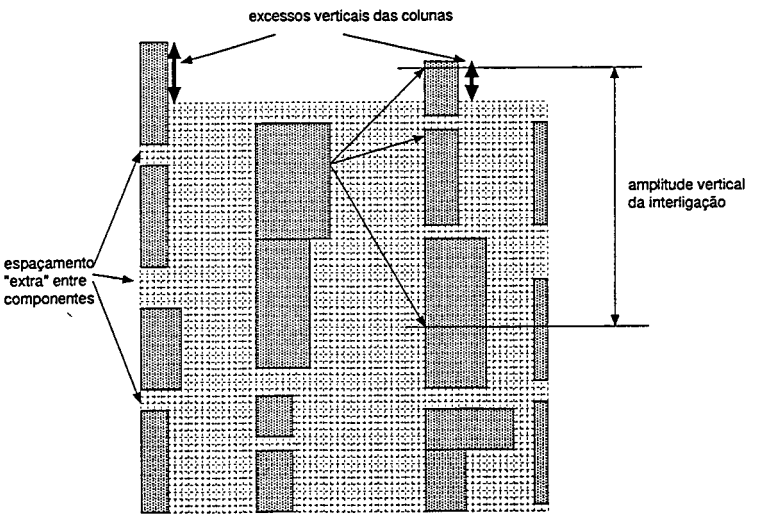


Figura 6.5: Função objectivo usada durante a fase de colocação.

dos casos, a implementação está organizada de maneira a ser relativamente expedito alterar os parâmetros dos algoritmos ou experimentar abordagens alternativas.

A heurística usada na colocação dos componentes é baseada numa pesquisa local no espaço das soluções. Neste contexto, uma solução é representada pela sequência dos componentes de cada fatia (a partir da ordem dos componentes de cada fatia é possível determinar as suas posições verticais). A solução é viável se a altura de todas as fatias não ultrapassar um limite máximo imposto pela arquitectura do circuito FPGA. A colocação dos componentes é determinada simultaneamente para todas as fatias, e não fatia a fatia.

Dada uma solução, a heurística de pesquisa local gera aleatoriamente várias soluções vizinhas (obtidas por alterações simples da solução inicial) e escolhe a melhor para nova solução. Se a nova solução não for superior à solução original, termina a busca. A busca também termina se for atingido o mínimo global da função objectivo, i.e. da função que determina a qualidade de cada solução.

A função objectivo escolhida é a soma pesada de três factores (fig. 6.5):

1. O excesso em altura das fatias em relação ao limite imposto pela arquitectura do circuito FPGA, acumulado para todas as fatias. Este valor tem de ser nulo para que a solução seja viável.

2. O espaçamento extra entre os componentes de uma fatia, acumulado para todas as fatias. A diminuição deste factor contribui para a redução da altura global do grupo de fatias.

3. A soma da maior amplitude vertical das interligações que saem de cada fatia. A ampli-

tude vertical máxima é usada como estimativa da distância entre uma fatia e a seguinte. Como o encaminhamento corresponde a determinar percursos sobre uma grelha rectangular, a ligação de pontos com uma dada separação vertical entre eles tem grandes probabilidades de causar um afastamento horizontal proporcional. A diminuição deste parâmetro tende a diminuir o afastamento entre fatias, e, portanto, a diminuir a largura da implementação final.

O peso relativo de cada um dos factores foi determinado empiricamente com base nos circuitos processados pelo sistema ReDiFlex. Os pesos usados são respectivamente 1000, 10 e 5. O peso muito elevado do primeiro factor justifica-se pelo facto de ser imprescindível reduzi-lo a zero para ter uma solução viável. Os pesos podem ser facilmente alterados: embora não se espere que o utilizador "comum" esteja ciente do funcionamento interno do sistema, é um ponto importante para o implementador que pretende avaliar o funcionamento da heurística.

A implementação do algoritmo de colocação vem simplificada se este lidar apenas com componentes. Contudo, uma boa solução não exige que os componentes de uma fatia sejam colocados contiguamente, pelo que é necessário representar os espaços entre componentes da mesma fatia. Para conciliar os dois factos, a implementação usa componentes virtuais (de dimensão 1×1) para representar os espaços.

A entrada para o algoritmo de colocação é a função `simple-slice-placer`:

```
1  (defun simple-slice-placer (clu)
2    "Place blocks in a cluster."
3    (let ((sol (funcall *simple-placement-heuristic* clu)))
4      (when sol
5        (unless (feasible-placer-solution-p sol)
6          (return-from simple-slice-placer nil))
7        (fix-cluster-according-to-solution sol)
8        (unless (initial-horizontal-slice-placement clu sol)
9          (return-from simple-slice-placer  nil))
10       (setf (height clu) (loop for sl across (slices clu)
11                                maximize (height sl)))
12       clu)))
```

Esta função começa por chamar a rotina referenciada pela variável `*simple-placement-heuristic*` para obter a solução para o problema de colocação (linha 3). A referência indirecta à heurística usada, permite mudar a operação de `simple-slice-placer` através da simples atribuição de uma função alternativa àquela variável. De seguida, a rotina verifica se a solução retornada é viável (linha 5); no caso afirmativo, a representação do grupo de fatias é actualizada para corresponder à solução adoptada (linha 7).

Antes de terminar, a função determina heuristicamente uma posição inicial para cada fatia (linha 8). A heurística usada (função `initial-horizontal-slice-placement`) tem tendência

para colocar as fatias relativamente próximas, já que na fase de encaminhamento a distância entre fatias pode aumentar, mas não diminuir (pág. 208). Por fim ainda se procede ao cálculo do valor final do campo `height` da classe `scluster` (linha 10), porque já existe informação suficiente para tal.

No sistema ReDiFlex, a variável `*simple-placement-heuristic*` referencia a função `local-search-placement`:

```
1  (defun local-search-placement (clu)
2    "Places a cluster using local search."
3    (declare (type scluster clu))
4    (let* ((sol (make-initial-placer-solution clu)))
5      (declare (type placer-solution sol current))
6      (when sol
7        (setf (objective-function-value sol) (eval-objective-function sol))
8        (when (solution-unimprovable-p sol)
9          (return-from local-search-placement sol))
10       (loop
11           (unless (funcall *local-search-improver* sol clu)
12             (return)))
13       sol)))
```

Esta função começa por criar uma solução inicial (linha 4) que não é necessáriamente um solução viável. De seguida, calcula o valor da função objectivo (linha 7) e verifica se a solução é óptima (linha 8). Em caso afirmativo, a rotina termina o seu trabalho; em caso contrário, entra num ciclo que invoca repetidamente uma função para obter uma solução melhor (linha 10). A iteração termina quando a função referenciada via `*local-search-improver*` indicar que não consegue encontrar uma solução melhor.

A função usada para procurar uma solução vizinha melhor é `random-first-fit-placer-solution`:

```
1  (defun random-first-fit-placer-solution (sol clu)
2    "Looks randomly for a better solution."
3    (let (rdc
4          (old-mfv (objective-function-value sol))
5          (iter-counter 0))
6      (loop
7          (setf rdc (funcall *local-search-random-change-fn* sol))
8          (when (< (apply-change rdc sol clu) old-mfv)
9            (return sol))
10       (apply-change (revert-change-spec rdc) sol clu)
11       (incf iter-counter)
12       (when (>= iter-counter *local-search-iteration-limit*)
13         (return nil)))))
```

O corpo da função é constituído por um ciclo (linha 6 e seguintes) em que é gerada aleatoriamente uma modificação da solução actual (através da função associada a `*local-search-random-change-fn*`). Se a aplicação dessa modificação resultar num valor menor da função objectivo (linha 8), a modificação é aceite e a função retorna. Caso contrário, a modificação é desfeita (linha 10) e um contador de iterações é incrementado. Se esse contador atingir um valor limite (`*local-search-iteration-limit*`, por omissão 30) a função termina sem sucesso, senão passa a nova iteração.

A modificação da solução é implementada pela rotina `simple-random-change-no-gaps` que troca a posição de dois componentes reais escolhidos aleatoriamente de uma fatia qualquer (os componentes virtuais que representam o espaçamento vertical entre componentes reais não são considerados). O sistema inclui uma função chamada `random-multi-change-no-gaps` que troca a posição de dois componentes escolhidos aleatoriamente em cada fatia. Por omissão, o sistema usa a primeira função. Nos testes empíricos efectuados, a segunda função não proporcionou melhorias significativas dos resultados que compensassem o aumento de tempo de processamento associado ao seu uso.

6.3.2 Encaminhamento de interligações

Devido à estrutura do modelo físico, a tarefa de encaminhamento das interligações de um grupo de n fatias é naturalmente decomposta em $n-1$ sub-tarefas independentes, que consistem em encaminhar as interligações entre cada fatia e a seguinte. O facto de as sub-tarefas serem independentes limita a dimensão do problema: no caso do circuito XC6216 existem no máximo 64 terminais de saída e 64 terminais de entrada. Apenas em casos excepcionais é que estes valores são atingidos na realidade; frequentemente o número de terminais envolvidas é significativamente menor.

Dada a estrutura do problema, o sistema ReDiFlex implementa a seguinte abordagem:

1. Usar uma heurística para colocar as n fatias de um grupo. Este trabalho é realizado pela função `initial-horizontal-slice-placement` (pág. 206).

2. Para cada par de fatias consecutivas $(i, i+1)$, tentar realizar o encaminhamento de todas as ligações da fatia i para a fatia $i+1$. Se tal não for possível, incrementar a distância entre as fatias i e $i+1$ (o que envolve incrementar a posição horizontal de todas as fatias j, com $j > i+1$). Se a largura do grupo ultrapassar o limite imposto pela infra-estrutura reconfigurável, terminar[2], senão tentar novamente.

3. Terminado o processamento das interligações entre um par de fatias, passar ao seguinte se possível: se $i < n-1$ incrementar i e voltar ao passo 2, senão terminar.

[2]Em geral esta condição não se verifica na implementação actual do sistema ReDiFlex, porque, conforme já foi referido anteriormente, o processamento é executado como se o circuito FPGA fosse infinitamente largo, procedendo-se posteriormente a uma decomposição, se necessário.

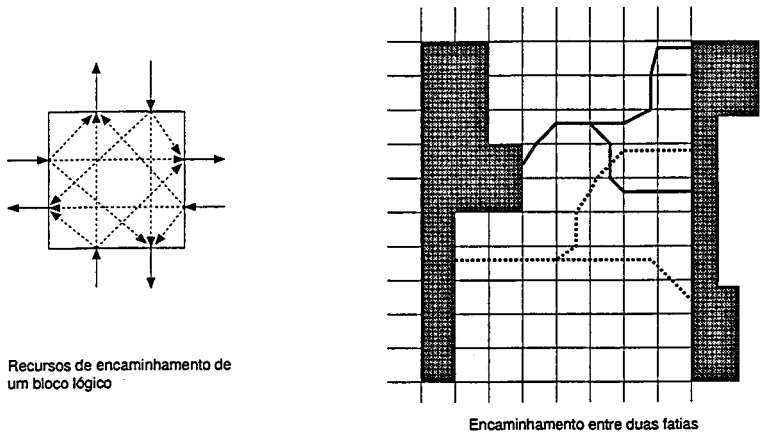


Figura 6.6: Modelo de encaminhamento.

O encaminhamento das interligações entre duas fatias consecutivas é feito por pesquisa exaustiva. Quando o encaminhamento de uma interligação falha, uma das interligações já efectuadas é desfeita e uma nova tentativa é efectuada. O algoritmo admite uma implementação recursiva simples, embora, na prática, a gestão de toda a informação associada ao processo de encaminhamento complique a implementação.

O encaminhamento é feito usando apenas as ligações locais entre blocos lógicos (fig. 6.6). O estabelecimento de um percurso "ocupa" os recursos de encaminhamento das células por onde passa (multiplexadores e terminais de entrada/saída).

Um percurso entre dois pontos é estabelecido por pesquisa exaustiva: a partir da origem, tenta-se estender o percurso para a direita; se tal for possível, usa-se a nova posição como ponto de partida para a próxima etapa do percurso. Embora todas as direcções sejam tentadas, o algoritmo privilegia a movimentação na direcção do terminal de destino. As direcções são tentadas pela seguinte ordem: direita, para cima ou para baixo, consoante o terminal desejado esteja acima ou abaixo do ponto actual, a direcção complementar à anterior e, por último, para a esquerda. Quando não é possível continuar o percurso a partir de um dado ponto, o movimento que permitiu atingir esse ponto é anulado, e um movimento alternativo é tentado, até se ter o percurso completo ou terem falhado todas as tentativas.

O mesmo princípio é aplicada às interligações entre um terminal de origem e vários terminais de destino. Para cada uma, o algoritmo determina um percurso entre a origem e um dos terminais de destino. Se conseguir, tenta construir um percurso entre algum ponto do percurso já construído e outro terminal de destino. O algoritmo prossegue assim até ter construído um percurso em forma de árvore da origem da interligação até todos os terminais de destino. Se, em alguma fase deste processo, não for possível estabelecer um percurso, o algoritmo volta

atrás na última escolha feita e faz uma escolha alternativa. As escolhas são feitas de forma a que todas as permutações possíveis sejam analisadas.

O encaminhamento de todas as interligações também se faz segundo o mesmo princípio: se o encaminhamento de alguma interligação falhar, o percurso de uma das interligações já encaminhadas é desfeito e o encaminhamento da interligação actual é tentado de novo. Também neste caso são tentadas todas as ordens possíveis para processamento das interligações.

Os cuidados a ter na implementação residem principalmente em não deixar de tentar alternativas possíveis, sem no entanto fazer tentativas que se possa vir a determinar virem a ser infrutíferas. Por exemplo, quando falha o encaminhamento de uma interligação I, não vale a pena tentar encaminhar outra ainda não processada, porque posteriormente continuaria a ser impossível encaminhar I; em vez disso, é necessário anular o encaminhamento de uma interligação já processada J, tentar encaminhar I e, se esta tentativa tiver sucesso, tentar então encaminhar J de novo.

A pesquisa exaustiva é um processo com complexidade temporal exponencial, pelo que a sua aplicação deve ser feita com cuidado. A abordagem é viável no contexto do modelo físico usado, porque é aplicada a sub-problemas relativamente pequenos. Nos exemplos em que o sistema ReDiFlex foi usado, a complexidade exponencial não se manifestou adversamente. Contudo isso não garante que para outros exemplos não venha a ter um mau desempenho, o que tornaria necessário implementar uma abordagem alternativa. Por essa razão houve uma preocupação especial em separar a implementação do encaminhamento do resto do sistema. Uma nova implementação deve providenciar uma alternativa à função `simple-slice-router`. Esta função aceita como argumento um objecto da classe `scluster` e o índice da fatia de origem das interligações a encaminhar. A função `simple-router` invoca repetidamente `simple-slice-router` para cada fatia do grupo (excepto a última).

Quando a função `simple-slice-router` executa com sucesso deixa no vector `sliceroutes` de `scluster` uma descrição simbólica da configuração dos recursos de encaminhamento. Essa descrição pode ser usada directamente para configurar a infra-estrutura de *hardware* (sec. 7.2.2).

Em algumas casos, é possível que duas fatias possam simplesmente ser "encostadas" uma à outra, sem necessidade de encaminhamento explícito. Trata-se de uma situação que ocorre com alguma frequência em circuitos simples. O sistema ReDiFlex detecta essas situações e não invoca o algoritmo de encaminhamento para esse caso.

O gerador de configurações dos contentores (pág. 197) usa a mesma abordagem para ligar os terminais do sub-componente aos terminais da "concha" do contentor. Também neste caso a tarefa tem dimensões reduzidas, o que não permite que a complexidade temporal do algoritmo se manifeste adversamente.

O encaminhamento é feito através de multiplexadores de saídas invertidas. Se o sinal passar através de um número ímpar de multiplexadores, chega do destino com polaridade invertida. Durante o encaminhamento, o sistema regista a polaridade de todos os percursos.

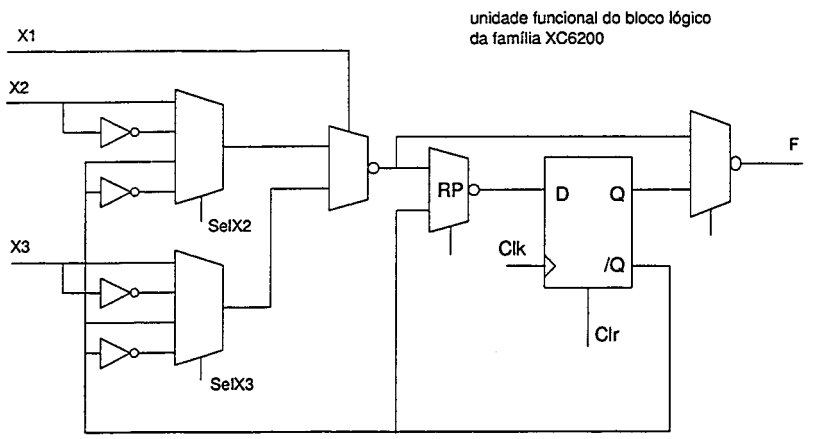


Figura 6.7: Aspecto detalhado da unidade funcional do circuito FPGA XC6200.

Após o encaminhamento estar finalizado, é necessário fazer uma compensação da polaridade em todos os terminais de entrada aonde chegaram sinais com a polaridade inversa da desejada. Isso é feito alterando a configuração do bloco lógico. Por exemplo, e por referência à fig. 6.7, a inversão de polaridade de um sinal ligado a `X2` pode ser compensada pela inversão do controlo do multiplexador, sinal `SelX2`. Os sinais ligados a `X2` e `X3` podem ser compensados desta maneira. Para compensar a inversão de polaridade do sinal `X1` é necessário trocar entre si os sinais `X2` e `X3`. A compensação das inversões de polaridade é efectuada pela função `adjust-all-net-inversions`.

6.4 Resumo

O modelo funcional suportado pelo sistema ReDiFlex pode ser mapeado directamente num modelo físico de estrutura simples. Cada operador é mapeado num componente, e estes são agrupados em fatias, que por sua vez são reunidas em grupos, representados pela classe `scluster`.

Cada componente tem um formato rectangular, com terminais de entrada e saída de lados opostos. Componentes de uma fatia são alimentados por sinais provenientes da fatia imediatamente anterior (à esquerda) e, por sua vez, alimentam componentes da fatia imediatamente seguinte (à direita).

A representação de componentes é feita por três classes, `lblock`, `mlblock` e `alblock`, respectivamente para componentes imutáveis, mutáveis e contentores. Cada componente inclui uma função encarregada de gerar a respectiva configuração interna e, no caso dos dois últimos tipos, outra função para criar configurações posteriores (quer por alteração da configura-

ção existente, que por criação de uma configuração completamente nova). As modificações de configuração nunca podem levar à alteração da geometria do componentes, i.e. das suas dimensões e das posições dos seus terminais. No caso dos componentes contentores, a configuração é composta por duas partes: a configuração do subcomponente e as interligações entre os terminais do subcomponente e os terminais exteriores do contentor. Qualquer uma delas pode ser gerada dinamicamente quando necessário.

Cada componente contém informação sobre os sinais globais a que deve ter acesso, bem como os registos internos que disponibiliza. Esta informação é coligida pelo sistema para criar uma especificação global para todo o grupo de fatias.

O sistema ReDiFlex fornece ferramentas automáticas para efectuar a colocação dos componentes e fazer o encaminhamento das respectivas ligações. A colocação é efectuado por um procedimento heurístico baseado em pesquisa local de uma vizinhança gerada aleatoriamente e tem como objectivo determinar a posição de cada componente no interior da respectiva fatia. O principal objectivo é ter uma implementação com altura inferior ao limite imposto pela arquitectura do circuito FPGA; acessoriamente pretende-se que, no seu todo, o grupo ocupe a menor área possível. Grupos de largura superior à do circuito FPGA, são decompostos em sub-grupos de forma a que cada um não ultrapasse os limites físicos do circuito.

A fase de colocação é seguida pelo encaminhamento das interligações. Esta tarefa processa as interligações entre cada par sucessiva de fatias de maneira independente.Para cada par é efectuada uma pesquisa exaustiva dos percursos que permitem realizar as interligações desejadas. A implementação é recursiva e utiliza *backtracking* para considerar todas as possibilidades, mas evitando prosseguir a pesquisa por alternativas para as quais se possa determinar (baseado nos resultados da pesquisa até essa altura) que não conduzem a uma solução. Durante este fase também é determinada a distância final entre as fatias.

O algoritmo de encaminhamento é de complexidade exponencial, mas como é aplicado a problemas de dimensão relativamente reduzida, esta complexidade não tem repercussões globalmente negativas sobre o desempenho do sistema, pelo menos para os modelos encontrados até ao momento. De qualquer forma, a implementação foi feita com o cuidado de permitir uma substituição do algoritmo de encaminhamento, se necessário. Uma variação deste algoritmo é usada para o encaminhamento no interior de contentores.

Os percursos gerados pelo encaminhamento podem inverter a polaridade dos sinais transmitidos; essa inversão é compensada por alteração da configuração dos blocos lógicos associados aos terminais de entrada.

Todo o processamento descrito neste capítulo é efectuado automaticamente sem qualquer intervenção directa do utilizador para além da invocação da função `make-opnet-impl`. Mas tratando-se de um sistema também destinado à exploração das possibilidades da reconfiguração dinâmica, é igualmente possível aceder à implementação interna dos diversos serviços.

7 Geração de configurações e controlo de *hardware*

O presente capítulo descreve o módulo HWCTRL (fig. 4.4), que é responsável pela inicialização da infra-estrutura reconfigurável, acesso aos seus recursos e mudança de configurações.

A primeira secção descreve a representação interna da informação de reconfiguração, i.e. o modelo que o sistema ReDiFlex mantém dos recursos físicos individuais do circuito FPGA, tanto lógicos como de encaminhamento. Embora o modelo seja capaz de representar directamente todos os recursos do circuito FPGA e a respectiva configuração, como é necessário para descrever o núcleo dos componentes, o sistema ReDiFlex usa uma representação simplificada para as zonas de interligação que facilita a implementação do algoritmo de encaminhamento. O modelo simplificado da zona de encaminhamento é igualmente descrito, bem como a forma de extrair dele os dados para configuração.

A secção 7.2 descreve como a informação interna é transformada em instruções de configuração do circuito FPGA, cobrindo tanto a representação detalhada das configurações dos componentes como a descrição simplificada das zonas de encaminhamento. A secção também descreve a forma como as operações de (re)configuração são organizadas.

A secção 7.3 apresenta os métodos básicos de inicialização da placa H.O.T. Works, de controlo do sinal de relógio e e de acesso ao circuito FPGA para configuração e leitura/escrita de dados. Estes métodos constituem a base para o controlo de *hardware* e são, em última análise, implementados por invocação dos serviços do controlador de dispositivos através de uma fina camada de funções em C++.

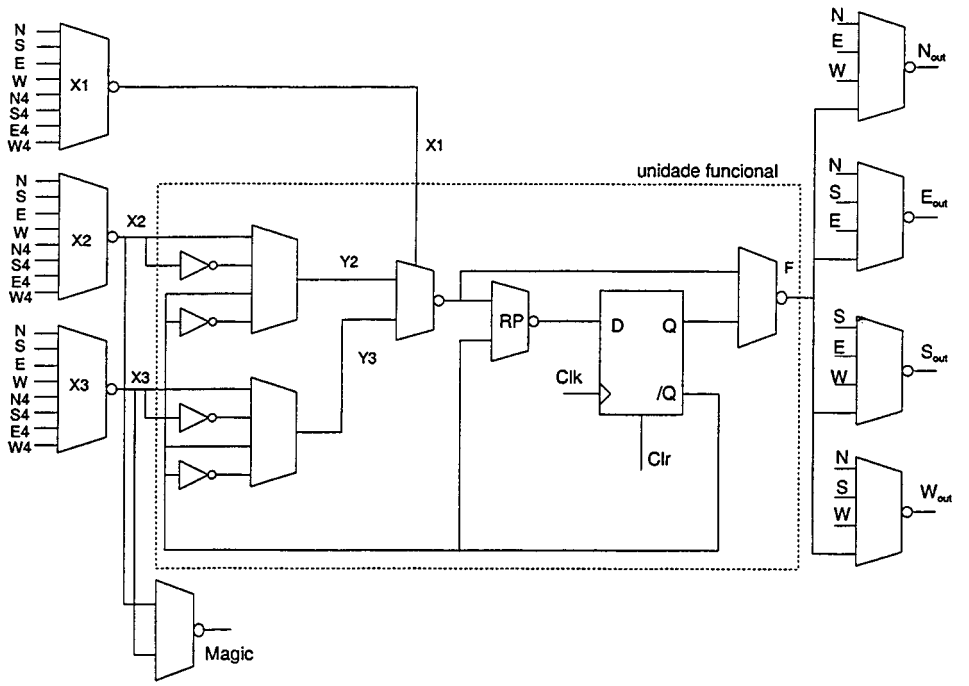


Figura 7.1: Detalhes do bloco lógico da família XC6200. (Cópia da figura 2.9, incluída aqui para facilitar a consulta.)

7.1 Modelo da infra-estrutura reconfigurável

A infra-estrutura reconfigurável é modelada como uma matriz bidimensional de blocos lógicos; a implementação de cada componente (o campo `core` da classe `lblock`) representa um fragmento rectangular dessa matriz.

A representação de um bloco lógico pode ser dividido conceptualmente em duas partes: a unidade funcional e os recursos de encaminhamento de/para os blocos vizinhos (fig. 7.1). No sistema ReDiFlex, a unidade funcional é representada pela classe `functional-cell`:

```
(defclass functional-cell ()
  ((cs :accessor cs :initarg :cs :type (integer 0 1) :initform 0
       :documentation "Combinatorial/sequential mux select.")
   (m :accessor m  :initarg :magic :type (integer 0 1) :initform 0
      :documentation "Source for magic wire.")
   (rp :accessor rp :initarg :rp :type (integer 0 1) :initform 0
       :documentation "Protection selection signal.")
   (y2 :accessor y2 :initarg :y2 :type (integer 0 3) :initform 0
```

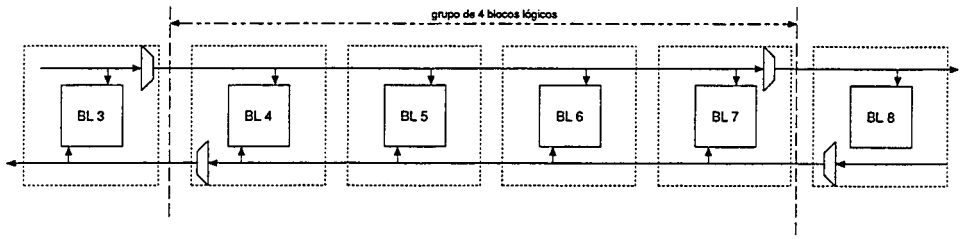


Figura 7.2: Recursos de encaminhamento partilhados por grupos de quatro blocos lógicos.

```
    :documentation "Source for internal mux Y2")
  (y3 :accessor y3 :initarg :y3 :type (integer 0 3) :initform 0
    :documentation "Source for internal mux Y3")))
```

A configuração consiste em definir, para cada multiplexador da unidade funcional qual a entrada a ser seleccionada. Um bloco lógico simples inclui ainda multiplexadores de entrada (que estabelecem os valores dos sinais X1, X2 e X3) e de saída (que definem os valores a disponibilizar aos quatro vizinhos imediatos). A representação do bloco lógico é feita pela classe `cell`:

```
(defclass cell (functional-cell)
 ((n :accessor n :initarg :n :type (integer 0 7) :initform 0
    :documentation "Selection for local north mux")
  (s :accessor s :initarg :s :type (integer 0 7) :initform 0
    :documentation "Selection for local south mux")
  (e :accessor e :initarg :e :type (integer 0 7) :initform 0
    :documentation "Selection for local east mux")
  (w :accessor w :initarg :w :type (integer 0 7) :initform 0
    :documentation "Selection for local west mux")
  (x1 :accessor x1 :initarg :x1 :type (integer 0 7) :initform 0
    :documentation "Source for X1 mux.")
  (x2 :accessor x2 :initarg :x2 :type (integer 0 7) :initform 0
    :documentation "Source for X2 mux.")
  (x3 :accessor x3 :initarg :x3 :type (integer 0 7) :initform 0
    :documentation "Source for X3 mux.")))
```

Cada bloco lógico pode ter ainda mais recursos de encaminhamento associados, cuja natureza exacta depende da posição que o bloco lógico ocupa no circuito FPGA. Por exemplo, todos os blocos lógicos nas colunas {3, 7, 11, ...} alimentam uma pista que abrange os quatro blocos lógicos colocados à sua direita; i.e., associado a um bloco da coluna 3 existe um multiplexador que alimenta uma pista que pode ser acedida pelas células, da mesma linha, que se

situam nas colunas 4–7 (fig 7.2). Por sua vez, os blocos das colunas 4, 8, 12, ... têm associados multiplexadores que podem alimentar quatro blocos à sua esquerda. Como o circuito XC6216 tem uma organização simétrica, ocorre uma situação semelhante para as linhas. Como consequência deste arranjo, um bloco lógico tem uma combinação de recursos que dependem da posição que ocupa no circuito FPGA. A organização básica dos blocos com recursos comuns a grupos de quatro blocos lógicos é representada pelas seguintes classes:

```
(defclass cell-e4 (cell)
  ((e4 :accessor e4 :initarg :e4 :type (integer 0 15) :initform 0
       :documentation "source of e4 mux at 4x4 boundary")
   (alt-east-mux :accessor alt-east-mux :initarg :alt-east-mux
                 :type (integer 0 3) :initform 0
                 :documentation "Source for original (alternative) east mux")))

(defclass cell-n4 (cell)
  ((n4 :accessor n4 :initarg :n4 :type (integer 0 15) :initform 0
       :documentation "source of n4 mux at 4x4 boundary")
   (alt-north-mux :accessor alt-north-mux :initarg :alt-north-mux
                  :type (integer 0 3) :initform 0
                  :documentation "source for original (alternative) north mux")
   (primary-clock :accessor primary-clock :initarg :primary-clock
                  :type (integer 0 3) :initform 0
                  :documentation "Source for primary clock mux")))

(defclass cell-s4 (cell)
  ((s4 :accessor s4 :initarg :s4 :type (integer 0 15) :initform 0
       :documentation "source for s4 mux at 4x4 boundary")
   (alt-south-mux :accessor alt-south-mux :initarg :alt-south-mux
                  :type (integer 0 3) :initform 0
                  :documentation "Source for original (alternative) south mux")))

(defclass cell-w4 (cell)
  ((w4 :accessor w4 :initarg :s4 :type (integer 0 15) :initform 0
       :documentation "source for w4 mux at 4x4 boundary")
   (alt-west-mux :accessor alt-west-mux :initarg :alt-west-mux
                 :type (integer 0 3) :initform 0
                 :documentation "Source for original (alternative) west mux")))
```

A classe `cell-n4` inclui ainda recursos para a definição do sinal de relógio usado no grupo de quatro blocos lógicos verticalmente acima da sua posição (campo `primary-clock`).

Existe ainda a possibilidade de um bloco lógico reunir os recursos inerentes à sua posição numa linha e numa coluna. O sistema usa classes derivadas das anteriores para representar

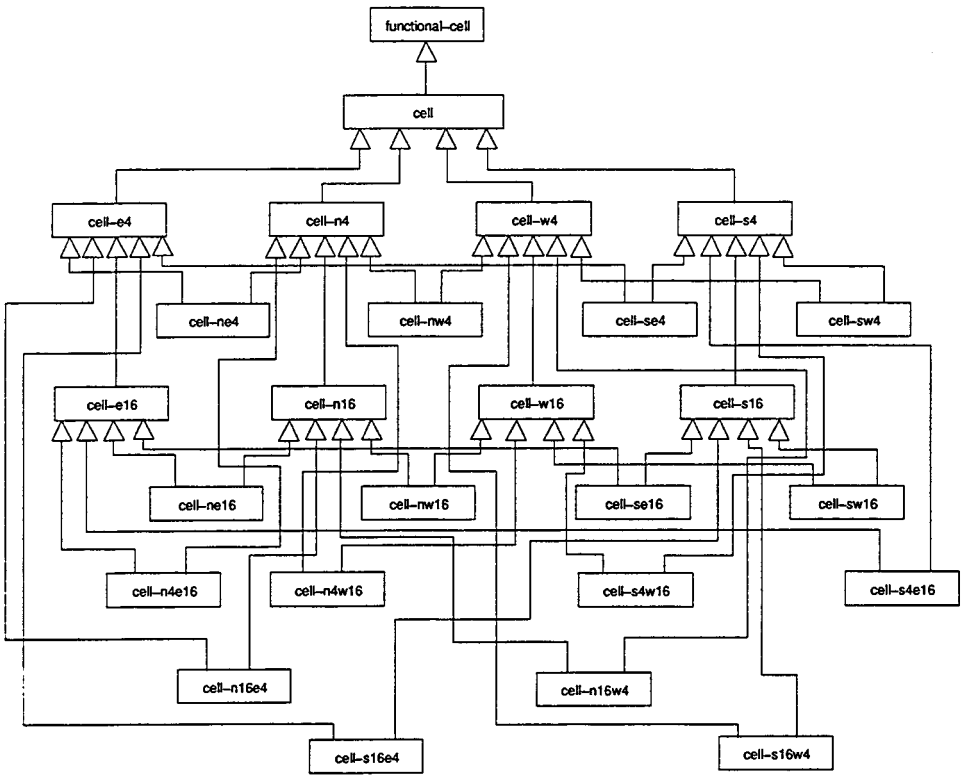


Figura 7.3: Hierarquia de classes que representam os diversos tipos de blocos lógicos.

as quatro combinações legais:

```
(defclass cell-ne4 (cell-n4 cell-e4) ())

(defclass cell-nw4 (cell-n4 cell-w4) ())

(defclass cell-se4 (cell-s4 cell-e4) ())

(defclass cell-sw4 (cell-s4 cell-w4) ())
```

Por exemplo, a classe `cell-ne4` inclui os recursos associados a `cell-n4` e `cell-e4`, sem duplicação (p. ex., inclui apenas uma unidade funcional e não uma por cada classe base).

Cada grupo de 16 blocos lógicos (linhas ou colunas) inclui recursos de encaminhamento adicionais. A sua modelação também é feita por uma hierarquia de classes similar à usada para os blocos lógicos associados aos grupos de quatro. Adicionalmente, existem classes

para representar blocos lógicos que incluem recursos de encaminhamento de nível quatro e dezasseis. A hierarquia completa está ilustrada na figura 7.3. Os campos de todas as classes têm nomes idênticos aos usados na especificação do circuito FPGA [Xilinx, 1997].

Os recursos de encaminhamento associados a cada bloco lógico são unicamente determinados pela sua posição no interior do circuito FPGA. A função que determina a classe apropriada para representar um bloco lógico dada a sua posição no circuito FPGA chama-se `cell-type`. Os resultados produzidos por esta função para os blocos da zona 16 × 16 que inclui a estão indicados na figura 7.4, onde se pode observar a organização regular do circuito FPGA. As restantes zonas de 16 × 16 blocos têm a mesma organização[1].

Todos os campos das classes derivadas de `functional-cell` assumem valores inteiros, que representam directamente os valores dos respectivos bits de configuração. Para o programador é mais fácil trabalhar com valores simbólicos que correspondam aos nomes usados na especificação do circuito FPGA. O mapeamento de nomes simbólicos em códigos numéricos é feito pela função `mux-index`. Esta função aceita como parâmetros o nome de uma classe, o nome de um multiplexador e a especificação simbólica do valor de saída do multiplexador, produzindo o respectivo código numérico de configuração. Por exemplo, para indicar que a saída e para o bloco lógico situado à esquerda de outro é o resultado (negado) produzido pela unidade funcional usa-se a invocação

```
(setf (w celula) (mux-index 'cell 'e '(not f)))
```

em que `celula` é um objecto da classe `cell` ou alguma das suas classes derivadas. Outro exemplo da utilização desta função é no ajustamento das configurações dos blocos lógicos de entrada para compensar as inversões introduzidas pelo encaminhamento (sec 6.3.2).

```
1  (defun invert-terminal (cell terminal)
2    "Modify the F generator to accept an inverted signal on the terminal."
3    (ecase terminal
4      (x1 (psetf (y2 cell) (svref *y3->y2-map* (y3 cell))
5                 (y3 cell) (svref *y2->y3-map* (y2 cell)))
6          (switch-x2-x3 cell))
7      (x2 (cond ((= (y2 cell) (mux-index 'functional-cell 'y2 '(not x2)))
8                 (setf (y2 cell) (mux-index 'functional-cell 'y2 'x2)))
9                ((= (y2 cell) (mux-index 'functional-cell 'y2 'x2))
10                (setf (y2 cell) (mux-index 'functional-cell 'y2 '(not x2))))))
11     (x3 (cond ((= (y3 cell) (mux-index 'functional-cell 'y3 'x3))
12                (setf (y3 cell) (mux-index 'functional-cell 'y3 '(not x3))))
13               ((= (y3 cell) (mux-index 'functional-cell 'y3 '(not x3)))
14                (setf (y3 cell) (mux-index 'functional-cell 'y3 'x3)))))))
```

[1] Todos os modelos da família XC6200 têm um número inteiro de zonas de 16 × 16 blocos lógicos.

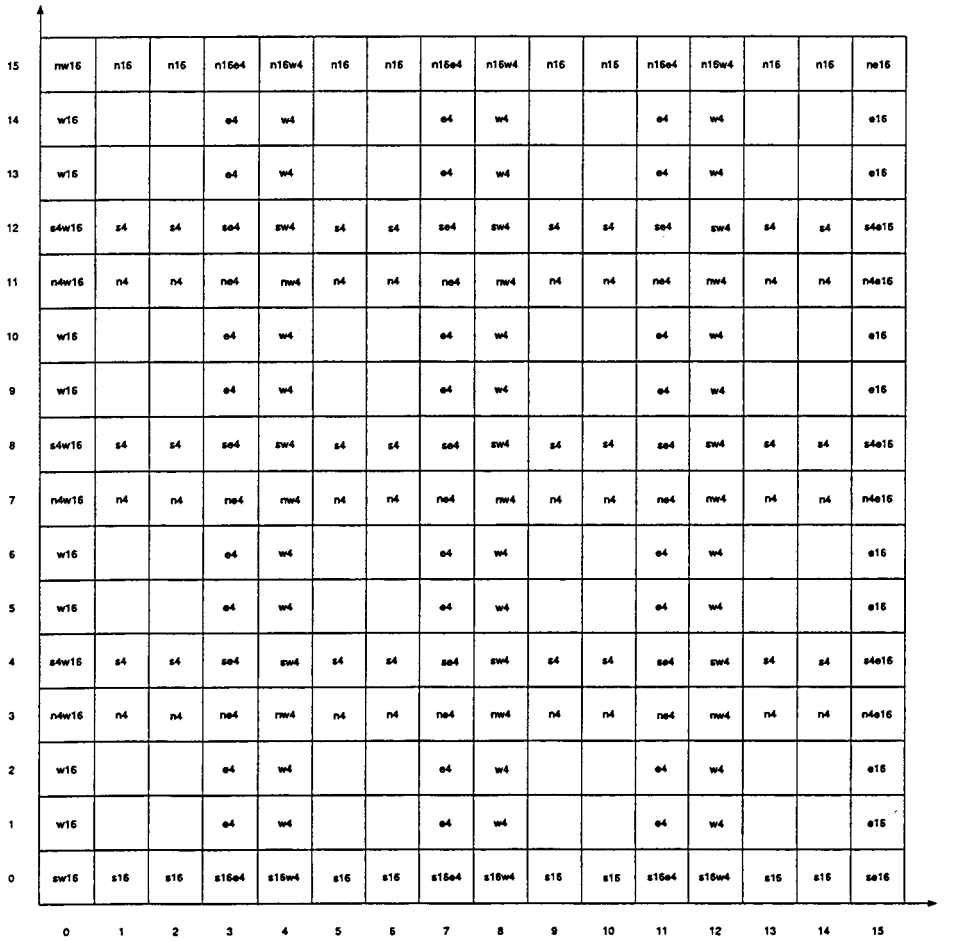


Figura 7.4: Classes usadas para representar os blocos lógicos segundo a posição destes. As células vazias são representadas por instâncias da classe `cell`. Todas as outras são representadas por instâncias da classe cujo nome é obtido juntando o prefixo `"cell-"` ao nome indicado na respectiva posição.

Dependendo do tipo de terminal (linha 3), esta função implementa as modificações necessárias para compensar a inversão das entradas; o caso da entrada X1 é ligeiramente mais complicado, pelo que a modificação efectiva (após determinação do mapeamento de valores segundo tabelas pré-definida, *y3->y2-map e *y2->y3-map*) é deixado para uma função auxiliar switch-x2-x3, que implementa as modificações necessárias através de uma abordagem muito similar à usada para X2 e X3. Mais exemplos da utilização da função mux-index aparecem na secção 6.2.2 sobre geradores de componentes.

Para a descrição dos núcleos de componentes produzidos pelos geradores é necessário modelar exactamente todos os recursos existentes. Porém o algoritmo de encaminhamento aproveita apenas um subconjunto dos recursos disponíveis, pelo que o sistema ReDiFlex usa uma representação interna mais simples para representar os recursos usados nas zonas de encaminhamento situadas entre as fatias, definida pela classe routing-cell:

```
(defclass routing-cell ()
  ((dont-use :accessor dont-use :initarg :dont-use
             :initform nil :type boolean
             :documentation "Flag indicating if cell can be used.")
   (in-e :accessor in-e :initarg :in-e :type snet
         :documentation "The net that comes in from the left.")
   (in-w :accessor in-w :initarg :in-w :type snet
         :documentation "The net that comes in from the right.")
   (in-n :accessor in-n :initarg :in-n :type snet
         :documentation "The net that comes in from the bottom.")
   (in-s :accessor in-s :initarg :in-s :type snet
         :documentation "The net that comes in from the top.")
   (out-e :accessor out-e :initarg :out-e :type snet
         :documentation "The net that goes out to the right.")
   (out-w :accessor out-w :initarg :out-w :type snet
         :documentation "The net that goes out to the left.")
   (out-n :accessor out-n :initarg :out-n :type snet
         :documentation "The net that goes out to the top.")
   (out-s :accessor out-s :initarg :out-s :type snet
         :documentation "The net that goes out to the bottom.")))
```

A classe routing-cell representa os terminais exteriores de cada bloco lógico, conforme ilustrados na figura 6.6. A cada terminal é associada a interligação que o usa (um terminal não usado não tem nenhuma interligação atribuída). A zona de encaminhamento é representada por uma matriz bidimensional de routing-cell. Como a zona não é necessariamente rectangular (fig. 6.1), alguns blocos lógicos não podem ser usados; o campo dont-use é usado para marcar as posições correspondentes.

7.2 Geração de configurações

Para configurar a infra-estrutura de *hardware* é necessário converter a descrição do núcleo dos componentes e das zonas de interligação em valores a enviar para o circuito FPGA. Existem basicamente duas situações em que a geração desses valores é feita: (i) quando um grupo de componentes é descarregado pela primeira vez para o circuito FPGA; (ii) quando é preciso reconfigurar um ou mais componentes já presentes no circuito FPGA. No primeiro caso geram-se os valores de configuração para cada componente e para as zonas de encaminhamento entre fatias; no segundo caso é apenas necessário reconfigurar a zona do circuito FPGA onde o componente está situado (devido às restrições impostas pelo modelo físico, o tamanho da zona atribuída a cada componente é fixa). A primeira operação é feita pela função `download-cluster`:

```
(defun download-cluster (clu &key (download t) (stream nil) (completely t))
  "Downloads a placed cluster to the X6200 RPU."
  (let ((clk-src 'gclk))
    (flet ((do-block (sb)
             (do ((i 0 (1+ i)))
                 ((= i (width sb)))
               (do ((j 0 (1+ j)))
                   ((= j (height sb)))
                 (let* ((loc-x (+ i (place-x sb)))
                        (loc-y (+ j (place-y sb))))
                   (output-configuration loc-x loc-y
                                         (encode-cell (aref (core sb) i j) completely)
                                         download stream)))))
           (do-routing (rr)
             (declare (type routing-region rr))
             (do ((i 0 (1+ i)))
                 ((= i (width rr)))
               (do ((j 0 (1+ j)))
                   ((= j (height rr)))
                 (let* ((loc-x (+ i (place-x rr)))
                        (loc-y (+ j (place-y rr))))
                   (output-configuration loc-x loc-y
                                         (encode-routing-cell (aref (cells rr) i j)
                                                              loc-x loc-y completely)
                                         download stream)))))
           (do-clock (cellpos)
             (output-clk-configuration cellpos download stream clk-src))
           (do-clear (cellpos)
             (output-clr-configuration cellpos download stream)))
```

```
30
31      (when (placed clu)
32        (loop for sli across (slices clu)
33              do   (map nil #'do-block (components sli))
34                   (setf clk-src (clock-source sli))
35                   (map nil #'do-clock (clock-cells sli))
36                   (map nil #'do-clear (clear-cells sli)))
37        (loop for rr across (sliceroutes clu)
38              if (not (routing-region-empty-p rr))
39              do (do-routing rr))))))
```

A função aceita como argumentos o grupo cuja configuração deve ser emitida e mais três valores que especificam detalhes do processamento. O argumento booleano `download` especifica se os valores emitidos devem ser enviados para o circuito FPGA. O argumento `stream`, se não for nulo, indica um ficheiro onde guardar os dados de configuração. Por fim, o argumento booleano `completely` indica se deve ser emitida uma configuração completa dos blocos lógicos ou apenas os valores que são diferentes do estado de um bloco logo após inicialização do circuito FPGA.

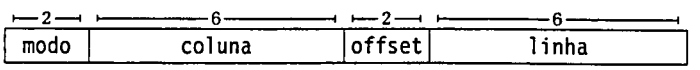
Esta função usa quatro funções auxiliares internas. A função `do-block` (linha 4) gera os valores de configuração do núcleo de um componente, usando a função `output-configuration` (linha 11); a função `d-routing` (linha 14) faz o mesmo para uma zona de encaminhamento. As funções `do-clock` (linha 26) e `do-clear` (linha 28) emitem os valores de configuração dos sinais globais de relógio e inicialização, respectivamente. O corpo da função (linha 31 e seguintes) emite a configuração de todas as fatias (linha 32), seguidos da configuração das zonas de encaminhamento (linha 37 e seguintes).

Os dados de configuração são guardados em ficheiro no mesmo formato que as configurações geradas pela ferramenta Xact6000 da companhia Xilinx, pelo que podem ser usadas com algumas ferramentas auxiliares que acompanham essa ferramenta. Esses ficheiros apenas contêm informação sobre uma dada configuração, não sendo possível incluir neles qualquer tipo de informação sobre reconfiguração dinâmica.

O sistema ReDiFlex implementa ainda a função `download-block` para emitir a configuração de apenas um componente. Esta função faz basicamente o mesmo que função auxiliar `do-block` indicada atrás, embora com uma verificação mais cuidada dos argumentos.

7.2.1 Configuração de componentes

Para efectuar a configuração do circuito FPGA é necessário construir o endereço do bloco a configurar e a sequência de valores a colocar aí. Os endereços do circuito XC6216 têm o seguinte formato:



Linha (6 bits) e coluna (6 bits) representam as coordenadas do bloco lógico, em que o bloco do canto inferior direito tem as coordenadas (0,0).

O campomodo (2 bits) selecciona o tipo de recurso endereçado, de acordo com a seguinte tabela:

modo[1]	modo[0]	Recurso
0	0	Configuração ou estado do bloco lógico
0	1	Multiplexadores esquerda/direita ou bloco E/S
1	0	Multiplexadores superiores/inferiores ou bloco E/S
1	1	Registos de controlo do circuito FPGA

O campo offset (2 bits) associa até 4 bytes de informação com cada tipo de recurso. por exemplo, um bloco lógico simples precisa de 3 bytes de configuração, distinguidos pelo valor do campo offset. A construção de um endereço é feita pela função make-cell-address:

```
(defun make-cell-address (x y mode offset)
(let ((addr y))
    (setf addr (dpb offset *offset-byte* addr)
          addr (dpb x *column-byte* addr)
          addr (dpb mode *mode-byte* addr))
    addr))
```

Para construir os dados de configuração de um bloco lógico, a rotina download-cluster emprega a função genérica encode-cell, cujos métodos estão especializados para as diversas classes da hierarquia ilustrada na figura 7.3. A construção desses valores consiste em juntar os diversos grupos de bits em bytes de acordo com as regras especificadas em [Xilinx, 1997]. Para o caso de uma célula simples, a informação de configuração tem o seguinte formato:

Offset	Dados							
	7	6	5	4	3	2	1	0
00	Mux N		Mux E		Mux W		Mux S	
01	CS	X1			X2[1:0]		X3[1:0]	
10	M	RP	Y2		Y3		X3[2]	X2[2]

A versão base de encode-cell produz os valores numéricos necessários para codificar a informação presente num objecto da classe cell.

```
1  (defmethod encode-cell ((c cell) completely)
2    "Generate encoding of a cell for downloading to RPU."
3    (let ((lst nil))
4      ;; byte 0
5      (when (or completely (not (= 0 (n c) (s c) (e c) (w c))))
6        (let ((v 0))
```

```
7         (setf v (dpb (rem (n c) 4) *cell-n-fld* v) ;rem is needed!!
8               v (dpb (rem (s c) 4) *cell-s-fld* v)
9               v (dpb (rem (e c) 4) *cell-e-fld* v)
10              v (dpb (rem (w c) 4) *cell-w-fld* v))
11        (push (list 0 0 v) lst)))
12    ;; byte 1
13    (when (or completely (not (= 0 (cs c) (x1 c) (x2 c) (x3 c))))
14      (let ((v 0))
15        (setf v (dpb (cs c) *cell-cs-fld* v)
16              v (dpb (x1 c) *cell-x1-fld* v)
17              v (dpb (rem (x2 c) 4) *cell-x2ls-fld* v)
18              v (dpb (rem (x3 c) 4) *cell-x3ls-fld* v))
19        (push (list 0 1 v) lst)))
20    ;; byte 3
21    (when (or completely (not (= 0 (m c) (rp c) (x2 c) (x3 c) (y2 c) (y3 c))))
22      (let ((v 0))
23        (setf v (dpb (m c) *cell-m-fld* v)
24              v (dpb (rp c) *cell-rp-fld* v)
25              v (dpb (floor (x2 c) 4) *cell-x2ms-fld* v)
26              v (dpb (floor (x3 c) 4) *cell-x3ms-fld* v)
27              v (dpb (y2 c) *cell-y2-fld* v)
28              v (dpb (y3 c) *cell-y3-fld* v))
29        (push (list 0 2 v) lst)))
30    lst))
```

A configuração de uma célula necessita de três bytes, que são construídos pelos blocos de código iniciados nas linhas 6, 14 e 22. A função retorna uma lista em que cada elemento tem três números: modo, "offset" e valor de configuração).

A geração dos valores de configuração para os outros tipo de de blocos lógicos é feita pelas versões apropriadas de encode-cell. Por, exemplo para os blocos cell-e4:

```
1   (defmethod encode-cell ((c cell-e4) completely)
2     "Generate encoding of a cell-e4 for downloading to RPU"
3     (let ((lst (call-next-method)))
4       ;; only one byte
5       (unless (typep c 'cell-e16)
6         (when (or completely (not (= 0 (e c) (e4 c))))
7           (let ((v 0))
8             (setf v (dpb (floor (e c) 4) *cell4-e-fld* v)
9                   v (dpb (e4 c) *cell4-e4-fld* v))
10            (push (list 1 0 v) lst))))
11    lst))
```

A codificação da configuração para um bloco lógico do tipo `cell-e4` começa pela chamada ao método da função base (`cell`) efectuada na linha 3. Ao resultado dessa chamada é acrescentado mais um byte, completando assim a informação necessária para configurar um bloco do tipo `cell-e4`. O sistema dispõe de especializações similares para os blocos do tipo `cell-n4`, `cell-w4` e `cell-s4`, bem como para `cell-e16`, `cell-n16`, `cell-w16` e `cell-s16`. Estas especializações cobrem todos os casos, não sendo necessário especificar especializações para os tipos restantes `cell-n4e16`, já que estes são automaticamente processados pela combinação de métodos apropriada, conforme definido pela semântica de Common Lisp.

A função `output-configuration` (usada em `download-cluster`, pág. 221) é chamada com a informação produzida pela função genérica `encode-cell` e com a posição do bloco lógico na grelha do circuito FPGA para efectuar a configuração ou gerar a representação ASCII a guardar no ficheiro:

```
1  (defun output-configuration (x y conf-data download stream)
2    "Outputs correctly formatted configuration data to RPU and STREAM"
3    (flet ((output-one (conf-item)
4             "Output one configuration item."
5             (let ((addr (make-cell-address x y (first conf-item) (second conf-item))))
6               (when stream
7                 (format stream "(#x~4,'0X #x~2,'0X)~%" addr (third conf-item)))
8               (when download
9                 (unless (drh-raw:write-6200 addr (third conf-item))
10                  (error "DRH-RAW:WRITE-6200 failed"))))))
11     (mapc #'output-one conf-data)))
12
```

Para cada código de configuração retornada por `encode-cell`, a função `output-configuration` constrói o endereço correcto (invocando a função `make-address` na linha 5). Se especificado, a informação é escrita num ficheiro (linha 7), no co-processador (linha 9) ou em ambos.

Os circuitos produzidos pelo sistema ReDiFlex não usam os pinos do circuito FPGA, pelo que a respectiva configuração é deixada no estado inicial, i.e. desactivada. Contudo, os blocos de controlo associados a cada pino são também fonte de sinais globais para as células adjacentes. A situação está ilustrada na figura 7.5 para o sinal de relógio. Como se pode ver, cada grupo de quatro blocos consecutivos da mesma coluna recebe um sinal de relógio cuja fonte é seleccionada no bloco superior do grupo anterior. A excepção é o grupo mais próximo da borda sul do circuito FPGA, que é alimentado a partir de um bloco de controlo de I/O. A função encarregada de configurar estes blocos é `output-clock-configuration`, que usa a informação do campo `clock-cells` existente em cada fatia.

```
1  (defun output-clk-configuration (cellpos download stream clk-src)
2    "Output the configuration data for bottom IOB on column xpos."
```

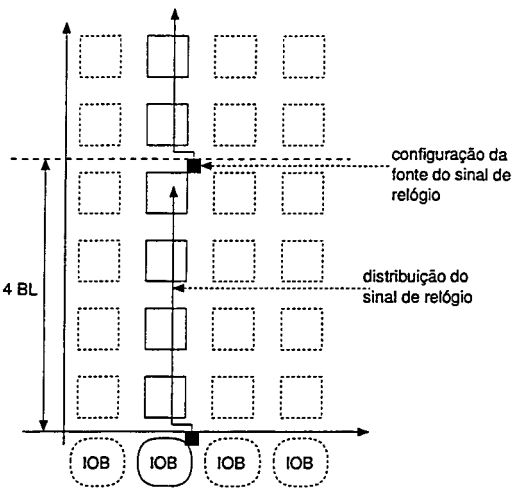


Figura 7.5: Distribuição do sinal de relógio.

```
3     (let ((xpos (car cellpos))
4           (ypos (cdr cellpos)))
5       (if (minusp ypos)   ;; row (y) = -1 means iob
6           (output-clk-iob xpos download stream clk-src)
7           (output-clk-switch xpos ypos download stream clk-src))))
```

A função `clock-output-configuration` determina a posição do bloco lógico ou bloco de E/S a configurar e invoca a função apropriada a cada uma das situações. Esta função aceita um parâmetro que especifica qual o sinal de relógio a usar (`G1` ou `GCLK`).

```
1  (defun output-clk-iob (xpos download stream clk-src)
2    "Activate clock in the bottom IOB"
3    (let ((addr1 #b1000000000000001)
4          (addr2 #b1000000001000001)
5          (byte1 (if (eq clk-src 'gclk) #b00100000 #b00110000))
6          (byte2 (if (eq clk-src 'gclk) #b00010000 #b00000000)))
7      (setf addr1 (dpb xpos *column-byte* addr1)
8            addr2 (dpb xpos *column-byte* addr2))
9      (when stream
10       (format stream "(#x~4,'0X #x~2,'0X)~%(#x~4,'0X #x~2,'0X)~%"
11               addr1 byte1 addr2 byte2))
12     (when download
13       (drh-raw:write-6200 addr1 byte1)
14       (drh-raw:write-6200 addr2 byte2))))
```

A configuração do sinal de relógio no bloco de E/S necessita de dois bytes, que são construídos de acordo com a codificação descrita em [Xilinx, 1997]. O resultado final é usado para programar o co-processador (linha 13 e seguinte).

A programação do sinal de relógio no interior da matriz de blocos é feito pela função `output-clk-switch`:

```
1  (defun output-clk-switch (xpos ypos download stream clk-src)
2    "Activates clock in the north switch at XPOS, YPOS."
3    (let ((addr0 ypos)
4          (byte0 (if (eq clk-src 'gclk) #b00000110 #b00000000)))
5      (setf addr0 (dpb xpos *column-byte* addr0)
6            addr0 (dpb 2 *mode-byte* addr0))
7      (when stream
8        (format stream "(#x~4,'0X #x~2,'0X)~%" addr0 byte0))
9      (when download
10       (drh-raw:write-6200 addr0 byte0))))
```

A distribuição do sinal de inicialização é feita de maneira semelhante, excepto que cada sinal é partilhado por grupos de 16 blocos lógicos e a distribuição do sinal se faz de cima para baixo, o que obriga a configurar os blocos de E/S situados na borda superior do circuito FPGA.

Como já foi referido anteriormente, a escrita de valores em registos de utilizadores existentes no interior de componentes requer a modificação da configuração de relógio destes. Essas operações são implementadas pelas funções `change-clk-to-gclk` e `change-clk-to-gl`, que alteram directamente a configuração existente no co-processador (e não alteram nenhum campo do modelo físico interno).

A escrita num registo só é asseguradamente correcta se o *flip-flop* de cada bloco lógico tiver a sua entrada isolada do restante circuito. Isso é obtido pela configuração do multiplexador `RP` (fig. 6.7) de maneira que a saída do *flip-flop* seja retro-alimentada para a sua entrada. As funções `protect-register-cells` e `unprotect-register-cells` estão encarregadas de efectuar essa modificações. Como as funções anteriores, também estas operam directamente sobre a configuração presente no co-processador e não sobre o modelo físico interno.

7.2.2 Configuração de zonas de encaminhamento

A configuração das zonas de encaminhamento pode ser efectuada segunda a abordagem usada em `encode-cell`, excepto que apenas existe um tipo de objectos (`routing-cell`) a ser codificado. Para aproveitar o código que processa os objectos da classe `cell` e classes derivadas, a função `encode-routing-cell` cria um objecto para representar o bloco lógico por onde passa o percurso de encaminhamento, configura-o para corresponder à informação correspondente em `routing-cell` e invoca a função `encode-cell` sobre esse objecto temporário.

```
(defun encode-routing-cell (rc x y completely)
  "Generates encoding of routing cell."
  (flet ((translate-input-spec (rc-input)
           (case rc-input
             ((in-e) '(not e)) ((in-n) '(not n))
             ((in-s) '(not s)) ((in-w) '(not w)))))
    (let ((cell (make-instance (cell-type x y))))
      (flet ((adjust-one-output (outspec mux)
               "Adjust one output multiplexer of core cell."
               (when (not (routing-cell-free-output-p rc outspec))
                 (let* ((net (routing-cell-terminal-net rc outspec))
                        (inp (routing-cell-net-inputs rc net)))
                   (mux-index 'cell mux (translate-input-spec (car inp))))))
             (adjust-west-edge ()
               (when (not (routing-cell-free-output-p rc 'out-w))
                 (let* ((net (routing-cell-terminal-net rc 'out-w))
                        (inp (routing-cell-net-inputs rc net)))
                   (mux-index 'cell-w4 'alt-west-mux (translate-input-spec (car inp))))))
             (adjust-east-edge ()
               ; omitido: similar a adjust-west-edge
               )
             (adjust-south-edge ()
               ; omitido: similar a adjust-west-edge
               )
             (adjust-north-edge ()
              ; omitido: similar a adjust-west-edge
               ))

        (let ((r (adjust-one-output 'out-e 'e))) (when r (setf (e cell) r)))
        (let ((r (adjust-one-output 'out-w 'w))) (when r (setf (w cell) r)))
        (let ((r (adjust-one-output 'out-n 'n))) (when r (setf (n cell) r)))
        (let ((r (adjust-one-output 'out-s 's))) (when r (setf (s cell) r)))
        (when (= 0 (rem x 4))
          (let ((r (adjust-west-edge))) (when r (setf (alt-west-mux cell) r))))
        (when (= 3 (rem x 4))
          (let ((r (adjust-east-edge))) (when r (setf (alt-east-mux cell) r))))
        (when (= 0 (rem y 4))
          (let ((r (adjust-south-edge))) (when r (setf (alt-south-mux cell) r))))
        (when (= 3 (rem y 4))
          (let ((r (adjust-north-edge))) (when r (setf (alt-north-mux cell) r))))
        (encode-cell cell completely)))))
```

Esta função usa várias funções auxiliares locais para apoiar a conversão entre `routing-cell` e os multiplexadores de cada bloco lógico (`translate-input-spec` na linha 3), para configurar os multiplexadores de saída (`adjust-one-output` na linha 8), bem como várias funções para ajustar a configuração de células na borda de uma zona 4×4. O corpo da função (com início na linha 29) configura os multiplexadores de saída. Se a célula pertencer à borda de uma zona 4×4 é preciso ajustar mais um valor interno (correspondente à configuração de multiplexador alternativo[2]). Esse ajuste é efectuado na linha 33 e seguintes. Finalmente na última linha (41) é chamada a função `encode-cell` para produzir a codificação desejada.

7.3 Operações básicas de controlo de *hardware*

Esta secção sumaria as operações básicas de controlo de *hardware* que o sistema ReDiFlex usa. Embora o sistema ReDiFlex tenha uma interface completa para todas as funções externas, apenas uma pequena parte é activamente usada. As funções externas são acedidas pelas funções de Lisp definidas no módulo `DRH-RAW`.

Acesso de leitura/escrita

O acesso ao circuito FPGA é feito pelas funções `drh-raw:write-6200` e `drh-raw:read-6200`, para escrita e leitura, respectivamente. A primeira função aceita como argumentos o endereço e um número (até 32 bits) que representa a informação a guardar. O tipo de informação deve estar de acordo com o endereço, já que este é usado para distinguir entre informação de configuração e informação do estado de *flip-flops*. A segunda função aceita um endereço e retorna os dados associados (que, dependendo do endereço, podem ser dados de configuração ou os valores de *flip-flops*). A interface FastMap permite configurar o número de bits (8, 16 ou 32) a usar em cada transferência. O sistema usa as funções `drh-raw:set-bus-width` e `drh-raw:get-bus-width` para controlar esse aspecto da interface.

O acesso aos registos do utilizador requer o estabelecimento da máscara de selecção de blocos lógicos (cf. fig 6.2). Uma nova máscara é criada pela função `drh:new-map-register` e estabelecida por `drh-raw:set-rpu-map-register`. As funções `drh-raw:write-usr-reg` e `drh-raw:-read-usr-reg` implementam o acesso aos registos dada a coluna em que estes se situam (o que envolve simplesmente formar o endereço apropriado e invocar as subrotinas de acesso ao circuito FPGA).

Com base nestas funções, o sistema ReDiFlex implementa uma abstracção de nível superior para associar toda a informação relacionada com um registo do utilizador a um identificador. As estruturas `user-register` e `design-setup` são os principais componentes dessa implementação:

[2] A necessidade deste ajuste deve-se a uma irregularidade das células localizadas nesta posição: um multiplexador de saída passa de 4:1 a 8:1, mas permanece a possibilidade de aceder ao que seria a saída do multiplexador original 4:1. Esta característica não é usada no sistema ReDiFlex, mas como a configuração dos dois multiplexadores não é independente, é preciso garantir que as respectivas especificações são coerentes.

```
(defstruct user-register
  "Representation of a user defined register on the RPU board."
  (map-reg (new-map-register) :type map-register)
  (column 0 :type (unsigned-byte 32))
  (mask 0 :type (unsigned-byte 32))
  (type 'read-write :type (member read-write read-only)))

(defstruct design-setup
  "Software view of a design setup on the RPU."
  (regs (make-hash-table :size 8 :test #'equal)
        :type hash-table))
```

Cada registo do utilizador tem uma máscara de selecção (`map-reg`), o endereço da coluna onde está situado (`column`) e o tipo de acesso suportado (`type`). Em geral, o número de bits de um registo não é igual à dimensão lógica do barramento de dados (que pode ser de 8, 16 ou 32 bits); consequentemente é preciso especificar que bits do barramento de dados são realmente significativos, o que é feito pelo conteúdo do campo `mask`.

A estrutura `design-setup` é uma tabela de dispersão (`regs`) que mapeia cadeias de caracteres (os nomes dos registos) em instâncias da estrutura `user-register`.

A utilização de registos do utilizador a este nível mais abstracto é feita através das rotinas `write-user-register` e `read-user-register`:

```
1 (defun write-user-register (setup regname value)
2   "Write a value to a user register on the RPU."
3   (let ((ur (gethash regname (design-setup-regs setup))))
4     (unless ur
5       (error "No register ~A in this setup.~%" regname))
6     (unless (set-rpu-map-register (user-register-map-reg ur))
7       (error "Unable to set map register to access user register ~A.~%" regname))
8     (unless (write-user-reg (user-register-column ur) value)
9       (error "Unable to write to user register ~A.~%" regname))))
```

A função começa por obter o objecto `user-register` correspondente ao nome (linha 3), para depois estabelecer a máscara de mapeamento (linha 6) e, por fim, escrever os dados no registo (linha 8).

```
1 (defun read-user-register (setup regname)
2   "Read a value from a user register on the RPU."
3   (let ((ur (gethash regname (design-setup-regs setup))))
4     (unless ur
5       (error "No register ~A in this setup.~%" regname))
6     (unless (set-rpu-map-register (user-register-map-reg ur))
7       (error "Unable to set map register to access user register ~A.~%" regname))
```

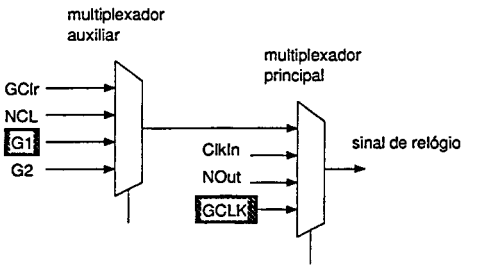


Figura 7.6: Circuito de selecção de relógio.

```
8       (let ((res (read-user-reg (user-register-column ur))))
9         (logand (user-register-mask ur) res))))
```

As operações da função são similares às da função `write-user-register`, excepto que é feita uma leitura em vez de uma escrita (linha 6) e que o valor lido deve ser combinado com a máscara para obter o valor correcto (linha 8).

Sinais de relógio

A placa H.O.T. Works pode fornecer o sinal de relógio ao circuito FPGA através de dois pinos diferentes, `GCLK` ou `G1`. O primeiro constitui o sinal de relógio principal, com que todas as operações do interface FastMap devem estar sincronizadas. Habitualmente, este sinal também é usado para controlar a operação do circuito configurado. Pelas razões indicadas na secção 6.1, os circuitos criados pelo sistema ReDiFlex usam o sinal `G1` como sinal de relógio. O controlo de relógio é feito através das seguintes funções:

`clock-on` Activa a geração do sinal de relógio (e configura a placa para alimentar o pino `GCLK` com esse sinal).

`clock-off` Inibe o sinal de relógio, mas mantém a placa configurada para usar o pino `GCLK`.

`g1-clock-on` Activa a geração do sinal de relógio e configura a placa para alimentar o pino `G1` com esse sinal.

`g1-clock-off` Inibe o sinal de relógio, mas mantém a placa configurada para usar o pino `G1`.

`step-clock` Activa o sinal de relógio por um número de ciclos especificado como argumento. O pino usado é o que estiver configurado por utilização das funções anteriores.

A utilização do sinal `G1` tem ainda uma complicação não mencionada anteriormente, porque está relacionada com a forma como o sinal de relógio é seleccionado no interior do circuito FPGA. A figura 7.6 apresenta o circuito de selecção do sinal de relógio. Para seleccionar o sinal

ligado ao pino GCLK, basta configurar o multiplexador principal. Para usar o sinal do pino G1 é necessário configurar os dois multiplexadores, o que é mais complicado pela seguinte razão: os dados de configuração do multiplexador auxiliar não estão associados ao endereço do bloco lógico onde está situado o multiplexador auxiliar. A configuração do multiplexador auxiliar das células situadas nas colunas 1, 5, 9, etc. é acedida através dos endereços das células vizinhas situadas nas colunas 0, 4, 8, etc. Por sua vez, a configuração das células das colunas 2, 6, 10, etc. é acedida através dos endereços correspondentes às células vizinhas nas colunas 3, 7, 11, etc.

Esta situação é a única excepção ao princípio organizacional de que cada célula contém todos os dados necessários para a sua própria configuração. O modelo da infra-estrutura apresentado na secção 7.1 segue este princípio (que simplifica grandemente a organização), e consequentemente, não considera o multiplexador auxiliar do sinal de relógio. A solução adoptada consiste em configurar previamente (após a inicialização do circuito FPGA) todas os multiplexadores auxiliares para seleccionarem o sinal G1; posteriormente, a escolha entre G1 e GCLK envolve apenas a reconfiguração do multiplexador principal, cuja configuração é efectivamente local. Uma outra alternativa possível consistiria em alterar o modelo da infra-estrutura para representar mais fielmente a implementação da infra-estrutura física; contudo isso quebraria a assunção de que a configuração de um componente é independente da configuração dos blocos lógicos situados na sua vizinhança e complicaria substancialmente o processo de geração de configurações. A complicação conceptual resultante não seria compensada pela utilidade prática dessa modificação.

A configuração dos multiplexadores auxiliares de relógio é efectuada pela rotina `set-default-secondary-clock`:

```
1  (defun set-default-secondary-clock ()
2    "Programs the RPU so that all secondary clocks output the G1 signal."
3    (let ((byte #b00001010))
4      (dotimes (x *die-width*)
5        (dotimes (y *die-height*)
6          (when (and (= 3 (rem y 4))
7                     (or (= 0 (rem x 4))
8                         (= 3 (rem x 4))))
9            (let ((addr (make-cell-address x y 2 3)))
10             (drh-raw:write-6200 addr byte)))))))
```

Esta rotina essencialmente percorre toda a matriz de blocos lógicos, configurando todos os multiplexadores secundários com o valor binário 00001010, que corresponde a seleccionar o sinal G1 para a saída do multiplexador auxiliar.

Inicialização do sistema

Antes de ser utilizada, a placa H.O.T. Works deve ser inicializada; também é preciso inicializar o circuito FPGA. O sistema ReDiFlex inclui para isso as seguintes funções:

`init-board` Executa a inicialização da placa H.O.T. Works. Esta função deve ser executada no início de qualquer sessão de trabalho.

`board-ok-p` Determina se a placa foi inicializada com sucesso. A maneira de saber se o computador contém uma placa H.O.T. Works identificada pelo sistema operativo consiste em invocar `init-board`, seguida de `board-ok-p`.

`reset-board` Re-inicializa o circuito FPGA existente na placa, eliminando qualquer configuração existente.

`clear-board` O sinal global de inicialização (GCLR) é enviado ao circuito FPGA, colocando a zero o estado de todos os elementos de memória do circuito configurado. A configuração do circuito FPGA não é alterada.

`set-rpu-id` Inicializa o registo de identificação do circuito FPGA com o valor apropriado (que depende do modelo). Apenas quando este registo contém o valor correcto é que a configuração é activada e o acesso para leitura/escrita de registos do utilizador possível.

Outras funções

O sistema ReDiFlex inclui ainda outras funções de baixo nível, p. ex. para controlar o monitorizador de corrente consumida pelo circuito FPGA ou para variar a operação do gerador de sinal de relógio existente na placa.

7.4 Resumo

Este capítulo descreve os serviços de baixo nível que controlam directamente a infra-estrutura reconfigurável. A sua principal tarefa é configurar o circuito FPGA de acordo com o modelo físico produzido pelo módulo PHYS.

Os blocos lógicos do circuito FPGA XC6216 têm uma unidade funcional configurável comum, mas os recursos de encaminhamento associados são muito variados. O conjunto de recursos depende da posição que o bloco lógico na matriz física dos blocos. Para representar todas as combinações possíveis, o sistema ReDiFlex define uma hierarquia de classes (com base na classe `cell`) cujas relações de dependência permitem modelar os recursos existentes. Baseada nessa hierarquia, o sistema define uma função genérica (`encode-cell`) que gera os valores de configuração e os usa para configurar efectivamente o circuito FPGA.

A configuração das zonas de encaminhamento entre fatias é feita usando a informação simbólica usada pelo algoritmo de encaminhamento para definir a configuração de blocos lógicos das classes apropriadas e invocando seguidamente os serviços de `encode-cell`.

Em algumas circunstâncias, a configuração presente no co-processador deve ser modificada, em particular, para modificar a origem do sinal de relógio ou para alterar a configuração do multiplexador de protecção RP. O sistema ReDiFlex inclui funções que executam essas modificações directamente sobre a configuração residente.

O sistema ReDiFlex inclui rotinas de baixo nível para aceder directamente aos blocos lógicos do co-processador (para configuração e para transferência de dados), controlo da origem e características do sinal de relógio, bem como inicialização da placa H.O.T. Works e do circuito FPGA. Estas rotinas são implementadas por chamadas (via uma fina camada implementada em C/C++) ao controlador de dispositivos integrado no sistema operativo.

8 Sistema de apoio à execução de aplicações

Após a especificação de uma rede de operadores, o utilizador pretende usar a correspondente implementação para o processamento de dados. A forma de definir uma rede foi tratada no capítulo 5; a maneira de a utilizar no processamento de dados é o objecto do presente capítulo, que assim termina a descrição dos meios postos ao dispor do utilizador interactivo do sistema ReDiFlex ou do programador de aplicações (conforme já anteriormente representado na figura 4.4). As tarefas descritas neste capítulo são implementadas pelo módulo RT (*runtime*).

A secção 8.1 descreve os vários passos associados ao processamento de dados com o co-processador reconfigurável e a forma como estes devem ser combinados, i.e. o modelo de execução que permite obter os resultados especificados pelo modelo funcional. As principais tarefas a coordenar são a reconfiguração do co-processador, a transferência dos valores de entrada para o circuito aí implementado e a recolha dos resultados produzidos.

A secção descreve ainda como é que o sistema ReDiFlex organiza o acesso do utilizador aos serviços disponibilizados por este módulo, em particular ao processamento de dados com o co-processador e à reconfiguração dinâmica de operadores. Também são abordadas as diferenças entre o tratamento de redes simples e de redes hierárquicas.

Na análise da maneira de controlar o processamento de dados é útil distinguir dois contextos de utilização (conforme já feito na secção 5.1). No primeiro existe apenas um conjunto de dados a processar; no segundo pretende-se tratar sequencialmente vários conjuntos de dados

de entrada. O interesse da distinção entre estes contextos, escalar e vectorial, bem como as suas consequências, são tratados na secção 8.1.1.

A secção 8.2 descreve os detalhes operacionais das funções de controlo de execução em contexto escalar (sec. 8.2.1) e vectorial (sec. 8.2.2). O ênfase da análise é colocado no tratamento de redes simples, já que as redes hierárquicas são tratadas por uma extensão conceptualmente simples já descrita na secção 8.1. A secção termina com uma análise da implementação da principal função de interface para reconfiguração dinâmica, `mutate-operator`. Embora esta função seja referida em capítulos anteriores, a sua descrição detalhada foi deixada para este capítulo porque envolve a combinação de serviços de vários módulos, combinação essa que no contexto do sistema ReDiFlex apenas ocorre no módulo RT.

Intimamente ligada com o controlo da operação do co-processador está a capacidade de obter informação pertinente sobre a implementação das redes e da forma como é efectuada a gestão do co-processador (incluindo transferências de dados e operações de reconfiguração). A secção 8.3 descreve os meios disponibilizados pelo sistema ReDiFlex para efectuar a traçagem das operações de gestão de *hardware*, bem como para inspeccionar interactivamente e ciclo-a-ciclo o estado de todo o sistema de apoio à execução, incluindo o estado do co-processador reconfigurável.

8.1 Modelo de execução

A execução do circuito que implementa uma rede de operadores é controlada por uma rotina executada pelo processador principal, o que permite implementar facilmente estratégias alternativas de controlo. O modelo de execução para uma rede não-hierárquica é conceptualmente simples:

- Configurar o co-processador. Esta operação só é necessária se a configuração desejada por diferente da configuração actual.
- Estabelecer os valores de entrada. Por construção, os registos de escrita usados para implementar as entradas da rede estão sempre na primeira fatia e podem ser inicializados sem influenciarem a operação dos circuito por eles alimentados.
- Activar o sinal de relógio o número de vezes necessário (o que é determinado pelas características dos componentes usados para implementar os diversos operadores). Em termos da implementação, usa-se o sinal global `G1` como sinal de relógio (cf. sec. 7.3).
- Recolher os resultados.

A execução das duas últimas tarefas está entrelaçada: um resultado está disponível num ciclo de relógio bem determinado, altura em que deve ser recolhido pelo sistema de execução, sob pena de se perder nos ciclos seguintes.

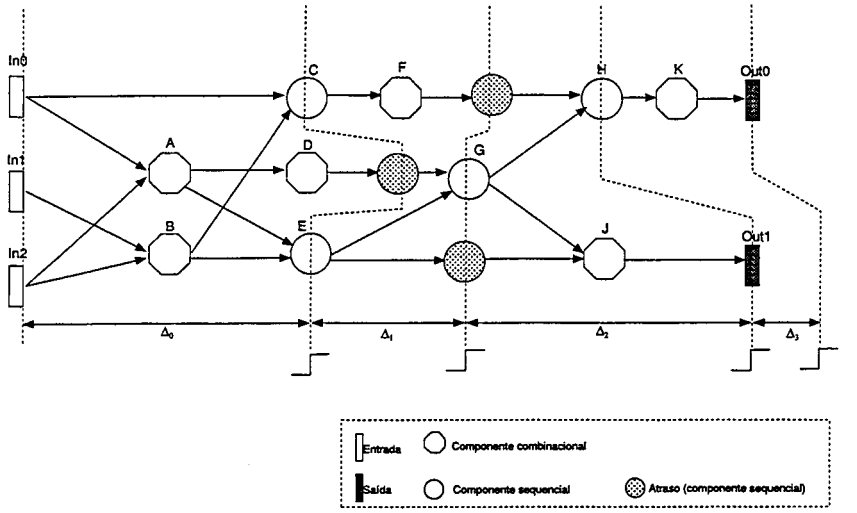


Figura 8.1: Exemplo da evolução temporal do estado de uma rede de operadores.

Como exemplo, considere-se a rede de operadores da figura 8.1, em que para além dos operadores e das suas interligações, se indica o tipo de implementação de cada operador (combinacional ou sequencial) e o ciclo de relógio em que cada operador produz os respectivos resultados. Ao todo, são necessários quatro ciclos de relógio para serem produzidos os resultados. O valor da saída `Out1` (i.e., o conteúdo do registo respectivo) está correcto durante o terceiro ciclo (Δ_3), altura em que deve ser "lido" pelo processador principal. Já o valor da saída `Out0` só está pronto depois do quarto flanco ascendente do sinal de relógio. Para esta rede, a sequência de operações é:

1. Inicializar `In0`, `In1` e `In2`.
2. Activar o sinal de relógio por três ciclos.
3. Recolher o valor de `Out1`.
4. Activar o sinal de relógio por um ciclo.
5. Recolher o valor de `Out0`.

A existência de operadores com variáveis de estado complica ligeiramente o procedimento, já que é necessário inicializar e/ou ler os valores dos registos internos dos componentes no ciclo de relógio exacto. Se, por suposição, o operador `G` tiver um registo interno que se deseje inicializar a zero e cujo valor depois da operação também deva ser incluído entre os resultados da execução, a sequência de operações passaria a ser a seguinte:

1. Inicializar `In0`, `In1` e `In2`.
2. Activar o sinal de relógio por um ciclo.
3. Inicializar o registo interno de `G` a zero.
4. Activar o sinal de relógio por um ciclo.
5. Recolher o valor do registo interno de `G`.
6. Activar o sinal de relógio por um ciclo.
7. Recolher o valor de `Out1`.
8. Activar o sinal de relógio por um ciclo.
9. Recolher o valor de `Out0`.

A inicialização de geradores e a recolha dos resultados de acumuladores são casos particulares do acesso a registos internos, para escrita e leitura, respectivamente.

O procedimento esboçado tem uma complicação adicional quando a implementação da rede simples está dividida em vários grupos de componentes, na sequência de uma decomposição a nível físico. Neste caso, deve ser efectuada uma sequência de reconfigurações, uma por cada grupo, de acordo com o seguinte esquema geral:

1. Configurar o co-processador com o primeiro grupo de componentes.
2. Activar o co-processador segundo a abordagem descrita anteriormente. Em consequência dessa activação podem ser obtidos alguns dos resultados desejados. Se este for o último grupo, está terminada a execução. Caso contrário, é obtido o valor do registo especial de saída inserido aquando da decomposição (cf. sec. 5.2.3).
3. Configurar o co-processador com o grupo de componentes seguinte. Esse grupo inclui um registo especial de entrada, que deve ser inicializado com o valor do registo especial de saída do grupo anterior. Repetir o passo anterior.

Em geral existem portanto três tarefas entrelaçadas: activar o sinal de relógio, ler/escrever registos (incluindo as entradas e saídas primárias) e reconfiguração do co-processador. O sistema ReDiFlex inclui uma rotina, `simple-opnet-runtime-management` (cf. sec. 8.2.1), que gere a execução combinada de todas estas actividades. Esta rotina evita proceder a reconfigurações desnecessárias, prescindindo dessa operação sempre que o co-processador esteja já devidamente configurado. Tal situação ocorre quando a rotina é invocada repetidamente para uma rede composta por um único grupo de componentes, uma situação particular que ocorre com alguma frequência na utilização prática do sistema.

Conforme já foi indicado na secção 5.3 (pág. 180),o campo `data-io-itf` da classe `operator-network-impl` contém a informação necessária para o sistema de apoio à execução efectuar a transferência de dados entre programa e co-processador. Este campo, do tipo `data-io-map`, inclui uma lista das operações de leitura/escrita de registos a efectuar em cada ciclo. Para além disso também define a interface da função de gestão de execução (a função `simple-opnet-runtime-management` ou outra função alternativa). A invocação de uma função de gestão de execução deve assumir a seguinte forma:

```
(func <opnet-impl> <arg1> <arg2> ... <argN>) ⇒ (<res1> <res2> ... <resM>)
```

em que `<opnet-impl>` é um objecto da classe `operator-network-impl` e `<arg1>` a `<argN>` representam os valores das `N` entradas primárias da rede. A correspondência entre argumentos e entradas primárias é definida por `data-io-itf` de tal maneira que `<arg1>` corresponde à primeira entrada mencionada num comando `define-operator-network`, `<arg2>` à segunda e assim sucessivamente. A função deve produzir uma lista de resultados com `M` valores correspondentes às saídas primárias e a valores de variáveis de estado especificadas pelo utilizador. Para uma rede com `P` saídas primárias, temos $P \leq M$, e os primeiros `P` resultados correspondem aos valores das saídas primárias pela ordem em que são mencionadas na definição da rede, enquanto os resultados seguintes correspondem a variáveis de estado dos operadores (i.e., registos internos de componentes) que o utilizador define como pertencendo à interface externa da rede.

A especificação das variáveis de estado que se entendem constituir resultados do processamento efectuado pela rede é feita por invocação da função `define-interface-function` da seguinte forma:

```
(define-interface-function <fn-name> <opnet-impl> (<op1> <var1>) ... (<opK> <varK>)
    ⇒ interface-func
```

em que `<fn-name>` é o nome da função a definir, `<opnet-impl>` é um objecto da classe `operator-network-impl`[1] que especifica a rede, e o par (`<opK>` `<varK>`) especifica o nome do operador e da respectiva variável de estado a observar. Apenas os dois primeiros argumentos são obrigatórios. A função `define-interface-function` retorna uma função de acesso e define o símbolo `<fn-name>` como o seu respectivo nome. Após a execução desta função o nome `<fn-name>` pode ser usado como outra função qualquer de Lisp para usar a implementação reconfigurável da rede. Esta nova função é invocada da mesma forma que uma função de gestão de execução, excepto que o parâmetro `opnet-impl` deve ser omitido, visto a função já estar especializada para a rede especificada na chamada a `define-interface-function`. Internamente, a função produzida por `define-interface-function` invoca uma função de gestão de execução (por exemplo, `simple-opnet-runtime-management`) com os argumentos que lhe são passados e retorna os respectivos resultados.

[1] Também pode ser usado um objecto da classe `hierarchical-operator-network-impl`.

Supondo que a variável `opnet-implementation` contém a especificação da rede da figura 8.1 e que o operador `G` tem um registo interno chamado `state` a invocação

```
(define-interface-function 'my-network opnet-implementation '(G "state"))
```

define uma função chamada `my-network` que aceita três argumentos (correspondentes às três entradas primárias) e produz uma lista de três resultados (dois correspondentes às saídas primárias e um terceiro com o valor da variável de estado "state"). Por exemplo, a invocação

```
(my-network 43 71 11) ⇒ (5 17 29)
```

configura o co-processador com a implementação da rede desejada, activa a sua execução de acordo com a estratégia descrita anteriormente, e retorna os resultados produzidos. Tudo isto é efectuado sem intervenção directa do utilizador.

Para resumir, um utilizador que queira usar a infra-estrutura reconfigurável deve executar as seguintes tarefas:

1. Definir uma rede de operadores com a funcionalidade desejada, usando o comando `define-operator-network`.
2. Produzir uma implementação dessa rede, invocando a função `make-opnet-impl`.
3. Definir uma função de interface via `define-interface-function`.
4. Invocar a função assim definida sempre que desejar utilizar o co-processador.
5. Entre invocações da função de interface, o utilizador pode usar as funções `set-parameter` e `mutate-operator` para alterar o comportamento da rede (e, por consequência, modificar a implementação usada para configurar o co-processador).

O tratamento de uma rede hierárquica é uma extensão da abordagem delineada atrás. Uma rede deste tipo é obtida por decomposição funcional da especificação, processo que resulta numa organização em dois níveis, na qual os elementos do nível inferior são sempre redes simples. O resultado da decomposição funcional é a especificação de um conjunto de sub-redes, do seu sequenciamento temporal e do fluxo de informação entre elas (cf. sec. 5.2.3). Cada sub-rede é implementada independentemente, sendo a informação resultante guardada num objecto do tipo `hierachical-operator-network-impl` (cf. sec. 5.3), cujo campo `data-io-itf`, do tipo `toplevel-data-io-map`, contém a informação necessária para construir a função de interface e para gerir as transferências de informação entre sub-redes.

A informação contida num objecto do tipo `toplevel-data-io-map` está conceptualmente dividida em duas partes. A primeira é especificação das características da função de interface, i.e. a correspondência entre argumentos da função e entradas primárias, por um lado, e a correspondência entre resultados e saídas primárias ou variáveis de estado, por outro. Esta

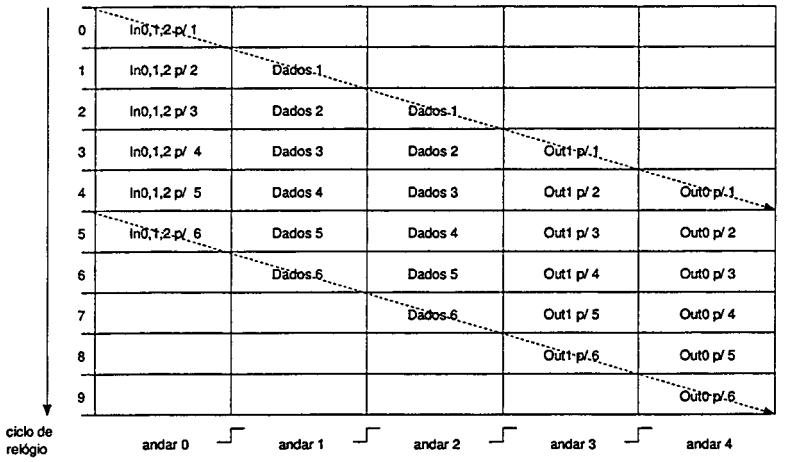


Figura 8.2: Operação em modo *pipelined* da rede da figura 8.1.

correspondência é mais complicada que para as redes simples, porque é preciso especificar em que sub-rede estão os respectivos componentes. A segunda parte é a especificação da informação a transferir entre as diversas sub-redes, que consiste essencialmente em especificar a forma de construir as chamadas às funções de interface das sub-redes, i.e. em definir a origem dos valores a passar como argumentos e o destino dos resultados obtidos.

A criação da função de interface implica ainda a criação das funções de interface das sub-redes constituintes. Esta tarefa é executada por uma versão de `define-interface-function` especializada para redes hierárquicas, que invoca a função homóloga para as sub-redes simples.

Conceptualmente, a função de interface de uma rede hierárquica invoca sucessivamente todas as funções de interface das sub-redes, utilizando os resultados de cada invocação para construir a sua própria lista de resultados ou a invocação da função de interface subsequente. No sistema ReDiFlex, a função `hierarchical-opnet-runtime-management` gere este processo, que ultimamente envolve a invocação da função de gestão das sub-redes simples (`simple-opnet-runtime-management` ou equivalente) via a função de interface correspondente.

8.1.1 Contextos de execução

Na secção 5.1 já se referiu a possibilidade de o processamento especificado por uma rede de operadores ser executada em dois contextos diferentes, escalar ou vectorial. O contexto escalar é caracterizado pelo processamento de apenas um conjunto de valores de entrada. Num contexto vectorial pretende-se usar a rede para processar consecutivamente uma sequência de n conjuntos de valores de entrada para se obterem n conjuntos de valores de saída.

Redes com geradores ou acumuladores apenas têm interesse se usados em modo vectorial. Em modo escalar, um gerador é conceptualmente equivalente a uma entrada e um acumulador a uma saída precedida de um circuito combinacional. Contudo o sistema ReDiFlex permite que redes com estes tipos de elementos sejam usadas em modo escalar, porque tal não é logicamente inconsistente.

Para além de permitir usar de modo útil geradores e acumuladores, outro factor de interesse em distinguir entre os dois contextos reside no facto de o contexto vectorial ser mais eficiente por ser possível usar a implementação da rede em modo *pipelined*. Neste modo, os valores de entrada são fornecidos ao co-processador em ciclos de relógio consecutivos, enquanto o co-processador vai produzindo os resultados para os valores de entrada anteriores. A evolução espácio-temporal dos dados para uma rede com cinco andares (i.e., que necessita de quatro ciclos de relógio para produzir todos os resultados correspondentes a um conjunto de valores de entrada) e seis conjuntos de valores de entrada está ilustrada na figura 8.2.

Em geral, para uma rede cuja implementação necessite de p ciclos de relógio para processar um conjunto de valores de entrada, são necessários $n + p$ ciclos para processar n conjuntos em modo *pipelined*. A operação em modo escalar necessitaria de $n \times p$ ciclos para obter os mesmos resultados. Para valores de n muito superiores a p, obtem-se uma redução do tempo de processamento por um factor de $\frac{n+p}{n \times p} \rightarrow \frac{1}{p}$.

A operação de uma rede em modo vectorial permite ainda reduzir o número de reconfigurações efectuadas, à custa de espaço de memória para guardar informações temporárias. A estratégia consiste em processar todos os n conjuntos de dados com o mesmo grupo de componentes, armazenando todos os n resultados intermédios; apenas então se procede à reconfiguração do co-processador com o grupo seguinte de componentes, que é alimentado com os n resultados intermédios, e assim sucessivamente. Consequentemente, o número de reconfigurações em modo vectorial para n conjuntos de dados é o mesmo que em modo escalar para *apenas um* conjunto de dados. Por comparação com n execuções em modo escalar, o número de reconfigurações vem reduzido de um factor de $\frac{1}{n}$. Esta estratégia permite assim amortizar o tempo de reconfiguração do co-processador pelos n conjuntos de dados.

Para gerir a execução de uma rede simples em modo vectorial, o sistema ReDiFlex dispõe da função `simple-opnet-v-runtime-management`, a versão vectorial de `simple-opnet-runtime-management`. As funções de controlo de execução em contexto vectorial devem ser invocadas da seguinte forma:

```
(vfunc <opnet-impl> n <arg1> <arg2> ... <argN>) ⇒ (<res1> <res2> ... <resM>)
```

onde os argumentos têm o mesmo significado que em contexto escalar, excepto que podem ser vectores de números e não simplesmente números. O argumento adicional `n` indica o número de conjuntos de entrada a processar. Os argumentos da função podem ser vectores de qualquer tamanho ou números. Estes são interpretados como sendo vectores de comprimento 1. Se o tamanho do vector for superior a n, apenas os primeiros n elementos são usados; se

for inferior, os elementos existentes são re-utilizados ciclicamente. Os elementos da lista de resultados são vectores de n elementos ou números. Este último caso acontece apenas para saídas que correspondem a variáveis de estado, conforme se descreve mais abaixo.

A definição de uma função de interface para operação em modo vectorial é feita por

```
(define-v-interface-function <fn-name> <opnet-impl> (<op1> <var1> <mode1>) ...
                                                    (<opK> <varK> <modeK>))
    ⇒ interface-func
```

em que os argumentos têm o mesmo significado que em contexto escalar, excepto que cada variável de estado tem associado um modo de leitura opcional, que pode ser `:always` ou `:at-end`. No primeiro caso, que é o valor por omissão, a variável de estado é lida sempre que é processado um novo conjunto de dados; no segundo caso, a variável de estado é lida apenas uma vez, no fim do processamento. O segundo modo é apropriado, por exemplo, para as variáveis de estado de acumuladores. De acordo com a especificação feita aqui, a invocação da função de gestão da execução em modo vectorial retorna um vector de n elementos ou apenas um número na correspondente posição da lista de resultados. Tal como acontece no caso escalar, a função criada por `define-v-interface-function` gere a infra-estrutura reconfigurável por invocação de uma função de controlo de execução apropriada.

A invocação de `define-v-interface-function` tem como resultado associar a função *interface-func* ao nome `<fn-name>`. A invocação desta função é similar à invocação da função de gestão correspondente, excepto que o parâmetro `opnet-impl` deve ser omitido. A função de interface retorna os mesmos valores que a função de gestão de execução em que se baseia.

Da perspectiva do utilizador, as diferença entre os dois modos de operação resumem-se à necessidade de usar comandos diferentes para definir as respectivas funções de interface e ao modo ligeiramente diferente de as invocar. Por último, note-se que nada impede a existência de duas funções de interface, uma vectorial e outra escalar, para a mesma rede.

A expansão a redes hierárquicas faz-se, à semelhança do caso escalar, através de uma versão especializada de `define-v-interface-function`, que, como anteriormente, delega a gestão da execução numa função, `hierarchical-opnet-v-runtime-management`, que sequencia apropriadamente a invocação das funções de interface vectorial das sub-redes.

8.2 Interface de execução

Esta secção apresenta alguns detalhes da implementação das funções de gestão de execução para os contextos escalar e vectorial. Também é apresentada a implementação do controlo de reconfiguração dinâmica de operadores, a função `change-operator` referida na secção 5.1 (pág. 161). O ênfase desta secção está colocado na gestão de execução de redes simples; a extensão a redes hierárquicas é feita segundo a abordagem delineada na secção anterior.

8.2.1 Contexto escalar

Por uma questão de flexibilidade, o sistema ReDiFlex dispõe internamente de funções para a gestão da execução de uma rede simples em contexto escalar com formas de invocação diferentes. A já referida função `simple-opnet-runtime-management` aceita múltiplos argumentos (conforme a especificação da secção anterior), enquanto `simple-opnet-runtime-management*` aceita apenas dois: a representação interna da implementação e uma lista de valores a fornecer ao co-processador. A função `simple-opnet-runtime-management` faz simplesmente a conversão dos argumentos e invoca `simple-opnet-runtime-management*` para executar o trabalho.

```
(defun simple-opnet-runtime-management (op-impl &rest args)
  "Multiple-argument interface to SIMPLE-OPNET-RUNTIME-MANAGEMENT."
  (simple-opnet-runtime-management* op-impl args))
```

A função `simple-opnet-runtime-management*` gere a execução do processamento definido por uma rede simples, de acordo com a abordagem descrita na secção 8.1.

```
(defun simple-opnet-runtime-management* (op-impl arg-lst)
  "Manage execution of network implementation and return a list of results."
  (unless *die-available*
    (return-from general-data-interface*  nil))
  (when (and *exclusive-mode* *current-opnet-impl*)
    (when (and *die-locked* (not (eq *current-opnet-impl* op-impl)))
      (return-from general-data-interface* nil)))
  (initialize-board-with-opnet op-impl)
  (load-initial-data op-impl arg-lst)
  (let ((current-subcluster-index 0)
        (results (make-array (n-outputs (data-io-itf op-impl))))
        (nticks (clock-ticks op-impl))
        (om (out-map (data-io-itf op-impl))))
    (dotimes (current-clock-cycle (1+ nticks))
      (let ((op-list (operations (svref om current-clock-cycle))))
        (loop for current-op in op-list
          do (progn
               (pop op-list)
               (if (read-operation-p current-op)
                   (setf (svref results (seq-number current-op))
                         (read-user-register (svref (setups op-impl) clu-idx)
                                             (reg-name current-op)))
                   (write-user-register (svref (setups op-impl) clu-idx)
                                        (reg-name current-op) (reg-value current-op)))))
        (unless (= current-clock-cycle nticks)
          (when *gl-clock-hook* (funcall *gl-clock-hook* current-clock-cycle))
```

```
27          (apply-clock-G1))
28        (when (and (= current-clock-cycle
29                      (last-clock-cycle op-impl current-subcluster-index))
30                   (/= current-subcluster-index (1- (length (clusters op-impl)))))
31          (switch-to-next-subcluster op-impl clu-idx)
32          (incf current-subcluster-index)))
33      (nreverse (coerce results 'list))))
```

A função começa por verificar se pode usar o co-processador (linha 3 e seguintes); em caso negativo, retorna imediatamente a lista vazia. De seguida, procede-se à inicialização do co-processador (linha 8) por invocação da função auxiliar `initialize-board-with-opnet`. Esta função configura a placa H.O.T. Works, inicializa as ligações ao sinal de relógio `G1` (cf. sec. 7.3, pág. 232) e configura o co-processador com o primeiro grupo de componentes.

O processo de execução continua com o estabelecimento dos valores iniciais dos registos de entrada a partir dos argumentos (invocação de `load-initial-data` na linha 9). O âmago do processamento começa na linha 10 e consiste principalmente de um ciclo iterativo (linha 14) que percorre todos os ciclos de relógio, executando para cada um as operações de acesso a registos apropriadas (conforme especificadas na lista `op-list`, linha 15). O corpo da iteração começa na linha 16 e a execução das operações propriamente ditas começa na linha 19 com a determinação do tipo de acesso, seguida da execução do respectivo acesso. Notar que cada grupo necessita de pelo menos um ciclo de relógio (para activar a captura dos valores dos registos de saída).

Após a execução de todas as operações de um ciclo, procede-se à activação do sinal de relógio (linha 25 e seguintes). Caso a variável especial `*g1-clock-hook*` esteja associada a uma função (um "gancho", em terminologia Lisp), a correspondente função é invocada. A finalidade de um "gancho" é permitir a execução de funções arbitrárias em estados bem definidos do processamento. Neste caso, a cada avanço do relógio é invocada a função associada a `*g1-clock-hook*`. A secção 8.3 descreve os "ganchos" existentes em mais pormenor.

Caso o ciclo de relógio seja o último ciclo do grupo actual, procede-se à reconfiguração do co-processador com o grupo seguinte (se existir). A reconfiguração é executa pela função auxiliar `switch-to-next-subcluster`, invocada na linha 31, e cujo processamento é similar ao executado por `initialize-board-with-opnet`.

Após o processamento de todos os ciclos de relógio, a função `simple-opnet-runtime-management*` retorna a lista dos resultados obtidos (linha 33).

Esta função de controlo da execução, por ser de utilização geral, é relativamente complexa, com um custo proporcional em termos de desempenho. Para reduzir este custo em certos casos particulares, o sistema ReDiFlex inclui duas funções alternativas:

`one-cluster-opnet-runtime-management` Esta função destina-se a ser usada com redes cuja implementação tenha apenas um grupo de componentes (p. ex., utilizável com as im-

plementações de sub-redes resultantes de decomposição funcional).

`one-combinational-opnet-runtime-management` Esta função destina-se a ser usada com redes implementadas por apenas um grupo de componentes combinacionais. Este tipo de redes usa apenas um ciclo de relógio para actualizar os registos de saída.

O sistema também inclui as respectivas variantes `one-cluster-opnet-runtime-management*` e `one-combinational-opnet-runtime-management*`.

Uma alternativa mais geral consiste em analisar a implementação de uma rede e construir uma função especialmente ajustada à respectiva estrutura. Uma função deste tipo permitiria evitar os diversos ciclos e testes inerentes às versões mais gerais. Os sistemas Lisp estão particularmente bem apetrechados para as tarefas de geração incremental de código como esta. Esta abordagem não está actualmente implementada no sistema ReDiFlex, mas constitui uma via de evolução interessante.

A implementação de `define-interface-function` é conceptualmente bastante simples, já que se limita a verificar se a implementação da rede corresponde a um dos casos particulares e a escolher a função de controlo de execução apropriada.

```
1  (defun define-interface-function (fn-name impl)
2    (when (or (macro-function fn-name)
3              (special-operator-p fn-name))
4      (error "Will not redefine macro or special operator."))
5    (setf (symbol-function fn-name) (create-interface-function impl)))
6
7  (defun create-interface-function (impl)
8    "Create a new function that interfaces with IMPL network."
9    (flet ((aux-geral (&rest args)
10             (simple-opnet-runtime-management* impl args))
11           (aux-one-cluster (&rest args)
12             (one-cluster-opnet-runtime-management* impl args))
13           (aux-one-combinational (&rest args)
14             (one-combinational-opnt-runtime-management* impl args)))
15     (if (one-cluster-opnet-p impl)
16         (if (one-combinational-cluster-opnet-p impl)
17             #'aux-one-combinational
18             #'aux-one-cluster)
19         #'aux-geral)))
```

A função `define-interface-function` verifica se o nome especificado para a função é aceitável (linha 2 e seguintes) e associa-o a uma função apropriada (linha 5) criada pela função `create-interface-function`. Por sua vez, esta função determina o tipo de implementação (linha 15 e seguintes) e retorna uma função auxiliar (definidas a partir da linha 9), cuja imple-

mentação consiste apenas na invocação da função de controlo apropriada com os argumentos correctos.

8.2.2 Contexto vectorial

O sistema ReDiFlex inclui também funções de controlo de execução em contexto vectorial. As diversas funções seguem uma organização similar à que está descrita na subsecção anterior. A função mais importante é `simple-opnet-v-runtime-management*`.

```
(defun simple-opnet-v-runtime-management* (op-impl n arg-lst)
  "Manage execution of network implementation in a vectorial context."
  (unless *die-available*
    (return-from general-vdata-interface*  nil))
  (when (and *exclusive-mode* *current-opnet-impl*)
    (when (and *die-locked* (not (eq *current-opnet-impl* op-impl)))
      (return-from general-vdata-interface* nil)))
  (initialize-board-with-opnet op-impl)

  (let* ((results (make-array (n-outputs (data-io-itf op-impl)))
         (temp-vect (make-array n :element-type 'integer))
         (global-ticks-min 0)
         (global-ticks-max (svref (cluster-ticks op-impl) 0))
         (om (out-map (data-io-itf op-impl))))
    (dotimes (i (length results))
      (if (scalar-result-p i (data-io-itf op-impl))
          (setf (svref results i) 0)
          (setf (svref results i) (make-array n :fill-pointer 0 :element-type 'integer))))
    (load-initial-vdata op-impl 0 arg-lst)

    (dotimes (cluster-counter (length (clusters op-impl)))
      (let ((nticks (svref (cluster-ticks op-impl) cluster-counter)))
        (dotimes (i (+ n nticks))
          (let ((global-start  (+ global-ticks-min
                                  (if (>= i n) (1+ (- i n)) 0)))
                (global-end (- global-ticks-max
                               (if (<= i nticks) (- nticks i) 0))))
            (loop for step from global-start upto global-end
                  do (let ((op-list (operations (svref om step))))
                       (loop
                          (let ((current-op (car op-list)))
                            (unless (and current-op
                                         (= cluster-counter
```

```
34                                              (cluster-idx current-op)))
35                                (return))
36                               (pop op-list)
37                               (do-operation current-op results temp-vect
38                                             (svref (setups op-impl) cluster-counter)))))))
39
40          (when (= i (1- (+ n nticks)))
41            (return))
42          (when *gl-clock-v-hook* (funcall *gl-clock-v-hook* i))
43          (apply-clock-G1)
44          (when (< (1+ i) n)
45            (if (zerop cluster-counter)
46                (load-initial-vdata op-impl (1+ i) arg-lst)
47                (load-temporary-vdata op-impl cluster-counter (1+ i) temp-vect))))
48
49        (setf global-ticks-min global-ticks-min
50              global-ticks-max (+ global-ticks-max nticks)))
51
52      (when (< cluster-counter (1- (length (clusters op-impl))))
53        (vswitch-to-next-subcluster op-impl cluster-counter)
54        (load-temporary-vdata op-impl cluster-counter 0 temp-vect)))
55
56    (nreverse (coerce results 'list))))
```

A função começa por verificar se o co-processador está disponível (linha 3 e seguintes) e por proceder à sua inicialização com o primeiro grupo de componentes (linha 8). De seguida procede-se à inicialização dos vectores de resultados (linha 15 e seguintes).

Os registos de entrada do co-processador são inicializados (linha 19) antes de se dar início a um ciclo por todos os grupos de componentes (linha 21 e seguintes). Para cada grupo, é executado um ciclo, em que cada iteração corresponde a uma activação do sinal de relógio (linha 23 e seguintes). Em cada iteração, determina-se quais os conjuntos de operações de leitura ou escrita de registos a efectuar, as quais são executadas no ciclo que começa na linha 28. Este ciclo corresponde a percorrer uma linha horizontal da tabela da figura 8.2. A parte da linha que deve ser considerada é delimitada pelas variáveis `global-start` e `global-end`.

Após efectuar as operações de leitura/escrita necessárias (que incluem guardar os valores temporários associados aos registos introduzidos aquando da decomposição física), o sinal de relógio é activado (linha 40), e novos valores de entrada são enviados para o co-processador (linha 44 e seguintes). Os valores correspondem às entradas primárias (para o primeiro grupo) ou aos valores dos registos intermédios (para os restantes grupos).

Depois de todos os dados terem sido processados por um grupo de componentes é necessário passar ao próximo grupo (linha 52 e seguintes). No final do processamento, retorna-se a

lista dos resultados (linha 56).

As observações gerais sobre desempenho feitas no final da subsecção anterior aplicam-se igualmente ao contexto vectorial, pelo que o sistema ReDiFlex contém versões especializadas da função de controlo de execução. À semelhança do caso escalar, também aqui a escolha da função apropriada é feita por `create-v-interface-function`, a rotina que cria as funções de interface.

8.2.3 Reconfiguração de operadores

Esta subsecção apresenta a implementação da função `change-operator` referida na secção 5.1 (pág. 161). O objectivo desta rotina é proceder à reconfiguração do componente que implementa o respectivo operador. Conforme descrito nas secções 5.1 e 6.2.2, a parte central desta tarefa (a modificação da informação de configuração) é executada pelo método `mutate`. Contudo, são ainda necessários alguns passos adicionais para se ter um implementação utilizável. Em primeiro lugar, a informação de configuração tem de ser modificada para compensar as inversões de polaridade dos sinais de entrada introduzidas pelos percursos de encaminhamento. A necessidade destas alterações é descrita na secção 6.3.2. Em segundo lugar, é necessário reconfigurar o co-processador, caso o componente pertença ao grupo activo aquando da invocação de `mutate-operator`.

A implementação de `mutate-operator` é a seguinte:

```
1  (defun mutate-operator (impl opid &rest params)
2    "Modify the implementation of an operator."
3    (let* ((opnet (global-spec impl))
4           (op (get-operator-by-name opid (ops opnet))))
5      (unless op
6        (error "There is no operator instance called ~a~ in the network." opid))
7      (unless (frozen-p op)
8        (error "There is no implementation of operator instance ~a~ that can be changed."
9               opid))
10     (let ((success (funcall #'mutate op params)))
11       (unless success
12         (error "Failed to mutate operator instance ~a~." opid))
13       (let* ((op-cluster-idx (map-level-to-cluster (level op) impl))
14              (op-cluster (svref (clusters impl) op-cluster-idx))
15              (op-blk (blk op))
16              (slice-idx (loop for sl across (slices op-cluster)
17                               for i upfrom 0
18                               when (loop for b across (components sl)
19                                          when (eq b op-blk) return t
20                                          finally (return nil))
```

```
21                          return i
22                          finally (return nil))))
23        (unless slice-idx
24          (error "[CHANGE-OPERATOR] Internal error: Block is not in cluster."))
25        (when (plusp slice-idx)
26          (let ((netlst (svref (slicenets op-cluster) (1- slice-idx))))
27            (loop for net across netlst
28                  do (adjust-net-inversions-for-block net op-blk))))
29
30        (when (and (mapped impl) (= (active-cluster impl) op-cluster-idx))
31          (when (and *die-available* (not *die-locked*) (eq impl *current-opnet-impl*))
32            (download-block op-blk)))))))
```

Após algumas verificações iniciais dos argumentos, a função `mutate-operator` invoca o método `mutate` do componente correspondente (linha 10). Em caso de sucesso procede-se à compensação das inversões de polaridade dos sinais de entrada (linha 13 e seguintes). Para tal é preciso determinar a fatia a que o componente pertence (ciclo com início na linha 16) para, de seguida, proceder efectivamente ao ajuste da configuração (linha 25 e seguintes). O último passo consiste em reconfigurar o co-processador caso o componente alterado esteja a ser utilizado nesse momento (linha 30 e seguintes).

8.3 Traçagem e depuração

Em todas as aplicações podem ocorrer erros ou outros comportamentos anormais. O sistema ReDiFlex inclui alguns meios que permitem ao utilizador averiguar o estado do sistema de controlo de execução e do co-processador para tentar determinar a razão de eventuais comportamentos anormais da sua aplicação. As mesmas ferramentas são úteis para o desenvolvimento do próprio sistema ReDiFlex.

Os sistemas de Lisp incluem normalmente apoio considerável para tarefas relacionadas com a depuração de programas. Em particular, as funções `trace` e `step` permitem, respectivamente, fazer a traçagem da execução de uma função (i.e. obter uma listagem de todas as funções invocadas durante a execução, incluindo os valores dos respectivos argumentos e resultados produzidos) e executar uma função passo a passo. Outra função comum, a função `inspect`, permite inspeccionar interactivamente o estado de qualquer estrutura de dados.

A aplicação destas funções de depuração à implementação do sistema ReDiFlex permitiria determinar exactamente o seu estado, mas exigiria da parte do utilizador um conhecimento dos detalhes internos da implementação que se pretendem precisamente esconder. Por outro lado, estas funções não permitem averiguar o estado do co-processador.

Para colmatar estas lacunas, o sistema ReDiFlex permite o acesso regrado à informação sobre a activação de uma rede. O acesso é baseado em "ganchos", i.e. funções que são chamadas

em pontos bem definidos do processamento. Estes pontos são escolhidos de maneira a terem significado segundo o modelo de execução descrito na secção 8.1.

A implementação da função `simple-opnet-runtime-management*` (pág. 244) mostra um dos "ganchos" suportados pelo sistema: A função associada à variável especial `*gl-clock-hook*` é chamada sempre que o sinal de relógio é activado.

Os principais "ganchos" suportados pelo módulo RT de apoio à execução de aplicações são os seguintes:

`*gl-clock-hook*` A função a chamar sempre que o relógio `G1` é activado (em contexto escalar).

`*gl-clock-v-hook*` A função a chamar sempre que o relógio `G1` é activado (em contexto vectorial).

`*cluster-download-hook*` A função a invocar sempre que o co-processador é configurado com um grupo de componentes.

`*cluster-switch-hook*` A função a invocar sempre que se procede a uma troca de grupos.

`*subnetwork-interface-pre-hook*` Durante a utilização de redes hierárquicas, a função a invocar *antes* de invocar a função de interface de uma sub-rede.

`*subnetwork-interface-post-hook*` Durante a utilização de redes hierárquicas, a função a invocar *depois* de invocar a função de interface de uma sub-rede.

O sistema ReDiFlex inclui duas funções que modificam os diversos "ganchos" para produzir informação durante a utilização de uma rede. A função `install-default-runtime-hooks` configura os diferentes "ganchos" para imprimirem informação sobre as diversas operações que vão sendo efectuadas, fornecendo assim uma traçagem das diferentes etapas da actividade do sistema de controlo do co-processador.

A função `install-interactive-runtime-hooks` configura os "ganchos" com funções que permitem ao utilizador investigar interactivamente o estado do sistema de controlo de *hardware* e o estado do co-processador. Em particular, a seguir a cada activação do sinal de relógio, o utilizador pode determinar o valor de todos os registos internos da implementação. Esta configuração dos "ganchos" permite seguir a actividade do co-processador ciclo a ciclo.

Para os casos em que se pretenda inspeccionar o sistema apenas quando ocorrem certas condições, a função `install-break-runtime-hooks` configura os "ganchos" com funções que detectam a ocorrência da condição desejada, passando automaticamente a modo interactivo nesses casos. As condições a detectar são definidas pela função `set-runtime-break-condition`. As condições podem envolver valores dos registos internos ou etapas do processamento (p.ex. interromper o processamento quando for atingido o n-ésimo ciclo de relógio ou quando for activado um determinado grupo de componentes).

A função `reset-runtime-hooks` torna todos os "ganchos" inactivos.

O sistema inclui ainda uma função interactiva chamada `inspect-opnet` que permite ao utilizador obter interactivamente toda a informação pertinente sobre uma dada rede e respectiva implementação, incluindo informação sobre os componentes usados.

O sistema inclui ainda funções que permitem visualizar a estrutura de uma rede e o resultado do processamento físico:

`show-opnet-spec` Esta função invoca o programa VCG [Sander, 1995] para visualizar um grafo que representa a especificação de uma rede e obter informação sobre o nível e ciclo de relógio associados a cada operador. O programa VCG determina automaticamente o posicionamento dos nós do grafo (operadores) e das suas interligações de maneira a obter uma imagem razoável do grafo. Também permite fazer a impressão da imagem obtida.

`show-placed-scluster` Esta função invoca um programa para visualizar o arranjo físico de um grupo de componentes, bem como dos respectivos percursos de interligação. O programa externo, chamado "v6200", foi escrito especialmente para o sistema ReDiFlex em linguagem TCL e usa a biblioteca gráfica Tk para a visualização interactiva. Esta função é usada por `inspect-opnet` e pelas funções interactivas usadas por `install-interactive-runtime-hooks` e `install-break-runtime-hooks`.

O conjunto de meios de depuração e traçagem descritos nesta secção permitem ao utilizador obter confortavelmente informação detalhada sobre a implementação de uma rede de operadores e sobre todas as etapas da sua utilização.

8.4 Resumo

O módulo RT do sistema ReDiFlex inclui todas as funções necessárias para controlar a utilização do co-processador reconfigurável. A implementação destas funções combina os serviços dos três outros módulos (OPNET, PHYS e HWCTRL) para fornecer ao utilizador uma interface simples e bem integrada no ambiente de programação.

No sistema ReDiFlex a utilização de uma rede de operadores para processar informação é feita por funções escritas em Lisp, que controlam o comportamento temporal das implementações de maneira a serem obtidos os resultados especificados pelo modelo funcional. O controlo temporal do co-processador envolve sequenciar as operações de configuração, transferir os dados de entrada para o co-processador e retirar daí os resultados. No caso das redes hierárquicas produzidas por decomposição funcional, este processo deve ser repetido para cada sub-rede.

Ao nível da utilização, é conveniente distinguir entre contexto escalar e vectorial. No primeiro caso, existe apenas um conjunto de dados de entrada a processar, enquanto no segundo

se pretende tratar uma sequência de conjuntos de dados. O segundo caso pode ser tratado eficientemente por uma estratégia do tipo *pipeline*, minimizando o número de ciclos de relógio e de operações de reconfiguração necessários.

O sistema ReDiFlex dispõe, para cada um dos contextos, de várias funções de controlo da execução. As versões mais gerais podem ser usadas com qualquer tipo de rede simples, enquanto as versões especializadas tratam mais eficientemente os casos particulares de implementações com apenas um grupo de componentes e redes puramente combinacionais. O facto de a implementação do controlo de execução ser feita em *software* torna relativamente fácil a criação de versões especializadas das funções de controlo.

O utilizador não precisa de conhecer todos estes detalhes de implementação, já que o sistema ReDiFlex lhe permite definir de forma simples funções de interface para cada rede. Assim, cada rede é usada por invocação de uma função própria, o que permite ao utilizador encará-la como qualquer outra função de Lisp e abstrair do facto de a sua invocação implicar um conjunto complexo de actividades envolvendo o co-processador reconfigurável. As funções de interface a usar em contexto escalar e vectorial são definidas separadamente. Em última análise, cada uma delas é implementada com base nas diversas funções de controlo de execução existentes no sistema.

Outra tarefa assegurada pelo módulo RT é a reconfiguração dinâmica de componentes. Com base nos serviços disponibilizados pelo módulo PHYS, a função `mutate-operator` cria as novas configurações, altera-as para compensar a inversão de polaridade dos sinais de entrada provocada pelos circuitos de encaminhamento, e, se necessário, re-configura imediatamente o co-processador com a nova versão do componente. Desta forma, qualquer modificação de parâmetros de um operador é imediatamente reflectida na configuração do co-processador.

O módulo RT disponibiliza também um conjunto de meios para seguir as operações do sistema de controlo de execução e inspeccionar as principais estruturas de dados usadas. É possível obter uma traçagem das diversas etapas da utilização de uma rede, bem como seguir ciclo a ciclo a respectiva evolução. Também é possível interromper o processamento efectuado pelo co-processador quando certas condições são detectadas para proceder à inspecção interactiva do estado do sistema.

O sistema ReDiFlex inclui ainda ferramentas que permitem visualizar a estrutura de um modelo funcional e, para o nível físico, os resultados das fases de colocação e encaminhamento.

9 Conclusão

Este capítulo começa com algumas reflexões sobre o uso e aplicação de reconfiguração dinâmica inspiradas pela experiência ganha na concepção e implementação do sistema ReDiFlex (sec. 9.1). O capítulo resume ainda o trabalho realizado (sec. 9.2), e encerra com uma análise das perspectivas de trabalho futuro (sec. 9.3), tanto no contexto mais restrito do sistema ReDiFlex, como no campo mais lato do desenvolvimento e aplicação de co-processadores reconfiguráveis dinamicamente .

9.1 Reflexões sobre reconfiguração dinâmica

As observações desta secção aplicam-se especialmente a sistemas reconfiguráveis dinamicamente de características similares aos baseados na infra-estrutura reconfigurável usada neste trabalho, i.e., co-processadores reconfiguráveis com suporte para reconfiguração parcial. O principal cenário de utilização desta infra-estrutura é também aquele que serve de base ao trabalho descrito nos capítulos anteriores: aplicações em que as operações de reconfiguração não podem ser, em geral, determinadas antes da execução do programa (*off-line*) porque dependem dos resultados que vão sendo obtidos durante o processamento (tanto para a criação dos dados para configuração propriamente ditos, como para a determinação do momento em que as novas configurações devem ser activadas). Apesar desta situação, algumas observações aplicam-se igualmente a outras utilizações ou implementações de reconfiguração dinâmica.

A utilização de reconfiguração dinâmica na implementação de sistemas digitais pode ser comparada à utilização de código auto-modificável em programas de computadores. Em ambos os casos existe uma alteração do "objecto" que realiza tratamento de dados (o sistema digital num caso, o programa no outro) durante o próprio processamento. Ou, visto de uma perspectiva diferente, a própria realização do processamento transforma-se em objecto desse mesmo processamento. Em ambos os casos, a flexibilidade acrescida é difícil de gerir de forma

construtiva, sendo praticamente imprescindível encontrar formas de regular a sua utilização. No caso da reconfiguração dinâmica, isso pode ser feito pela adopção de modelos conceptuais que permitam realizar as vantagens da reconfiguração dinâmica (melhor aproveitamento dos recursos físicos) sem produzir aplicações cuja estrutura interna seja imperscrutável. O sistema ReDiFlex inclui um modelo funcional simples para cumprir este objectivo, mas é clara a necessidade de investigar outros modelos, idealmente passíveis de especificação formal.

No sistema ReDiFlex optou-se por um modelo relativamente abstracto, em que grande parte dos detalhes de processamento são omitidos. A escolha foi feita por duas razões: (i) ter um modelo facilmente acessível a programadores sem experiência de sistemas digitais, e (ii) porque permite representar a reconfiguração dinâmica em termos de alteração de parâmetros dos elementos constituinte, um processo conceptualmente simples. Contudo este tipo de modelos pode levar ao sub-aproveitamento de alguns recursos físicos dos circuitos FPGA, pelo que se justifica igualmente a investigação de modelos que permitam a manipulação mais directa dos recursos físicos. Neste contexto, teria particular interesse a expansão semanticamente integrada das linguagens de descrição de *hardware* actuais para permitirem a alteração dinâmica dos circuitos descritos, uma tarefa que não se afigura trivial.

Uma característica importante do modelo funcional usado no sistema ReDiFlex é o de preservar a localidade das alterações, i.e., a reconfiguração dinâmica de um elemento nunca implica a realização de alterações globais da implementação. Esta é uma característica desejável porque permite aproveitar de forma directa a capacidade de reconfiguração parcial de circuitos FPGA, para além de circunscrever o trabalho de geração de reconfigurações durante a execução. Caso fosse permitido alterar número e dimensão dos portos de entrada/saída dos operadores ou a dimensão geométrica dos componentes associados, tal limitação não seria possível; pela mesma razão o modelo não contempla a inserção ou remoção de operadores. O sistema ReDiFlex disponibiliza, contudo, os meios para se obterem efeitos similares, já que é possível definir novas redes de operadores durante a execução da aplicação, redes essas que podem ser variações de redes já definidas. As novas redes são tratadas de forma independente, i.e., os resultados do tratamento de redes similares não é aproveitado. Contudo, como por suposição as diferenças entre as redes implicam diferenças globais ao nível da implementação, esse aproveitamento não traria vantagens significativas ao nível do desempenho. A possibilidade de "clonagem" e alteração de redes não faz parte do modelo de especificação, é antes uma consequência da flexibilidade do sistema ReDiFlex.

Uma característica importante do circuito FPGA XC6216 é a regularidade da sua arquitectura interna, que permitiu simplificar significativamente o modelo físico. Em particular, no sistema ReDiFlex foi possível assumir que as configurações dos componentes são praticamente independentes do seu posicionamento (embora isso tenha implicado sub-aproveitar algumas pistas internas). As simplificações resultantes são particularmente importantes quando se pretende gerar dinamicamente as novas configurações durante a execução da aplicação.

A evolução das arquitecturas internas de circuitos FPGA tem ocorrido no sentido de uma

complexidade crescente e da inclusão de módulos com funcionalidade específica (memórias, multiplicadores e circuitos PLL são os principais exemplos). Tudo isto se reflecte na necessidade de um tratamento muito mais complexo do modelo físico. Por exemplo, a comparação entre a organização do sistema JBits da companhia Xilinx, uma biblioteca em linguagem Java que permite definir configurações para circuitos FPGA Virtex-II, e a representação de configurações no sistema ReDiFlex, mostra claramente a enorme dimensão do esforço requerido para representar os recursos físicos e a organização complexa dos circuitos FPGA mais avançados. Apesar disso, a família Virtex-II suporta reconfiguração parcial (mas com uma granularidade superior à da família XC6200 e sem o benefício da interface FastMap), embora necessite de cuidados acrescidos, resultantes do facto de poderem ocorrer conflitos eléctricos durante a reconfiguração. A arquitectura da família XC6200 é exemplarmente imune a este tipo de problemas, o que em muito contribui para a simplicidade e robustez da sua utilização.

A complexidade crescente das arquitecturas internas dos circuitos FPGA não constitui, por si só, um impedimento essencial à utilização de reconfiguração dinâmica. Existem muitas aplicações em que as configurações podem ser geradas *off-line*, deixando-se para a fase de execução apenas a determinação dos momentos de reconfiguração. Nestas aplicações a complexidade da arquitectura interna não tem influência sobre o desempenho da aplicação (excepto indirectamente, já que uma arquitectura complexa necessita de uma quantidade maior de informação de configuração). Contudo essa complexidade reflecte-se directamente na necessidade de criar ferramentas automáticas mais sofisticadas. Aumenta também a dificuldade em aproveitar todos os recursos disponíveis, em especial, quando a especificação é feita a um nível elevado, i.e., o problema do sub-aproveitamento de recursos referido anteriormente tende a agravar-se.

A utilização de reconfiguração dinâmica é complexa e obriga, muitas vezes, a reformular a solução do problema em análise, para se poderem aproveitar os benefícios da sua implementação. Pode argumentar-se que a capacidade crescente dos circuitos FPGA (10 milhões de portas lógicas equivalentes para os maiores circuitos actuais) disponibiliza tantos recursos simultaneamente que a utilização de reconfiguração dinâmica pode deixar de fazer sentido. Este argumento não tem em conta que a reconfiguração dinâmica permite aumentar a densidade funcional de uma implementação (cf. análise da secção 3.1, pág. 64) e, consequentemente, o trabalho útil por unidade de energia consumida. Aliás, a utilização de reconfiguração dinâmica na implementação sistemas digitais de baixo consumo de potência tem vindo a merecer uma atenção crescente.

Para além deste argumento geral, existem vários cenários em que a quantidade de recursos reconfiguráveis disponíveis é limitada. Num deles, o circuito FPGA, embora grande, dever ser partilhados por vários circuitos de processamento a operarem concorrentemente; neste caso, a área disponível para a implementação de cada um é limitada e pode ser interessante, da perspectiva do desempenho global, aumentar o índice de processamento concorrente à custa da área de implementação de cada processo concorrente. A redução de área poderia ser obtido

por recurso a configuração dinâmica.

Um outro cenário, que tem vindo a ganhar interesse crescente, prende-se com a utilização de circuitos reconfiguráveis como parte integrante de um sistema implementado num só circuito integrado (*system-on-a-chip—SOC*). Neste contexto, desenham-se actualmente duas abordagens diferentes, mas ambas acabam por redundar em oportunidades para o emprego de reconfiguração dinâmica.

Uma primeira tendência consiste em implementar o sistema completo num só circuito FPGA. Vários fabricantes (Xilinx, Actel) têm anunciado a intensão de incluir processadores nos seus circuitos FPGA, quer como blocos dedicados, quer por configuração dos recursos existentes. Conjuntamente com a capacidade de memória já existente (e com os restantes blocos especializados) pretendem desta forma posicionar os seus produtos como uma plataforma para a implementação de sistemas completos. Neste contexto, surge naturalmente a possibilidade de reservar uma parte da área disponível para circuitos adicionais (p.ex. unidades de processamento) a serem configurados apenas se necessário para o processamento em curso num dado momento. Como a área disponível para esta finalidade seria naturalmente limitada, a reconfiguração dinâmica surge como uma maneira de proceder ao seu eficiente aproveitamento.

A outra tendência consiste em incluir blocos lógicos reconfiguráveis no interior de sistemas implementados em circuitos integrados de aplicação específica, i.e. o projectista pode incluir circuitos FPGA como componente do CIAE que implementa o SOC. Embora apenas actualmente os fabricantes de circuitos integrados comecem a disponibilizar aos seus clientes esta possibilidade, a sua importância futura já foi reconhecida. Também aqui a reconfiguração dinâmica pode desempenhar um papel importante [Roy, 2001], desde que estejam disponíveis metodologias e ferramentas apropriadas. Para tal, terão que ser encontradas soluções para problemas similares aos abordados neste trabalho, mas desta feita no âmbito mais lato do projecto combinado de *hardware/software*.

9.2 Resumo do trabalho realizado

As múltiplas formas de aproveitar a reconfiguração dinâmica para implementação de sistemas digitais, bem como a complexidade inerente à aplicação desta técnica, tornam imprescindível a utilização de sistemas eficazes de apoio ao desenvolvimento. O sistema ReDiFlex, cuja concepção e implementação são apresentadas nesta dissertação, procura precisamente fornecer ferramentas automáticas de apoio às principais etapas de implementação, integrando-as num ambiente de desenvolvimento caracterizado pela interactividade e expansibilidade necessárias ao aproveitamento desta nova técnica de implementação.

O sistema ReDiFlex está orientado para o desenvolvimento interactivo de aplicações que usem um co-processador parcialmente reconfigurável e, simultaneamente, pretende consti-

tuir um veículo para mais actividades de investigação nesta área. No cenário mais comum, o seu utilizador é um especialista da área de aplicação, com conhecimentos de programação, mas não necessariamente experiência ampla em implementação de sistema digitais. Como se pretende oferecer um ambiente interactivo de desenvolvimento de aplicações que vá de encontro ao cariz exploratório das actividades de concepção nesta área, optou-se por integrar o sistema de desenvolvimento num ambiente de programação que já satisfizesse à partida esses objectivos. Para o protótipo actual foi escolhido um ambiente de programação baseado na linguagem Common Lisp.

A reconfiguração dinâmica precisa de ser efectuada num enquadramento que defina as regras de funcionamento e de utilização. A actual implementação do sistema ReDiFlex suporta um modelo funcional da especificação do comportamento do co-processador reconfigurável em que as mutações dos componentes obedecem a um conjunto de regras gerais bem definidas. Toda a utilização da infra-estrutura reconfigurável é feita em termos desse modelo. O modelo é baseado numa rede acíclica de operadores e constitui uma variante dos tradicionais diagramas de fluxo de dados. Cada operador contém um número bem definido de entradas e saídas. As ligações entre operadores são unidireccionais e têm uma capacidade bem definida (em bits), que deve corresponder à dimensão das entradas e saídas a que estão associadas. Um operador pode ter parâmetros que especificam aspectos particulares da sua funcionalidade e pode permitir aceder a informação sobre o seu estado interno (para implementações sequenciais). Além de operadores regulares, o modelo inclui operadores sem entradas (geradores) ou sem saídas (acumuladores).

O modelo suporta a reconfiguração dinâmica ao permitir a modificação em tempo de execução de alguns parâmetros dos operadores; cada novo valor pode dar origem a uma nova implementação da funcionalidade genérica associada ao operador. Para além disso, o sistema suporta contentores, que são operadores que disponibilizam alternadamente a funcionalidade de vários outros operadores compatíveis. Em qualquer altura, o contentor pode ser reconfigurado para assumir a personalidade de um dos elementos constituintes (o que inclui a geração dinâmica da configuração respectiva e das interligações entre os portos do elemento interno e do contentor). Estes dois mecanismos permitem uma modificação dinâmica controlada da implementação física subjacente.

As mutações da implementação são desencadeadas directamente pelo utilizador, o que permite fazer depender as reconfigurações de critérios arbitrariamente complexos a determinar em tempo de execução e que podem envolver naturalmente os resultados obtidos até à altura.

A definição de modelos pode ser efectuada de maneira integrada com a linguagem de programação do ambiente hospedeiro, sendo os modelos funcionais representados por objectos dessa linguagem, o que permite a sua utilização como quaisquer outros objectos do sistema de programação, i.e. podem ser usados como argumento ou resultado de funções, copiados, etc.

Dado um modelo funcional, o sistema gera uma implementação física baseada directamen-

te na sua estrutura. Em geral, a implementação de um modelo pode exceder a capacidade da infra-estrutura de *hardware*. A capacidade de reconfiguração é então usada para dar a ilusão de uma infra-estrutura muito maior (*hardware* virtual), multiplexando a infra-estrutura existente entre as implementações de partes do modelo global. A decomposição necessária pode ser feita sobre o próprio modelo funcional ou, em algumas situações, já sobre a implementação física. A decomposição funcional usa um método heurístico rápido, inicialmente descrito por Kaul e Vemuri [1998] e também adoptado por Cardoso e Neto [1999]. O sistema procede automaticamente à decomposição funcional sempre que necessário.

O mapeamento automático entre os domínios funcional e físico preserva a estrutura do modelo funcional, o que facilita a tarefa de estimar a influência do modelo funcional sobre a implementação (o que é importante quando se procede à decomposição funcional). Por outro lado, a implementação física adquire assim uma estrutura que torna as tarefas de colocação e encaminhamento mais simples e relativamente rápidas, o que é um objectivo importante do módulo encarregado do tratamento da informação física, cuja tarefa é determinar quais os recursos a usar (blocos lógicos, recursos de encaminhamento, sinais globais). O tratamento efectuado está especialmente adaptado à estrutura dos circuitos gerados durante a fase de mapeamento funcional/físico.

Cada operador do modelo funcional é mapeado num componente de nível físico. Cada componente inclui uma função encarregada de gerar a respectiva configuração interna e, para componentes reconfiguráveis dinamicamente, outra função encarregada de criar configurações posteriores (quer por alteração da configuração existente, que por criação de uma configuração completamente nova). As modificações de configuração nunca podem levar à alteração da geometria do componentes, i.e. das suas dimensões e das posições dos seus terminais. No caso dos contentores, a configuração é composta por duas partes: a configuração do subcomponente e as interligações entre os terminais do subcomponente e os terminais exteriores do agregador. Qualquer uma delas pode ser gerada dinamicamente quando necessário.

O sistema ReDiFlex faz automaticamente a colocação dos componentes e o encaminhamento das respectivas ligações. A colocação é efectuada por um procedimento heurístico baseado em pesquisa local de uma vizinhança gerada aleatoriamente e tem como objectivo determinar a posição de cada componente no interior da respectiva fatia. O principal objectivo é ter uma implementação com altura inferior ao limite imposto pela arquitectura do circuito FPGA; acessoriamente pretende-se que, no seu todo, o grupo ocupe a menor área possível. Se o grupo tiver uma largura superior à do circuito FPGA, pode ser decomposto em sub-grupos de forma a que cada um não ultrapasse os limites físicos do circuito.

A fase de colocação é seguida pelo encaminhamento das interligações. Esta tarefa processa as interligações entre cada par sucessivo de fatias de maneira independente. Para cada par é efectuada uma pesquisa exaustiva dos percursos que permitem realizar as interligações desejadas. A implementação é recursiva e utiliza *backtracking* para considerar todas as possibilidades, mas evitando prosseguir a pesquisa por alternativas para as quais se possa

determinar (baseado nos resultados da pesquisa até essa altura) que não conduzem a uma solução. Durante esta fase também é determinada a distância final entre as fatias.

O algoritmo de encaminhamento é de complexidade exponencial, mas como é aplicado a problemas de dimensão relativamente reduzida, esta complexidade não tem repercussões globalmente negativas sobre o desempenho do sistema, pelo menos para os modelos encontrados até ao momento. De qualquer forma, a implementação foi feita com o cuidado de permitir uma substituição do algoritmo de encaminhamento, se necessário. Uma variação deste algoritmo é usada para o encaminhamento no interior de contentores.

O sistema ReDiFlex inclui também um módulo pra controlar directamente a infra-estrutura reconfigurável. A sua principal tarefa é configurar o circuito FPGA XC6216 de acordo com os resultados do tratamento do modelo físico, sendo capaz de efectuar reconfigurações totais ou parciais. Este módulo também assegura o controlo dos sinais globais de relógio e de inicialização.

O sistema ReDiFlex inclui rotinas de baixo nível para aceder directamente aos blocos lógicos do co-processador (para configuração e para transferência de dados), controlo da origem e características do sinal de relógio, bem como inicialização da infra-estrutura reconfigurável.

O módulo de apoio à execução (*run-time support*) implementa as funções necessárias para controlar globalmente a operação do co-processador reconfigurável. A implementação destas funções combina os serviços dos restantes módulos para fornecer ao utilizador uma interface simples e bem integrada no ambiente de programação.

No sistema ReDiFlex a utilização de uma rede de operadores para processar informação é feita por funções escritas em Lisp, que controlam o comportamento temporal das implementações de maneira a serem obtidos os resultados especificados pelo modelo funcional. O controlo temporal do co-processador envolve sequenciar as operações de configuração, transferir os dados de entrada para o co-processador e retirar daí os resultados. No caso das redes hierárquicas produzidas por decomposição funcional, este processo deve ser repetido para cada sub-rede.

Ao nível da utilização é conveniente distinguir entre contexto escalar e vectorial. No primeiro caso, existe apenas um conjunto de dados de entrada a processar, enquanto no segundo se pretende tratar uma sequência de conjuntos de dados. O segundo caso é tratado eficientemente por uma estratégia baseada nos princípios de funcionamento dos circuitos tipo *pipeline*, minimizando o número de ciclos de relógio e de operações de reconfiguração necessários.

O sistema ReDiFlex dispõe, para cada um dos contextos, de várias funções de controlo da execução. As versões mais gerais podem ser usadas com qualquer tipo de rede simples, enquanto as versões especializadas tratam mais eficientemente certos casos particulares. O facto de a implementação do controlo de execução ser feito em *software* torna relativamente fácil a criação de versões especializadas das funções de controlo.

O utilizador não precisa de conhecer todos estes detalhes de implementação, já que o siste-

ma ReDiFlex lhe permite definir de forma simples funções de interface para cada rede. Assim, a implementação de cada rede é usada por invocação de uma função própria, o que permite ao utilizador encará-la como qualquer outra função de Lisp e abstrair do facto de a sua invocação implicar um conjunto complexo de actividades envolvendo o co-processador reconfigurável.

Outra tarefa assegurada pelo módulo de apoio à execução é a reconfiguração dinâmica de componentes, coordenando a criação das novas configurações de componentes (que, no caso da implementação de contentores, implica também o encaminhamento das ligações entre portos do componente interno e os portos externos do contentor) e a reconfiguração imediata do co-processador com a nova versão do componente. Desta forma, qualquer modificação de parâmetros de um operador é imediatamente reflectida na configuração do co-processador.

O módulo de apoio à execução disponibiliza ainda um conjunto de meios de apoio à depuração. Estes permitem obter uma traçagem das diversas etapas da utilização de uma rede, bem como seguir ciclo a ciclo a respectiva evolução. Também é possível interromper o processamento efectuado pelo co-processador quando certas condições são detectadas para proceder à inspecção interactiva do estado do sistema.

O sistema ReDiFlex inclui ainda ferramentas que permitem visualizar a estrutura de um modelo funcional (recorrendo ao programa VCG [Sander, 1995]) e, para o nível físico, os resultados das fases de colocação e encaminhamento (usando um programa especialmente desenvolvido para o efeito).

9.3 Perspectivas de trabalho futuro

O protótipo do sistema ReDiFlex serviu o duplo objectivo de veículo de investigação dos problemas associadas à utilização de reconfiguração dinâmica e de validação por implementação completa da abordagem escolhida. O esforço de implementação esteve focado nos aspectos mais centrais para a prossecução dos fins em vista. Como em todos os protótipos existem aspectos susceptíveis de melhoria.

Do ponto de vista do utilizador, seria interessante dispor de uma interface gráfica para definir as redes de operadores por manipulação directa de objectos gráficos. Também seria interessante dispor de uma ferramenta que apresentasse uma animação das diversas etapas do processamento, i.e. uma interface gráfica para o subsistema de traçagem e execução ciclo-a-ciclo.

Do ponto de vista da especificação do modelo, seria útil introduzir a especificação hierárquica de redes de operadores, ou o que seria equivalente, permitir descrever novos operadores por interligação de operadores definidos anteriormente. Esta característica é fundamental para um salto importante na complexidade dos sistemas que é possível definir facilmente.

Do lado da melhoria do desempenho, seria interessante automatizar a geração de versões especializadas das funções de controlo de execução, conforme já foi referido na secção 8.1. A

utilização da linguagem Common Lisp é particularmente favorável a este tipo de tarefas.

Outra maneira de melhorar o desempenho global das aplicações, consistiria em permitir a geração de configurações *off-line*. Por exemplo, para uma dada rede especificar-se-ia quais as combinações de valores dos parâmetros a considerar, e o sistema criaria automaticamente todas as configurações necessárias. Posteriormente, durante a execução, o sistema de controlo limitar-se-ia a colocar a configuração apropriada no co-processador (poderia ser um reconfiguração completa ou parcial).

Uma outra abordagem, similar a esta, mas transparente para o utilizador, consistiria em fazer com que os geradores de configurações memorizassem os resultados que vão produzindo, de maneira a que uma segunda invocação com os mesmos valores dos parâmetros reaproveitasse a configuração gerada anteriormente. De igual maneira, as funções de tratamento dos contentores poderiam memorizar as configurações já usadas. Esta abordagem permitiria melhorar facilmente o desempenho global de aplicações que geram um número limitado de configurações diferentes, que são usadas repetidamente.

Do ponto de vista do modelo físico, as melhorias passam por permitir a utilização de componentes multi-ciclo e pela flexibilização das restrições geométricas, em particular o agrupamento dos componentes em fatias. Neste contexto é importante investigar mais sistematicamente abordagens à colocação e encaminhamento que permitam obter rapidamente resultados de qualidade aceitável. Este problema, que é objecto de uma abordagem inicial por Tessier [1999], é tanto mais importante quanto mais cresce a complexidade interna dos circuitos FPGA e das respectivas redes de interligação.

Outro aspecto merecedor de atenção é a extensão do modelo funcional utilizado para incluir memória e entradas/saídas de dados directamente através dos pinos do co-processador. Com a existência de fontes de dados independentes do programa a executar no processador principal, ganha interesse a possibilidade de fazer com que o co-processador opere concorrentemente sobre os dados fornecidos por essas fontes. Seria preciso investigar outros modelos de comunicação entre processador e co-processador, em particular o uso de interrupções.

Ainda relacionado com a extensão do modelo funcional, merecem atenção as questões relacionadas com o levantamento da restrição de aciclicidade das redes de operadores. Em particular, é preciso avaliar se a complexidade adicional do modelo, tanto a nível conceptual como de implementação, é compensada pelo número acrescido de sistemas especificáveis. Uma resposta definitiva depende de uma análise detalhada do factores envolvidos e pode, inclusivamente, depender das áreas de aplicação.

Outro ponto a merecer atenção é a gestão das operações de reconfiguração, especialmente na presença de entradas/saídas directas do co-processador. Em particular, merece investigação a possibilidade de a gestão ser feita completa ou parcialmente em *hardware*. Entre as diferentes facetas deste problema a ter em conta, contam-se: (i) a maneira de especificar as actividades de controlo; (ii) sincronização com o co-processador; (iii) escolha entre um sistema de controlo fixo ou reconfigurável; (iv) repartição das tarefas de controlo entre *hardware* e *software*.

O estudo de modelos diferentes para sistemas reconfiguráveis dinamicamente é uma tarefa importante para a aplicabilidade e aceitação desta técnica. Para além da utilização de linguagens de programação convencionais (como, p.ex., em Cardoso e Neto [1999]), também merecem atenção modelos mais específicos. A título de exemplo, podem referir-se a especificação de sistemas reactivos por conjuntos de máquinas de estados finitas interactuantes, conforme usado no sistema de verificação VIS [Brayton *et al.*, 1996]. Neste cenário, as máquinas de estados seriam decompostas em sub-máquinas; em cada instante, apenas uma sub-máquina de cada máquina de estados interactuante estaria presente na infra-estrutura reconfigurável. Desta forma seria possível implementar sistema compostos por dezenas (ou mesmo centenas) de máquinas de estados. Também aqui seria possível implementar o sistema de controlo em *hardware* ou numa mistura de *hardware* e *software*. Como exemplo desta última, considere-se uma implementação em que a mudança de sub-máquina gera uma interrupção, cuja rotina de atendimento procede à reconfiguração necessária.

A utilização de reconfiguração dinâmica abre muitas perspectivas interessantes; aproveitá-las efectivamente e demonstrar a aplicabilidade desta técnica em grande escala necessitará ainda de muito trabalho futuro.

Referências

Abd-Alla, A. M. e Karlgaard, D. C. [1974] "Heuristic synthesis of microprogrammed computer architecture." *IEEE Trans. Computers*, volume C-23(8), págs. 802–807.

Alves, J. C. [1997] *Processadores Dedicados: Concepção e Realização em Sistemas Digitais Reconfiguráveis*. Tese de Doutoramento, Universidade do Porto, Portugal.

Alves, J. C., Ferreira, J. C., Albuquerque, C., Oliveira, J. F., Ferreira, J. S. e Matos, J. S. [1999] "Fafner–accelerating nesting problems with FPGAs." In Pocek e Arnold [1999], págs. 168–176.

Amerson, R., Carter, R. J., Culbertson, W. B., Kuekes, P. e Snider, G. [1995] "Teramac—configurable custom computing." In Athanas e Pocek [1995], págs. 32–38.

ANSI Common Lisp [1994] *Programming Language Common Lisp – ANSI/NCITS X3.226-1994*. National Committee for Information Technology Standards.

Athanas, P. e Pocek, K. L. (orgs.) [1995] *Proceedings The IEEE Symposium on FPGAs for Custom Computing Machines*.

Athanas, P. M. e Silverman, H. F. [1993] "Processor reconfiguration through instruction-set metamorphosis." *Computer*, volume 26(3), págs. 11–18.

Babb, J., Tessier, R. e Agarwal, A. [1993] "Virtual wires: Overcoming pin limitations in FPGA-based logic emulators." In Buell e Pocek [1993], págs. 142–151.

Ban, M. [1998] "Controlling run-time reconfigurable hardware designs with Java." In *Proceeings of the Programmable Logic Design Conference, DesignCon 98*, págs. 55–76.

Beazley, D. M. e Van Rossum, G. [1999] *Python Essential Reference*. New Riders Publishing.

Bellows, P. e Hutchings, B. [1998] "JHDL – an HDL for reconfigurable systems." In Pocek e Arnold [1998], págs. 175–184.

Betz, V. e Rose, J. [1996] "Directional bias and non-uniformity in FPGA global routing architectures." In *IEEE/ACM International Conference on Computer Aided Design: Digest of Tecnical Papers*, págs. 626–659.

— [1997] "Cluster-based logic blocks for FPGAs: Area-efficiency vs. input sharing and size." In *Proc. IEEE 1997 Custom Integrated Circuits Conference*, págs. 551–554. Santa Clara, CA, USA.

— [1998] "Effect of the prefabricated routing track distribution on FPGA area-efficiency." *IEEE Trans. on VLSI Sys.*, volume 6(3), págs. 445–456.

Bhat, N. B., Chaudhary, K. e Kuh, E. S. [1993] "Performance-oriented fully routable dynamic architecture for a field programmable logic device." Rel. téc., U. C. Berkeley.

Bittner, R. e Athanas, P. [1997] "Wormhole run-time reconfiguration." In FPGA'97.

Bittner, Jr., R. A. [1997] *Wormhole Run-Time Reconfiguration: Conceptualization and VLSI Design of A High Performance Computing System*. Tese de Doutoramento, Virginia Polytechnic Institute and State University, Blacksburg, VA.

Bjesse, P., Claessen, K., Sheeran, M. e Singh, S. [1998] "Lava: Hardware design in Haskell." In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, págs. 174–184.

Bondalapati, K. e Prasanna, V. K. [1999a] "DRIVE: An interpretative simulation and visualization environment for dynamically reconfigurable systems." In *International Workshop on Field Programmable Logic and Applications*.

— [1999b] "Dynamic precision management for loop computations on reconfigurable architectures." In Pocek e Arnold [1999], págs. 249–258.

— [1999c] "Hardware object selection for mapping loops onto reconfigurable architectures." In *Proc. International conference on Parallel and Distributed Processing Techniques and Applications*.

van den Bout, D. E., Morris, J. N., Thomae, D., Labrozzi, S., Wingo, S. e Hallman, D. [1992] "AnyBoard: An FPGA-based reconfigurable system." *IEEE Des. & Test Comp.*, volume 9(3), págs. 21–30.

Brayton, R. K., Hachtel, G. D., Sangiovanni-Vincentelli, A., Somenzi, F., Aziz, A., Cheng, S.-T., Edwards, S., Khatri, S., Kukimoto, Y., Pardo, A., Qadeer, S., Ranjan, R. K., Sarwary, S., Shiple, T. R., Swamy, G. e Villa, T. [1996] "VIS: A system for verification and synthesis." In Alur, R. e Henzinger, T. (orgs.), *Proceedings of the 8th International Conference on Computer Aided Verification*, Springer Lecture Notes in Computer Science, págs. 428–432.

Brebner, G. e Donlin, A. [1998] "Runtime reconfigurable routing." In Rolim [1998].

Brown, J., Chen, D., Eslick, I., Tau, E. e DeHon, A. [1995] "DELTA: Prototype for a first-generation dynamically programmable gate array." Rel. Téc. TN #112, M.I.T.

Brown, S. D., Francis, R. J., Rose, J. e Vranesic, Z. G. [1992] *Field-Programmable Gate Arrays*. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers. ISBN: 0-7923-9248-5.

Buell, D. A., Arnold, J. M. e Kleinfelder, W. J. (orgs.) [1996] *Splash 2: FPGAs in a Custom Computing Machine*. IEEE Computer Society Press.

Buell, D. A. e Pocek, K. L. (orgs.) [1993] *Proceedings The IEEE Workshop on FPGAs for Custom Computing Machines*.

— [1994] *Proceedings The IEEE Workshop on FPGAs for Custom Computing Machines*. IEEE Computer Society Press.

Burns, J., Donlin, A., Hogg, J., Singh, S. e de Wit, M. [1997] "A dynamic reconfiguration run-time system." In Pocek e Arnold [1997], págs. 66–75.

Butts, M., Batcheller, J. e Varghese, J. [1992a] "An efficient logic emulation system." In *Proceedings of the International Conference on Computer Design*, págs. 138–141.

— [1992b] "An efficient logic emulation system." In *Proc. IEEE International Conference on Computer Design*, págs. 138–141.

Cadambi, S., Weener, J., Goldstein, S. C., Schmit, H. e Thomas, D. E. [1998] "Managing pipeline-reconfigurable FPGAs." In FPGA'98.

Callahan, T. J. e Wawrzynek, J. [1998] "Instruction-level parallelism for reconfigurable computing." In Hartenstein e Keevallik [1998].

Cardoso, J. M. P. e Neto, H. [1999] "Macro-based hardware compilation of Java bytecodes into a dynamic reconfigurable computing system." In Pocek e Arnold [1999], págs. 2–11.

Carmichael, C. [1999] *Virtex FPGA Series Configuration and Readback*. Xilinx. Application note.

Casselman, S. [1993] "Virtual computing and the virtual computer." In Buell e Pocek [1993], págs. 43–48.

Chan, P. K., Schlag, M. D. F. e Martin, M. [1992] "BORG: A reconfigurable prototyping board using field-programmable gate arrays." In *ACM/SIGDA First International Symposium on Field-Programmable Gate Arrays*, págs. 47–51.

Chang, D. e Marek-Sadowska, M. [1999] "Partitioning sequential circuits on dynamically reconfigurable FPGAs." *IEEE Trans. on Comp.*, volume 48(6), págs. 565–578.

Chia, K.-N., Kim, H. J., Lansing, S., Mangione-Smith, W. H. e Villasenor, J. [1998] "High-performance automatic target recognition through data-specific VLSI." *IEEE Trans. on VLSI Sys.*, volume 6(3), págs. 364–371.

Chow, H. A., Alnuweiri, H. e Casselman, S. [1995] "FPGA-based transformable computers for fast digital signal processing." In Athanas e Pocek [1995], págs. 197–203.

Churcher, S., Kean, T. e Wilkie, B. [1995] "The XC6200 FastMap processor interface." In Moore e Luk [1995], págs. 36–43.

Clark, D. A. e Hutchings, B. L. [1996] "Supporting FPGA microprocessors through retargetable software tools." In Pocek e Arnold [1996], págs. 195–203.

Cox, C. E. e Blanz, W. E. [1992] "GANGLION—a fast field-programmable gate array implementation of a connectionist classifier." *IEEE J. Solid-State Cir.*, volume 27(3), págs. 288–299.

Cuccaro, S. A. e Reese, C. F. [1993] "The CM-2X: A hybrid CM-2/Xilinx prototype." In Buell e Pocek [1993], págs. 121–130.

Dahl, M., Babb, J., Tessier, R., Hanono, S., Hoki, D. e Agarwal, A. [1994] "Emulation of the Sparcle microprocessor with the MIT virtual wires emulation system." In Buell e Pocek [1994], págs. 14–22.

DATE'99 [1999] *Proceedings Design, Automation and Test in Europe*.

Dave, B. P. [1999] "CRUSADE: Hardware/software co-synthesis of dynamically reconfigurable heterogeneous real-time distributed embedded systems." In DATE'99, págs. 97–104.

DeHon, A. [1994] "DPGA-coupled microprocessors: Commodity ICs for the early 21st century." In Buell e Pocek [1994], págs. 31–39.

— [1995] "Notes on coupling processors with reconfigurable logic." Rel. Téc. TN #118, M.I.T.

— [1996a] "Reconfigurable architectures and general-purpose computing in the MOS VLSI era." Rel. Téc. TN #131, M.I.T.

— [1996b] *Reconfigurable Architectures for General-Purpose Computing.* Tese de Doutoramento, M.I.T.

El Gamal, A. [1981] "Two-dimensional stochastic model for interconnections in master slice integrated circuits." *IEEE Transactions on Circuits and Systems*, volume CAS-28(2), págs. 127–138.

El Gamal, A., Greene, J. e Roychowdhury, V. [1991] "Segmented channel routing is nearly as efficient as channel routing (and just as hard)." In *Proceedings of the Conference on Advanced Research in VLSI*, págs. 192–211.

Eldredge, J. G. e Hutchings, B. L. [1994] "RRANN: The run-time reconfiguration artificial neural network." In *Proceedings of the 1994 Custom Integrated Circuits Conference*, págs. 77–80.

— [1996] "Run-time reconfiguration: A method for enhancing the functional density of SRAM-based FPGAs." *Journal of VLSI Signal Processing*, volume 12, págs. 67–86.

Estrin, G., Russell, B., Turn, R. e Bibb, J. [1963] "Parallel processing in a restructurable computer system." *IEEE Transactions on Electronic Computers*, volume EC-12(12), págs. 747–755.

Faura, J., Aguirre, M. A., Moreno, J. M., van Duong, P. e Insenser, J. M. [1997a] "FIPSOC: A field programmable system on a chip." In Garcia Franquelo, L., Torralba, A. e Aguirre, M. A. (orgs.), *Proceedings of the XII Design of Circuits and Integrated Systems Conference DCIS'97*, págs. 597–602.

Faura, J., Moreno, J. M., Aguirre, M. A., van Duong, P. e Insenser, J. M. [1997b] "Multicontext dynamic reconfiguration and real time probing of a novel mixed signal programmable device with on-chip microprocessor." In *Proceedings Conference on Field Programmable Logic.*

Ferreira, J. C., Albuquerque, J. C. A. C., Oliveira, J. F., Ferreira, J. S. e Matos, J. S. [1998] "A flexible custom computing machine for nesting problems." In *Proceedings of the XIII Design of Circuits and Integrated Systems Conference DCIS'98*, págs. 686–691.

Fleischmann, J., Buchenrieder, K. e Kress, R. [1999] "Codesign of embedded systems based on Java and reconfigurable hardware components." In DATE'99, págs. 768–769.

Fornaciari, W. e Piuiri, V. [1998] "Virtual FPGAs: Some steps behind the physical barriers." In Rolim [1998].

FPGA'97 [1997] *Proceedings of the Fifth International Symposium on Field-Programmable Gate Arrays.*

FPGA'98 [1998] *Proceedings of the ACM/SIGDA Sixth International Symposium on Field-Programmable Gate Arrays.*

French, P. C. e Taylor, R. W. [1993] "A self-reconfiguring processor." In Buell e Pocek [1993], págs. 50–59.

Fujii, T., Furuta, K., Motomura, M., Nomura, M., Mizuno, M., Anjo, K., Wakabayashi, K., Hirota, Y., Nakazawa, Y., Ito, H. e Yamashina, M. [1999] "A dynamically reconfigurable logic engine with a multi-context/multi-mode unified-cell architecture." In *1999 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, págs. 364–365.

Gajski, D. D., Dutt, N. D., Wu, A. C.-H. e Lin, S. Y.-L. [1992] *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers.

Ganapathy, G., Narayan, R., Jorden, G., Fernandez, D., Wang, M. e Nishimura, J. [1996] "Hardware emulation for functional verification of K5." In *33rd Design Automation Conference Proceedings*.

de Garis, H., Korkin, M., Gears, F., Nawa, N. E. e Hough, M. [1999] "Building an artificial brain using an fpga-based CAM-brain machine." *Applied Mathematics and Computation Journal*.

Gokhale, M., Holmes, W., Kopser, A., Lucas, S., Minnich, R., Sweely, D. e Lopresti, D. [1991] "Building and using a highly parallel programmable logic array." *Computer*, volume 24(1), págs. 81–89.

Gokhale, M. e Minnich, R. [1993] "FPGA computing in a data parallel C." Rel. Téc. SRC-TR-93-097, Supercomputing Research Center, 17100 Science Dr., Bowie, Md 20715, USA.

Goldstein, S. C., Schmit, H., Moe, M., Budiu, M., Cadambi, S., Taylor, R. R. e Laufer, R. [1999] "PipeRench: A coprocessor for streaming multimedia acceleration." In *Proceedings of the 26th International Symposium on Computer Architecture*.

Guccione, S. A. e Gonzalez, M. J. [1993] "A data-parallel programming model for reconfigurable architectures." In Buell e Pocek [1993], págs. 79–87.

Gunther, B., Milne, G. e Narasimhan, L. [1996] "Assessing document relevance with run-time reconfigurable machines." In Pocek e Arnold [1996].

Guo, S. e Luk, W. [1995] "Compiling Ruby into FPGAs." In Moore e Luk [1995], págs. 188–197.

Hadley, J. D. [1995] *The Performance Enhancement of a Run-Time Reconfigurable FPGA System through Partial Reconfiguration*. Tese de Mestrado, Brigham Young University.

Hartenstein, R. W. e Keevallik, A. (orgs.) [1998] *Field-Programmable Logic and Applications: From FPGAs to Computing Paradigms*, volume 1482 de *Lecture Notes in Computer Science*. Springer-Verlag.

Hauck, S. [1995] *Multi-FPGA Systems*. Tese de Doutoramento, University of Washington.

— [1998a] "Configuration prefetch for single context reconfigurable coprocessors." In FPGA'98.

— [1998b] "The roles of FPGA's in reprogrammable systems." *Proc. of the IEEE*, volume 86(4), págs. 615–637.

Hauck, S., Borriello, G. e Ebeling, C. [1994] "Springbok: A rapid-prototyping system for board-level designs." In *ACM/SIGDA Second International Symposium on Field-Programmable Gate Arrays*.

Hauser, J. R. e Wawrzynek, J. [1997] "Garp: A MIPS processor with a reconfigurable coprocessor." In Pocek e Arnold [1997], págs. 12–21.

Higuchi, T., Iwata, M., Kajitani, I., Iba, H., Hirao, Y., Furuya, T. e Manderick, B. [1997] "Evolvable hardware and its applications to pattern recognition and fault-tolerant systems." In *Towards Evolvable Hardware: The Evolutionary Engineering Approach*, volume 1062 de *Lecture Notes in Computer Science*, págs. 118–135. Springer-Verlag.

Hoang, D. T. [1993] "Searching genetic databases on Splash 2." In Buell e Pocek [1993], págs. 185–191.

Hogg, J., Singh, S. e Sheeran, M. [1996] "New HDL research challanges posed by dynamically reprogrammable hardware." In *Third Asia Pacific Conference on Hardware Description Languages*.

Högl, H., Kugel, A., Ludvig, J., Männer, R., Noffz, K. H. e Zoz, R. [1995] "Enable++: A second generation FPGA processor." In Athanas e Pocek [1995], págs. 45–53.

Hudson, R. D., david I. Lehn e Athanas, P. M. [1998] "A run-time reconfigurable engine for image interpolation." In Pocek e Arnold [1998], págs. 88–95.

Hughes, K. e Gunther, B. K. [1998] "Viable run-time reconfiguration of hardware." In Morris, J. (org.), *Proc. ACAC'98: The 3rd Australasian Computer Architecture Conference*, págs. 67–74. Springer-Verlag Singapore.

Hutchings, B., Bellows, P., Hawkings, J., Hemmert, S., Nelson, B. e Rytting, M. [1999] "A CAD suite for high-performance FPGA design." In Pocek e Arnold [1999], págs. 12–24.

Hutchings, B. L. e Wirthlin, M. J. [1995] "Implementation approaches for reconfigurable logic applications." In Moore e Luk [1995], págs. 419–428.

I-Cube [1999] *IQX Family Data Sheet.*

Iseli, C. [1996] *A Reconfigurable Processor Development System.* Tese de Doutoramento, École Polytechnique Fédérale de Lausanne.

Jean, J., Liang, X., Drozd, B., Tomko, K. e Wang, Y. [2000] "Automatic target recognition with dynamic reconfiguration." *Journal of VLSI Signal Processing*, volume 25(1), págs. 39–53.

Jean, J. S. N., Tomko, K., Yavagal, V., Shah, J. e Cook, R. [1999] "Dynamic reconfiguration to support concurrent applications." *IEEE Trans. on Comp.*, volume 48(6), págs. 591–602.

Kaul, M., Srinivasan, V., Govindarajan, S., Ouaiss, I. e Vemuri, R. [1998] "Partitioning and synthesis for run-time reconfigurable computers using the SPARCS system." In *Military and Aerospace Applications of Programmable Devices and Technologies Conference.*

Kaul, M. e Vemuri, R. [1998] "Optimal temporal partitioning and synthesis for reconfigurable architectures." In *Proceedings Design, Automation and Test in Europe*, págs. 389–396.

— [1999] "Temporal partitioning combined with design space exploration for latency minimization of run-time reconfigured designs." In DATE'99, págs. 202–209.

Kaul, M., Vemuri, R., Govindarajan, S. e Ouaiss, I. [1999] "An automated temporal partitioning and loop fission approach for FPGA based reconfigurable synthesis for DSP applications." In *36th Design Automation Conference Proceedings*, págs. 616–622.

Kelem, S. [1999] *Virtex Configuration Architecture Advanced User's Guide*. Xilinx. Application note.

Keymeulen, D., Konaka, K., Iwata, M., Kuniyoshi, Y. e Higuchi, T. [1998] "Robot learning using gate-level evolvable hardware." In Birk, A. e Demiris, J. (orgs.), *Proceedings of the Sixth European Workshop on Learning robots*, Lecture Notes in Artificial Intelligence. Springer-Verlag.

Khalid, M. A. S. e Rose, J. [1995] "The effect of fixed I/O pin positioning on the routability and speed of FPGAs." Rel. Téc. CSRI-325, Computer Systems Research Institute, University of Toronto.

Kouloheris, J. L. e El Gamal, A. [1991] "FPGA performance versus cell granularity." In *Proceedings IEEE 1991 Custom Integrated Circuits Conference*, págs. 6.2.1–6.2.4.

Kumar, J., Strader, N., Freeman, J. e Miller, M. [1995] "Emulation verification of the Motorola 68060." In *Proceedings International Conference on Computer Design*, págs. 150–158.

Laufer, R., Taylor, R. R. e Schmit, H. [1999] "PCI-PipeRench and the SwordAPI: A system for stream-based reconfigurable computing." In Pocek e Arnold [1999], págs. 200–208.

Lemoine, E. e Merceron, D. [1995] "Run time reconfiguration of FPGA for scanning genomic databases." In Athanas e Pocek [1995], págs. 90–98.

Li, J. e Cheng, C.-K. [1995] "Routability improvement using dynamic interconnect architecture." In Athanas e Pocek [1995], págs. 61–67.

Liu, H. e Wong, D. F. [1998] "Network flow based circuit partitioning for time-multiplexed FPGAs." In *IEEE/ACM Digest of Technical Papers: International Conference on Computer Aided Design*, págs. 497–504.

— [1999] "A graph theoretic optimal algorithm for schedule compression in time-multiplexed FPGA partitioning." In *IEEE/ACM Digest of Technical Papers: International Conference on Computer Aided Design*, págs. 400–405.

Ludwig, S. H.-M. [1997] *Hades: Fast Hardware Synthesis Tools and a Reconfigurable Coprocessor*. Tese de Doutoramento, ETH Zürich. Diss. ETH No. 12276.

Luk, W., Andreou, P., Dervyshire, A., Dupont-De-Dinechin, F., Rice, J., Shirazi, N. e Siganos, D. [1998] "A reconfigurable engine for real-time video processing." In Hartenstein e Keevallik [1998], págs. 169–178.

Luk, W. e Guo, S. [1998] "Visualising reconfigurable libraries for FPGAs." In *Proc. 31 Asilomar Conference on Signals, Systems, and Computers*. IEEE Computer Society Press.

Luk, W. e McKeever, S. [1998] "Pebble: A language for parametrised and reconfigurable hardware design." In Hartenstein e Keevallik [1998], págs. 9–18.

Luk, W., Shirazi, N. e Cheung, P. Y. K. [1996] "Modelling and optimising run-time reconfigurable systems." In Pocek e Arnold [1996], págs. 167–176.

— [1997] "Compilation tools for run-time reconfigurable designs." In Pocek e Arnold [1997], págs. 56–65.

Lysaght, P., Dick, H., McGregor, G., McConnell, D. e Stockwood, J. [1995] "Prototyping environment for dynamically reconfigurable logic." In Moore e Luk [1995], págs. 409–418.

Lysaght, P. e Dick, H. P. [1994] "Implementation of adaptive signal processing architectures based on dynamically reconfigurable FPGAs." In *Proceedings of VII European Signal Processing Conference*, volume III, págs. 1871–1874.

Lysaght, P. e Dunlop, J. [1993] "Dynamic reconfiguration of field programmable gate arrays." In Moore, W. e Luk, W. (orgs.), *More FPGAs: Proceedings of the Third International Workshop on Field Programmable Logic and Applications*. Oxford, England.

Lysaght, P. e Stockwood, J. [1996] "A simulation tool for dynamically reconfigurable field programmable gate arrays." *IEEE Trans. on VLSI Sys.*, volume 4(3), págs. 381–390.

Mackinlay, P. I., Cheung, P. Y. K., Luk, W. e Sandiford, R. [1997] "Riley-2: A flexible platform for codesign and dynamic reconfigurable computing research." In Luk, W., Cheung, P. Y. K. e Glesner, M. (orgs.), *Field-Programmable Logic and Applications*, volume 1304 de *Lecture Notes in Computer Science*, págs. 91–100. Springer-Verlag.

Marple, D. [1992] "An MPGA-like FPGA." *IEEE Des. & Test Comp.*, volume 9(2), págs. 51–60.

McGregor, G., Robinson, D. e Lysaght, P. [1998] "A hardware/software co-design environment for reconfigurable logic systems." In Hartenstein e Keevallik [1998].

McKay, N. e Singh, S. [1998] "Dynamic specialisation od XC6200 FPGAs by partial evaluation." In Hartenstein e Keevallik [1998], págs. 298–307.

de Micheli, G. [1994] *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, Inc., 1 ed.

Minnick, R. C. [1967] "A survey of microcellular research." *Journal of the ACM*, volume 14(2), págs. 203–241.

Moe, M., Schmit, H. e Goldstein, S. C. [1998] "Characterization and parametrization of a pipeline reconfigurable FPGA." In Pocek e Arnold [1998], págs. 294–295.

Moore, W. e Luk, W. (orgs.) [1995] *Field Programmable Logic and Applications*, volume 975 de *Lecture Notes in Computer Science*. Springer-Verlag.

Moreno, J. M., Madrenas, J., Faura, J., Cantó, E., Cabestany, J. e Insenser, J. M. [1998] "Feasible evolutionary and self-repairing hardware by means of the dynamic recionfiguration capabilities of the FIPSOC devices." In Sipper, M., Mange, D. e Pérez-Uribe, A. (orgs.), *Evolvable Systems: From Biology to Hardware*, volume 1478 de *Lecture Notes in Computer Science*, págs. 354–355. Springer-Verlag. Proceedings of the Second International Conference on Evolvable Systems ICES'98.

Nawa, N. E., Korkin, M. e de Garis, H. [1998] "ATR's CAM-brain project: The evolution of large-scale recurrent neural network modules." In *Proceedings International Conference on Parallel and Distributed Processing Techniques and Applications*.

Oldfield, J. V. e Dorf, R. C. [1995] *Field Programmable Gate Arrays: Reconfigurable Logic for Rapid Prototyping and Implementation of Digital Systems*. Wiley-Interscience. ISBN: 0-471-55665-3.

Ousterhout, J. K. [1994] *Tcl and the Tk Toolkit*. Addison-Wesley.

Papachristou, C. A. e Immaneni, V. R. [1993] "Vertical migration of software functions and algorithms using enhanced microsequencing." *IEEE Trans. Computers*, volume 42(1), págs. 45–61.

Paulin, P. G. e Knight, J. P. [1989] "Force-directed scheduling for the behavioral synthesis of ASIC's." *IEEE Trans. on Computer-Aided Design*, volume 8(6), págs. 661–679.

Pocek, K. L. e Arnold, J. M. (orgs.) [1996] *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines.*

— [1997] *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines.*

— [1998] *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines.*

— [1999] *Proceedings IEEE Symposium on Field-Programmable Custom Computing Machines.*

Purna, K. M. G. e Bhatia, D. [1999] "Temporal partitioning and scheduling data flow graphs for reconfigurable computers." *IEEE Trans. on Comp.*, volume 48(6), págs. 579–590.

Quénot, G. M., Kraljić, I., Sérot, J. e Zavidovique, B. [1994] "A reconfigurable compute engine for real-time vision automata prototyping." In Buell e Pocek [1994], págs. 91–100.

Raimbault, F., Lavenier, D., Rubini, S. e Pottier, B. [1993] "Fine grain parallelism on a MIMD machine using FPGAs." In Buell e Pocek [1993], págs. 2–8.

Rath, K. e Li, J. [1998] "Synthesizing reconfigurable sequential machines using tabular models." In Rolim [1998].

Ratha, N. K., Jain, A. K. e Rover, D. T. [1995] "Convolution on splash 2." In Athanas e Pocek [1995], págs. 204–213.

Rauscher, T. G. e Agrawala, A. K. [1978] "Dynamic problem-oriented redefinition of computer architecture via microprogramming." *IEEE Trans. Computers*, volume C-27(11), págs. 1006–1014.

Razdan, R. [1994] *PRISC: Programmable Reduced Instruction Set Computers.* Tese de Doutoramento, Harvard University.

Reddi, S. S. e Feustel, E. A. [1978] "A restructurable computer system." *IEEE Trans. Computers*, volume C-27(1), págs. 1–20.

Rivest, R. L., Shamir, A. e Adleman, L. [1978] "A method for obtaining digital signatures and public-key cryptosystems." *Communications of the ACM*, volume 21(2), págs. 120–126.

Robinson, D., McGregor, G. e Lysaght, P. [1998] "New CAD framework extends simulation of dynamically reconfigurable logic." In Hartenstein e Keevallik [1998], págs. 1–8.

Rolim, J. (org.) [1998] *Parallel and Distributed Processing*, nº 1388 in Lecture Notes in Computer Science. Springer-Verlag.

Rose, J. e Brown, S. [1991] "Flexibility of interconnection structures for field-programmable gate arrays." *IEEE J. Solid-State Cir.*, volume 26(3), págs. 277–282.

Rose, J., El Gammal, A. e Sangiovanni-Vincentelli, A. [1993] "Architecture of field-programmable gate arrays." *Proceedings of the IEEE*, volume 81(7), págs. 1013–1029.

Rose, J., Francis, R. J., Lewis, D. e Chow, P. [1990] "Architecture of field-programmable gate arrays: The effect of logic block functionality on area efficiency." *IEEE J. Solid-State Cir.*, volume 25(5), págs. 1217–1225.

Rosenberg, J. [1995] "Implementing cache logic with FPGAs." Application note, Atmel Corporation.

Roy, K. [2001] "Adding reconfigurable logic to SOC devices." *IEEE Des. & Test Comp.*, volume 18(4), págs. 65–71.

Rubini, S. [1995] *Intégration des architectures logiques reconfigurables dans les architectures parallèles MIMD.* Tese de Doutoramento, Université de Rennes 1.

Sait, S. M. e Youssef, H. [1995] *VLSI Physical Design Automation: Theory and Practice.* McGraw-Hill Book Company.

Sami, M. e Stefanelli, R. [1986] "Reconfigurable architectures for VLSI processing arrays." *Proc. IEEE*, volume 74(5), págs. 712–722.

Sander, G. [1995] "VCG–visualization of compiler graphs." Rel. Téc. A01-95, Universität des Saarlandes.

Schmit, H. [1997] "Incremental reconfiguration for pipelined applications." In Pocek e Arnold [1997], págs. 47–55.

Schmit, H. e Thomas, D. [1995] "Hidden markov modeling and fuzzy controllers in FPGAs." In Athanas e Pocek [1995], págs. 214–221.

Seitz, C. L. [1984] "Concurrent VLSI architectures." *IEEE Trans. Computers*, volume C-33(12), págs. 1247–1265.

Shirazi, N., Luk, W. e Cheung, P. Y. K. [1998a] "Automating production of run-time reconfigurable designs." In Pocek e Arnold [1998], págs. 147–156.

— [1998b] "Run-time management of dynamically reconfigurable designs." In Hartenstein e Keevallik [1998], págs. 59–68.

Singh, S. e Bellec, P. [1994] "Virtual hardware for graphics applications using FPGAs." In Buell e Pocek [1994], págs. 49–58.

Singh, S., Hogg, J. e McAuley, D. [1996] "Expressing dynamic reconfiguration by partial evaluation." In Pocek e Arnold [1996], págs. 188–194.

Singh, S., Rose, J., Chow, P. e Lewis, D. [1992] "The effect of logic block architecture on FPGA performance." *IEEE J. Solid-State Cir.*, volume 27(3), págs. 281–287.

Sipper, M., Sanchez, E., Mange, D., Tomassini, M., Pérez-Uribe, A. e Stauffer, A. [1997] "A phylogenetic, ontogenetic, and epigenetic view of bio-inspired hardware systems." *IEEE Trans. on Evolutionary Computation*, volume 1(1), págs. 83–97.

Skahill, K. [1995] *VHDL for Programmable Logic.* Cypress Semiconductor Corporation.

Smith, D. e Bhatia, D. [1996] "RACE: Reconfigurable and adaptive computing environment." In Hartenstein, R. W. e Glesner, M. (orgs.), *Field-Programmable Logic: Smart Applications, New Paradigms and Compilers*, volume 1142 de *Lecture Notes in Computer Science*, págs. 87–95. Springer-Verlag.

Steele, Guy L., J. [1990] *Common Lisp – The Language*. Digital Press, 2 ed.

Susanto, K. W. e Melham, T. [1998] "Formally analysed dynamic synthesis of hardware." In *Theorem Proving in Higher Order Logics: Emerging Trends, Supplementary Proceedings*, págs. 105–117.

Tau, E., Chen, D., Eslick, I., Brown, J. e DeHon, A. [1995] "A first generation DPGA implementation." Rel. Téc. TN #114, M.I.T.

Tempesti, G. [1998] *A Self-Repairing Multiplexer-Based FPGA Inspired by Biological Processes*. Tese de Doutoramento, EPFL.

Tessier, R. G. [1999] *Fast Place and Riute Approaches for FPGAs*. Tese de Doutoramento, M.I.T.

Thompson, A. [1996] "Silicon evolution." In Koza, J. R., Goldberg, D. E., Fogel, D. B. e Riolo, R. L. (orgs.), *Genetic Programming 1996: Proceedings of the First Anual Conference*, págs. 444–452. MIT Press.

Trimberger, S. [1998] "Scheduling designs into a time-multiplexed FPGA." In FPGA'98, págs. 153–160.

Trimberger, S., Carberry, D., Johnson, A. e Wong, J. [1997] "A time-multiplexed FPGA." In Pocek e Arnold [1997], págs. 22–28.

Triscend [1998] *Triscend E5 Configurable Processor Family*. Triscend.

Varghese, J., Butts, M. e Batcheller, J. [1993] "An efficient logic emulation system." *IEEE Trans. on VLSI Sys.*, volume 1(2), págs. 171–174.

Vasilko, M. [1999] "DYNASTY: A temporal floorplanning based CAD framework for dynamically reconfigurable logic systems." In Lysaght, P., Irvine, J. e Hartenstein, R. (orgs.), *Field-Programmable Logic and Applications*, volume 1673 de *Lecture Notes in Computer Science*, págs. 124–133. Springer-Verlag.

Vasilko, M. e Cabanis, D. [1999] "Improving simulation accuracy in design methodologies for dynamically reconfigurable logic systems." In Pocek e Arnold [1999], págs. 123–133.

Vasilko, M. e Long, D. [1998a] "Design of a prototyping system for high-speed dynamically reconfigurable logic." In *Proceedings 8th Annual Advanced PLD & FPGA Conference and Exhibition*, págs. 208–219.

— [1998b] "RIFLE-62: A flexible environment for prototyping dynamically reconfigurable systems." In *Proc. 9th Int. Workshop on Rapid System Prototyping*, págs. 130–135.

VCC [1997] *H.O.T. Works*. Virtual Computer Corporation, 1 ed.

Vick, C. R., Kartashev, S. P. e Kartashev, S. I. [1980] "Adaptable architectures for supersystems." *Computer*, volume 13(11), págs. 17–35.

Villasenor, J., Jones, C. e Schoner, B. [1995] "Video communications using rapidly reconfigurable hardware." *IEEE Trans. on Circ. and Sys. for Video Technol.*, volume 5(6), págs. 565–567.

Vuillemin, J. [1998] "Reconfigurable systems: Past and next 10 years." In Palma, J. M. L. M., Dongarra, J. e Hernandez, V. (orgs.), *Vector and Parallel Processing–VECPAR'98*, nº 1573 in Lecture Notes in Computer Science, págs. 519–539. Springer.

Vuillemin, J. E., Bertin, P., Roncin, D., Shand, M., Touati, H. H. e Boucard, P. [1996] "Programmable active memories: Reconfigurable systems come of age." *IEEE Trans. on VLSI Sys.*, volume 4(1), págs. 56–69.

Wirthlin, M. J. e Hutchings, B. L. [1995] "A dynamic instruction set computer." In Athanas e Pocek [1995], págs. 99–107.

— [1997] "Improving functional density through run-time constant propagation." In FPGA'97, págs. 86–92.

— [1998] "Improving functional density using run-time circuit reconfiguration." *IEEE Trans. on VLSI Sys.*, volume 6(2), págs. 247–256.

Wittig, R. D. e Chow, P. [1996] "OneChip: An FPGA processor with reconfigurable logic." In Pocek e Arnold [1996], págs. 126–135.

Woolfries, N., Lysaght, P., Marshall, S., McGregor, G. e Robinson, D. [1998] "Fast adaptive image processing in FPGAs using stack filters." In Hartenstein e Keevallik [1998].

Xilinx [1997] *XC6200 Field Programmable Gate Arrays*. Xilinx.

Yao, X. e Higuchi, T. [1996] "Promises and challenges of evolvable hardware." In *Proceedings First International Conference on Evolvable Systems: From Biology to Hardware*. Tsukuba, Japan.

Yeh, D., Heygin, G. e Chow, P. [1996] "RACER: A reconfigurable constraint-length 14 Viterbi decoder." In Pocek e Arnold [1996], págs. 60–69.

A Instalação do sistema ReDiFlex

Este apêndice descreve sumariamente a forma de obter uma versão do sistema ReDiFlex e ferramentas associadas.

Acesso

O repositório do sistema ReDiFlex pode ser acedido através do endereço `http://www.fe.up.pt/~jcf/rediflex`.

Requisitos básicos

Para usar a versão actual do sistema ReDiFlex é necessário dispor de um computador pessoal com o sistema operativo GNU/Linux (versão 2.0.29) e uma placa H.O.T. Works da companhia Virtual Computer Corporation.

O controlador da placa apenas funciona com núcleos da série 2.0.X; núcleos mais recentes gerem o barramento PCI de maneira diferente, pelo que a versão disponível não pode ser usada sem modificações.

O sistema ReDiFlex está configurado para usar o ambiente de Common Lisp designado por CLISP, disponível em `http://clisp.cons.org`. Para gestão dos ficheiros que compõem o sistema, é necessário ter instalada o pacote `defsystem`. Este pacote pode ser obtido em `http://sourceforge.net/projects/clocc`.

Para usar a ferramenta de visualização do modelo físico (v6200) é necessário ter instalado o sistema `Tk` (qualquer versão igual ou superior a 8.0 pode ser usada).

A visualização do modelo funcional utiliza o programa `vcg` disponível em `http://rw4.cs.uni-sb.de/users/sander/html/gsvcg1.html`.

Procedimento de instalação

A versão pré-compilada do sistema ReDiFlex e de todas os restantes programas necessários está contida no arquivo `rediflex-0.1-bin.tar.gz` existente no repositório. Esta versão usa a versão 5 da biblioteca `libc`. Após proceder ao desarquivamento, basta executar o programa `install` que está situado no topo da directoria criada.

A versão pré-compilada contém o ambiente CLISP e as bibliotecas C/C++ de acesso ao *hardware*. É ainda necessário compilar o controlador de dispositivo, o que exige proceder à reconfiguração do núcleo. As instruções para esta tarefa, que requer intervenção manual do utilizador, estão incluídas no arquivo de distribuição.

Quem pretender compilar todas as ferramentas a partir do código fonte deve obter o ficheiro

`rediflex-0.1-src.tar.gz` do mesmo repositório. Este arquivo contém o código fonte do sistema Rediflex, do ambiente de programação CLISP, do pacote `defsystem`, das bibliotecas C/C++ e do controlador de dispositivos. Para instalar o sistema, devem executar-se sucessivamente os programas `build` e `install` localizados no directório onde se proceder ao desarquivamento do sistema. Também neste caso é necessário intervenção manual para compilar o controlador de dispositivo.

Teste da instalação

Em primeiro lugar é necessário assegurar que o controlador de dispositivo está presente. Para fazer o núcleo carregar o controlador, executa-se o comando:

```
$ insmod x6200.0
```

De seguida, executa-se o ambiente de programação e carrega-se o sistema ReDiFlex:

```
$ clisp
[1]> (mk:oos :drh :load)
...
[2]> (drh:init-vcc-session)
```

Após estes passos, o sistema ReDiFlex está activo e pode ser usado. Por exemplo, para executar alguns testes pré-definidos pode usar-se o comando:

```
[3]> (drh:do-standard-tests)
```

Se as ferramentas de visualização se encontrarem instaladas, pode usar-se o comando:

```
[3]> (drh:do-standard-tests :visual t)
```

O sistema ReDiFlex pode ser usado vantajosamente a partir do editor Emacs, o que é particularmente facilitado pelo pacote ILISP (disponível em `http://sourceforge.net/projects/ilisp`), que gere a interacção entre o editor e o ambiente de programação CLISP.

B Breve introdução à linguagem Common Lisp

Este apêndice faz uma apresentação muito breve de alguns aspectos da linguagem Common Lisp com o objectivo de facilitar a compreensão dos fragmentos do código fonte do sistema ReDiFlex que são apresentados no corpo da dissertação. A informação contida neste apêndice não é suficiente para programar em Lisp.

A maneira mais simples de ler código Lisp é proceder como se os parêntesis não fossem visíveis. Quando o código fonte está devidamente formatado, o seu arranjo mostra de forma imediata grande parte da respectiva estrutura.

Lisp é uma linguagem baseada em expressões, apresentadas sempre notação pré-fixa (i.e., o operador antecede operandos). Uma expressão pode ser atómica ou uma lista. Números, símbolos, cadeias de caracteres são exemplos de expressões atómicas. Uma lista a ser sujeita ao processo de avaliação pode representar uma aplicação funcional ou uma forma especial (*special form*). Uma aplicação funcional tem um símbolo que indica a operação a executar seguido dos argumentos necessários:

```
(+ 2 3 4) ⇒9
(* 6 9 (/ 3 4)) ⇒ 81/2
```

Uma forma especial é uma lista cujo primeiro elemento é operador especial. A expressão é avaliada de acordo com as regras particulares do operador usado. Por exemplo o operador condicional `if` é um operador especial. A expressão

```
(if (test-something) (expr-1) (expr-2))
```

retorna o resultado de expressão `(expr-1)` se a expressão `(test-something)` retornar um resultado verdadeiro (i.e. diferente do valor `nil`) ou o resultado de `(expr-2)` no caso contrário. Em qualquer dos casos, apenas uma das expressões `(expr-1)` ou `(expr-2)` é avaliada.

A forma especial `defun` define novas funções. O início da definição pode ter uma cadeia de caracteres de documentação (*documentation string*). Estas podem também surgir em outros contextos, por exemplo na definição de estruturas de dados. Exemplo de definição de uma função:

```
(defun double (x)
  "Multiplica argumento X por 2."
  (* 2 x))
```

```
(double 3) ⇒ 6
```

O resultado de uma função é determinado pela última expressão do seu corpo.

Em Common Lisp, funções são objectos de primeira classe e podem ser usadas como argumentos de outras funções ou retornadas como resultados. Por exemplo, a função `mapcar` aplica uma função a cada elemento de uma lista e retorna uma lista com os resultados.

```
(mapcar #'double '(1 2 3)) ⇒ (2 4 6)
```

É possível introduzir funções anónimas com a forma especial `lambda`. Por exemplo, o próximo exemplo define uma função anónima que triplica o seu argumento e aplica-a ao valor 10.

```
((lambda (x) (* 3 x)) 10) ⇒ 30
```

O próximo exemplo mostra como uma função pode retornar um resultado funcional e como este pode ser usado. A forma especial `setf` atribui um valor a uma variável. A função `apply` aplica uma função aos seus argumentos. O carácter ";" inicia um comentário que termina no fim da linha.

```
(defun double-or-treble (arg)
  (if (evenp arg)          ; arg é par?
      (lambda (x) (* 2 x))
      (lambda (x) (* 3 x))))

(setf func-a (double-or-treble 5)) ⇒ func-object
(setf func-b (double-or-treble 4)) ⇒ another-func-object
(apply func-a 4) ⇒ 12
(apply func-b 4) ⇒ 8
```

Os argumentos de funções podem ter valores por omissão; também é possível referir os argumentos por nome ao invocar a função.

Common Lisp suporta ainda funções que modificam o código de expressões antes de estas serem avaliadas. Estas funções designam-se por *macros*. Macros são usados para expandir as capacidades da linguagem. Por exemplo, Common Lisp não tem uma forma que permita avaliar um conjunto de expressões repetidamente enquanto uma condição for verdadeira (o equivalente a `while` em outras linguagens). O idioma correspondente em Lisp é:

```
(loop (unless (test-cond) (return nil))
      (expr-1)
      ...
      (expr-n))
```

em que a expressão (return nil) força a terminação do ciclo. Uma macro pode ser usada para transformar uma expressão escrita em termos de while numa expressão em termos de loop. Depois da seguinte definição (para a compreensão geral não é preciso atender às plicas, vírgulas ou símbolo @ que surgem na definição)

```
(defmacro while (test &rest body)
  '(loop (unless ,test (return nil))
         ,@body))
```

a rotina de leitura de expressões de Lisp transforma transparentemente a expressão

```
(while (< i 10)      ; supondo que i é inicialmente igual a 7
   (print (* i i))
   (setf i (+ i 1))) ⇒
49
64
81
NIL
```

na expressão equivalente

```
(loop (unless (<i 10) (return nil))
      (print (* i i))
      (setf i (+ i 1)))
```

É esta última expressão que o interpretador ou compilador de Lisp "vêem", e não a expressão inicial. A utilização de macros permite expandir de forma muito flexível a linguagem aceite pelo sistema.

Em Common Lisp, os símbolos pertencem a "pacotes" (*packages*). A especificação completa de uma símbolo tem a forma package:symbol. Símbolo que pertencem ao pacote especial KEYWORD não precisam de especificar o nome do pacote, i.e., :sym e keyword:sym são equivalentes. Em qualquer ponto do processamento existe um pacto activo. Quando o pacote não é especificado explicitamente, presume-se que o símbolo pertence ao pacote activo.

A forma especial defstruct define um grupo de dados relacionados (corresponde a record na linguagem Pascal ou struct em linguagem C) e define automaticamente funções para criar objectos desse tipo e para aceder aos seus componentes.

```
(defstruct nome
   inicial
   (medio nil)   ; valor inicial deste componente é NIL
   final)

(setf b (make-nome :inicial 'Amadeu :final 'Zacarias)) ⇒
```

```
    #S(NOME :PRIMEIRO AMADEU :ULTIMO ZACARIAS)

(nome-primeiro b) ⇒ AMADEU
(nome-medio b) ⇒ NIL
(nome-final b) ⇒ ZACARIAS
(setf (nome-medio b) 'Lopes) ⇒ LOPES
b ⇒ #S(NOME :PRIMEIRO AMADEU :MEDIO LOPES :ULTIMO ZACARIAS)
```

Common Lisp permite um estilo flexível de programação orientada aos objectos. O subsistema que permite este estilo de programação é designado por CLOS (*Common Lisp Object System*). A macro `defclass` é usada para definir novas classes:

```
(defclass conta ()
  ((nome :initarg :nome :reader nome)
   (saldo :initarg :saldo :initform 0.00 :accessor saldo)
   (taxa-de-juro :allocation :class :initform 0.05 :reader taxa-de-juro)))
```

Neste exemplo, a classe `conta` tem três campos (*slots*): `nome`, `saldo` e `taxa-de-juro`. O nome de cada campo é seguido de pares palavra-chave/valor que definem a maneira de usar esse campo. Para o campo `nome`, o significado das palavras-chave é o seguinte: a palavra-chave `:initarg` indica que pode ser especificado um valor para o campo aquando da criação de um objecto; a palavra-chave `:reader` indica que o sistema deve criar um método de acesso para leitura chamada `nome`. A utilização do mesmo símbolo para identificar um campo e um método não causa confusão, porque o objecto referido pode ser inferido da forma de utilização. Em terminologia Lisp, um método é uma função especializada para determinado tipo de argumento, neste caso para objectos da classe `conta`.

A palavra-chave `:initform` usada na especificação do campo `saldo` indica que este deve ter um valor inicial de 0.00, se nada de diferente for especificado durante a criação. Este campo também tem um método de acesso, neste caso para leitura e escrita (especificado pelo argumento a seguir a `:accessor`).

O campo `taxa-de-juro` é partilhada por todos os objectos da classe, já que o tipo de alocação de memória especificado é `:class`. Esta classe pode ser usada da seguinte forma:

```
(setf a1 (make-instance 'conta :saldo 10000.00 :nome "Tavares"))
(nome a1) ⇒ "Tavares"
(saldo a1) ⇒ 10000.0
(taxa-de-juro a1) ⇒ 0.05
```

Ao contrário de outros sistemas orientados aos objectos, em CLOS os métodos são definidos separadamente das classes. Por exemplo, para efectuar um levantamento, podemos definir o seguinte método:

```
(defmethod efectuar-levantamento ((c conta) quantia)
  (if (< quantia (saldo c))
      (setf (saldo c) (- (saldo c) quantia))
      'fundos-insuficientes))
```

É simples definir uma classe derivada de conta, por exemplo conta-limitada, e um método correspondente para efectuar levantamentos:

```
(defclass conta-limitada (conta)
  ((limite :initarg :limite :reader limite)))

(defmethod efectuar-levantamento ((c conta-limitada) quantia)
  (if (> quantia (saldo c))
      'acima-do-limite
      (call-next-method)))
```

A chamada (call-next-method) é usada para invocar o método efectuar-levantamento da classe conta. Todos os métodos definidos para conta também podem ser usados com conta-limitada, com excepção de efectuar-levantamento que tem uma versão própria.

```
(setf a2 (make-instance 'conta-limitada
                        :nome "Alves dos Reis"
                        :saldo 5000.0 :limite 1000.0))
(nome a2) ⇒ "Alves dos Reis"
(efectuar-levantamento a2 2000.0) ⇒ ACIMA-DO-LIMITE
(efectuar-levantamento a2 700.0) ⇒ 4300.0
```

Em geral, podem existir vários métodos aplicáveis a um dado objecto. Nesse caso, todos os métodos são reunidos e ordenados, com o mais específico em primeiro lugar. O método mais específico é então invocado; esse método pode invocar os restantes através da função call-next-method.